UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMATICA

TESIS DOCTORAL

User-defined execution relaxations for enhanced programmability in high-performance parallel computing

Relajaciones de ejecución definidas por el usuario para la mejora de la programabilidad en computación paralela de altas prestaciones

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Andrés Antón Rey Villaverde

Directores

Francisco Daniel Igual Peña
Manuel Prieto Matías

Madrid

User-defined Execution Relaxations for Enhanced Programmability in High-Performance Parallel Computing

–

Relajaciones de Ejecución Definidas por el Usuario para la Mejora de la Programabilidad en Computación Paralela de Altas Prestaciones

TESIS DOCTORAL

Andrés Antón Rey Villaverde

Dirigida por:

Francisco Daniel Igual Peña y Manuel Prieto Matías

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2019

User-defined Execution Relaxations for Enhanced Programmability in High-Performance Parallel Computing

–

Relajaciones de Ejecución Definidas por el Usuario para la Mejora de la Programabilidad en Computación Paralela de Altas Prestaciones

Memoria que presenta para optar al título de Doctor en Informática

Andrés Antón Rey Villaverde

Dirigida por los Doctores

Francisco Daniel Igual Peña y **Manuel Prieto Matías**

Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2019

UNIVERSIDAD COMPLUTENSE MADRID

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña. Andrés Antón Rey Villaverde,
estudiante en el Programa de Doctorado Doctorado en Ingeniería Informática (RD99/2011),
de la Facultad de Informática de la Universidad Complutense de Madrid, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

User-defined Execution Relaxations for Enhanced Programmability in High-Performance Parallel Computing

Relajaciones de Ejecución Definidas por el Usuario para la Mejora de la Programabilidad en Computación Paralela de Altas Prestaciones

y dirigida por: Francisco Daniel Igual Peña y Manuel Prieto Matías

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita.

Del mismo modo, asumo frente a la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada de conformidad con el ordenamiento jurídico vigente.

En Madrid, a 3 de octubre de 2019

Fdo.: 

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

I hereby declare that all the content presented in this thesis entitled "User-defined Execution Relaxations for Enhanced Programmability in High-Performance Parallel Computing" has been developed by me, and all other content has been appropriately referenced.

Andrés Antón Rey Villaverde

This work has been supported by the Spanish Ministry of Innovation, Science and Universities under the grants TIN 2015-65277-R, RTI2018-093684-B-I00 and BES-2016-076806, and the Government of Madrid under the grant S2018/TCS-4423. The associated research internships have been supported by the Erasmus+ International Programme and the HiPEAC Network.

Por la presente, declaro que todo el contenido presentado en esta tesis titulada "Relajaciones de Ejecución Definidas por el Usuario para la Mejora de la Programabilidad en Computación Paralela de Altas Prestaciones" ha sido desarrollado por mí, y cualquier otro contenido ha sido apropiadamente referenciado.

Andrés Antón Rey Villaverde

Este trabajo ha sido financiado por el Ministerio de Innovación, Ciencia y Universidades bajo los proyectos TIN 2015-65277-R, RTI2018-093684-B-I00 y BES-2016-076806, y por el Gobierno de la Comunidad de Madrid bajo el proyecto S2018/TCS-4423. Las estancias de investigación asociadas a este trabajo han sido financiadas por el programa de intercambio Erasmus+ Internacional y por la Red HiPEAC.

To my family.

A mi familia.

Acknowledgements

I would like to thank my advisors Fran and Manuel for helping me and counting on me during these years, showing me their trust, support and giving me the proper advices at the proper moments. I also greatly thank their support during my research internships abroad, in which I learned so much, and also their trust to let me explore some rather tangential topics, in relation to the core thesis discipline, which however showed to be crucial during the exploratory scientific process. Thanks to Fran, specially for the valuable feedback received throughout this thesis, and for having helped me so much in the achievement of the thesis objectives, encouraging me to *get things done* in the most important moments. Thanks to Manuel, also for the accurate advices, specially for opening the ArTeCS doors for me, both the first and the second time, for showing me his trust throughout all of these years, not only during the Ph.D. years, and also for counting on me when Ph.D. sponsoring opportunities appeared.

Thanks to my family, for the constant support received not only during the development of this thesis, but also for supporting me in those personal decisions in which I prioritized *learning*, the *personal introspection*, and the *scientific training* over other options presumably more *ordinary*, *expected* and less *precarious* (and probably more boring). I want to thank my mother and father, from whom I learned personality traits very important in life and crucial to finish this Ph.D. degree, such as the culture of effort, the importance of education, persistence, and honesty; and also for having prioritized their sons and daughter over anything, always respecting our independence. I thank my brother for his influence in my education, and for passing his ambition on me, so important to visualize the *big picture* and to keep the motivation alive. I also thank my sister for the support received from the very beginning of these Ph.D. studies and for always passing her positivity on me.

Thanks to Amparo, specially for supporting me and bear me during these last stressful months, and for sharing with me such amazing vacations. Thanks to my Cisneros friends Javier, Juan Miguel, Antonio, Alejandro, Xabier, Pablo and Eliseo for keeping loyal to our (increasingly rare) meetings and (increasingly frequent) weddings. Thanks also to my childhood friends Victoria, Manuel, Valentín, Jaime, Iago and Domingo, for being *more closer than farther* after so many years. Thanks to Agathe, for having unconditionally stayed with me in the beginning of this thesis, both in the good and worse moments, and to Dominic, specially for those conversations (initiated at the end of the world five years ago and still maintained), for passing his idealism and motivation on me, and for his interest in the developments of this thesis. Thanks to my *industrial* friends Ángel Rosso, Elena Saiz, Elena Garzón, Mar Robledo, Alberto Palomar and Laura Vallejo and to my *Impanati / Jamadan* friends, Marco, Davide and Javier, for those meetings, beers and rehearsals that helped me so much to get away from the thesis when I needed it the most. Thanks also to the *Impanati* guys to bear with patience the rehearsal interruptions during my internships abroad.

Thanks to the people in the Computer Architecture division, starting from Iñaki and Manuel, who opened the ArTeCS doors for me, and specially to all those people I had the pleasure to meet throughout these years, such as Daniel Tabas, Jorge Quintás, Roberto Cano, Luis Costero, Nacho Gómez, Javier Setoain, Edgardo Mejía, Juan Carlos Sáez, Luis Piñuel, Christian Tenllado, Fernando Castro, Guillermo Botella, Katzalin Olcoz, Rafael Sánchez, Joaquín Recas and María Guijarro.

Thanks to Jan Prins for accepting to be my advisor in Chapel Hill, for inviting me to his home,

and for our discussions and his accurate insights that identified the limitations of my ideas, also helping me to address them. Thanks to the people I met in North Carolina, specially Joshua, Christian and Camila, for the interesting conversations and for making my internship so much fun. Thanks to the Codeplay people, Ruyman, Marya, Peter, Marios, Alex, Gordon, Uwe and Christopher for giving me the opportunity to work and learn from them, giving me such an excellent research experience in Edimburgh, which helped me so much to focus my further developments.

Recalling my beginnings in computation and simulation worlds, I want to thank the professors that helped me to get experience in numerical methods applied to computational physics. I want to thank Carlos Spa, for giving me the opportunity to work and learn from him in Chile, and for encouraging me to start the Ph.D. studies. I also want to thank Víctor Martín, who initiated me in the fascinating world of statistical mechanics (to which I will return); and Leo González, who initiated me in computational fluid dynamics, appreciating my motivation over my previous experience.

I also want to thank the best university professors I had, who have reinforced my passion for learning, also initiating me in *the ways of the science*, who have greatly inspired me during my studies of engineering and physics. Despite several years have passed, I still have vivid memories of their lectures and the sensations that they awakened on me back then, which years later have somehow guided me toward starting Ph.D. studies. They are Enrique Maciá, Estrella Alonso, Ángela Jiménez, Juan Pedro Villaluenga, Luis Garay, Felipe Llanes, José Ramón Pelaez and again Víctor Martín.

I want to thank the reviewers Aleksandar and Ricardo for their valuable suggestions that, together with my advisors, have greatly contributed toward enhancing the quality of the current dissertation. Moreover, I want to thank the anonymous reviewers of the published articles, as their feedback has also guided the research conducted in this thesis. I also want to thank in a general sense the developer communities that are constantly pushing computer technology to new heights in a passionate and idealistic way, either developing open source tools, contributing to programming language standardizations, and also producing documentation and disseminating it in open and free media. Specifically, the developments in this thesis depart to a great extent from the Standard C++ committee works, and some of the proposed ideas exposed in this thesis would not have been possible without all the new functionalities incorporated in modern C++ standards.

Agradecimientos

Quisiera agradecer a mis directores Fran y Manuel haberme ayudado tanto y contar conmigo durante estos años, dándome la confianza, el apoyo y los consejos en los momentos adecuados. También agradezco enormemente su apoyo durante mis estancias de investigación en el extranjero en las que tanto he aprendido, así como su confianza para permitirme explorar temas a priori *tangenciales* a la disciplina central de la tesis, pero que han sido clave durante el proceso de exploración científica. Gracias a Fran, especialmente por el valioso feedback recibido a lo largo de esta tesis y haber influido tanto en la consecución de los objetivos, ayudándome a empujar hasta *sacar el trabajo adelante* en los momentos en los que más se necesitaba. Gracias a Manuel, también por los consejos acertados y especialmente por abrirme las puertas de ArTeCS, tanto la primera como la segunda vez, por la confianza demostrada durante todos estos años, no solamente durante la dirección de esta tesis, y también a la hora de contar conmigo para la financiación de la misma.

Gracias a mi familia, por el constante apoyo recibido no sólo durante el desarrollo de esta tesis, sino también por apoyarme en aquellas decisiones personales en las que prioricé el *aprendizaje*, la *exploración personal* y el *desarrollo científico*, por encima de otras opciones quizás más *habituales*, *esperables* y menos *precarias* (y probablemente más *aburridas*). Le doy las gracias a mi madre y a mi padre, de quienes he aprendido cualidades clave para la vida y para terminar este doctorado, como la practicar la cultura del esfuerzo, la valoración de la educación, la constancia y la honestidad; y también por haber puesto a sus hijos por encima de todo, siempre respetando y valorando nuestra independencia. Agradezco a mi hermano su influencia en mi formación, y por contagiarme su ambición, tan fundamental para visualizar el *big picture* y mantener la motivación viva. Agradezco también a mi hermana el apoyo recibido desde el *minuto cero* de este doctorado y por haberme transmitido siempre su positivismo.

Gracias a Amparo, especialmente por haberme apoyado y aguantado durante estos últimos meses de estrés, y por compartir tan geniales vacaciones. Gracias a mis amigos del Cisneros, Javier, Juan Miguel, Antonio, Alejandro, Xabier, Pablo y Eliseo por seguir fieles a nuestras (cada vez más ocasionales) quedadas y (cada vez más frecuentes) bodas. Gracias también a mis amigos de Coruña, Victoria, Manuel, Valentín, Jaime, Iago y Domingo, por seguir *más cerca que lejos* después de tantísimos años. Gracias a Agathe, por haberme acompañado incondicionalmente en los inicios de la tesis, tanto en los buenos como en los malos momentos, y a Dominic, especialmente por esas conversaciones (iniciadas en los confines del mundo hace cinco años y todavía mantenidas), por contagiarme su idealismo y motivación, y por su interés constante en el desarrollo de esta tesis. Gracias a mis amigos *industriales*, Ángel Rosso, Elena Saiz, Elena Garzón, Mar Robledo, Alberto Palomar y Laura Vallejo, y a mis amigos *Impanati / Jamadan*, Marco, Davide y Javier, por aquellas quedadas, cervezas y ensayos que me han ayudado tanto a desconectar de la tesis cuando más lo necesitaba. Gracias también a los *Impanati* por aguantar con paciencia los parones ensayísticos durante mis estancias en el extranjero.

Gracias a la gente del departamento de Arquitectura de Computadores y Automática empezando por Iñaki, que junto con Manuel me abrió las puertas de ArTeCS, y especialmente a toda la gente que he tenido el placer de conocer a lo largo de estos años, como Daniel Tabas, Jorge Quintás, Roberto Cano, Luis Costero, Nacho Gómez, Javier Setoain, Edgardo Mejía, Juan Carlos Sáez, Luis Piñuel, Christian Tenllado, Fernando Castro, Guillermo Botella, Katzalin Olcoz, Rafael Sánchez, Joaquín Recas y María Guijarro.

Gracias a Jan Prins, por aceptar ser mi supervisor en Chapel Hill, por abrirme las puertas de su casa, y por nuestras conversaciones y sus acertadas observaciones que identificaron con gran precisión las limitaciones de mis ideas, ayudando también a resolverlas. Gracias a la gente que conocí en Carolina del Norte, especialmente a Joshua, Christian y Camila, por las interesantes conversaciones y por hacerme mucho más divertida la estancia. Gracias a la gente de Codeplay, Ruyman, Marya, Peter, Marios, Alex, Gordon, Uwe y Christopher, por darme la oportunidad de trabajar y aprender tanto con ellos, dándome una experiencia de investigación inmejorable en Edinburgo, y que tanto me ha ayudado en focalizar mis desarrollos.

Recordando mis inicios en el mundo de la computación y la simulación, quiero agradecer a los profesores que me han ayudado a introducirme y desarrollarme en métodos numéricos aplicados a la física computacional. Quiero dar las gracias a Carlos Spa, por darme la oportunidad de trabajar y aprender con él en Chile, y por animarme a empezar el doctorado; así como a Víctor Martín, quien me ayudó a introducirme en el apasionante mundo de la mecánica estadística (al que volveré); y a Leo González, quién me introdujo en la dinámica de fluidos computacional valorando más mi motivación que mi experiencia previa.

También quiero dar las gracias a los mejores profesores y profesoras que he tenido en la universidad, que han reforzado mi pasión por aprender, introduciéndome en *los caminos de la ciencia*, y que más me han inspirado durante mis estudios de industriales y físicas. Aunque hayan pasado unos cuantos años, todavía tengo vívidos recuerdos de sus clases y de las sensaciones que me despertaron entonces, y que de algún modo, años después, me han llevado a iniciar los estudios de doctorado. Ellos son Enrique Maciá, Estrella Alonso, Ángela Jiménez, Juan Pedro Villaluenga, Luis Garay, Felipe Llanes, José Ramón Pelaez y de nuevo Víctor Martín.

Quiero agradecer a los revisores Aleksandar y Ricardo por su valiosos comentarios que, junto con mis directores, han contribuido enormemente a mejorar la calidad de la presente memoria. Así mismo, quiero agradecer a los revisores anónimos de los artículos publicados, ya que también han contribuido a guiar la investigación llevada a cabo en esta tesis. Quiero agradecer también de forma general a las comunidades de desarrolladores que constantemente, de forma apasionada e idealista, empujan la tecnología informática hacia nuevas cotas, ya sea mediante desarrollo de herramientas de código abierto, estandarización de lenguajes de programación, o generación de documentación y divulgación desinteresada en medios abiertos y gratuitos. En concreto, los desarrollos de esta tesis derivan en buena parte del trabajo del comité de estandarización del lenguaje C++, y parte de las propuestas planteadas en esta memoria no hubieran sido posibles sin todas las nuevas funcionalidades que han sido incorporadas en los nuevos estándares C++.

Contents

List of Figures

List of Tables

Listings

Abstract

This thesis proposes the **development and implementation of a new programming model based on execution relaxations, and focused on High-Performance Parallel Computing**. Specifically, the main goals of the thesis are:

1. Advocate a development methodology in which users define the basic computing units (tasks), together with a set of *relaxations* in, possibly, multiple dimensions. These relaxations will be translated, at execution time, into expanded (and complex) scheduling opportunities depending on the underlying architectural features, yielding improvements in terms of desired output metrics (e.g., performance or energy consumption).

2. Abstract away users from the complexity of the underlying heterogeneous hardware, delegating the proper exploitation of *expanded scheduling* choices to a system software component (typically referred as a *runtime*). This piece of software, armed with knowledge from static architectural characteristics and dynamic status of the hardware at execution time, will exploit those combinations considered optimal among those relaxations proposed by the user for each task ready for execution.

3. Extend this abstraction in order to describe both computing systems, by means of executor / allocator hierarchies that describe the heterogeneous computing architecture, and applications, in terms of sets of interdependent tasks. In addition, the relations between executors and tasks are categorized into a new *task-executor taxonomy*, which motivates *ambiguity-free* HPC programming frontends based on the *STSE, Single Task - Single Executor* classification, distinguished from *fully-automated* runtime backends.

4. Propose a new programming model (STEEL) based on previous ideas, that gathers features considered to be basic for future task-based programming models, namely: performance, composability, expressivity and *hard-to-misuse* interfaces.

5. Specify an API to support the STEEL programming model, and a runtime implementation leveraging techniques and programming paradigms supported by modern C++, illustrating its flexibility, ease of use and performance impact by means of simple use cases and examples.

Hence, the proposed methodology stands for a clear and strict separation of concerns between the involved actors in a parallel executions: user / codes and underlying hardware. This kind of abstractions allows a delegation of the expert knowledge from the user toward the system software (runtime) in a systematic way, and facilitates the integration of mechanisms to automate optimizations, adapting performance to the specificities of the heterogeneous parallel architecture in which the code is instantiated and executed.

From this perspective, the thesis designs, implements and validates mechanisms to perform a so-called *complexity formalization*, classifying many actions that are currently done by the user and building a framework in which these complexities can be delegated to the runtime system. The delegation of these decisions is already a step forward to next generation of programming models seeking performance, expressivity, programmability and portability.

Resumen

La presente tesis doctoral propone el **diseño e implementación de un nuevo modelo de programación basado en relajaciones de ejecución y enfocado al ámbito de la Computación Paralela de Altas Prestaciones**. Concretamente, los objetivos principales de la tesis son:

1. Abogar por una metodología de desarrollo en la que el usuario define las unidades básicas de cómputo (tareas), junto con un conjunto de *relajaciones* en, posiblemente, múltiples dimensiones. Estas relajaciones se traducirán, en tiempo de ejecución, en oportunidades expandidas (y complejas) de planificación en función de la arquitectura subyacente, impactando así en métricas como rendimiento o consumo energético.

2. Abstraer al usuario de la complejidad del *hardware* subyacente, delegando la correcta explotación de dichas posibilidades de planificación expandidas a un componente *software* de sistema (típicamente conocido como *runtime*). Dicho *software*, dotado de conocimiento tanto de las características estáticas de la arquitectura subyacente como del estado puntual de la misma en el momento de la ejecución, explotará las combinaciones consideradas óptimas de entre las relajaciones propuestas por el usuario para cada tarea lista para set ejecutada.

3. Extender dicha abstracción para describir tanto sistemas de cómputo, en forma de jerarquías de ejecutores y alojadores de memoria que en último término describen una arquitectura de cómputo heterogénea, como aplicaciones, en forma de un conjunto de tareas interrelacionadas. Además, las relaciones entre ejecutores y tareas son clasificadas en una nueva *taxonomía tarea-ejecutor*, la cual motiva frontends de programación HPC *sin ambiguedad* basados en la clasificación *STSE, Single Task - Single Executor*, separada de backends runtime totalmente automatizados.

4. Proponer un nuevo modelo de programación (STEEL) basado en la clasificación STSE que aglutine ciertas características consideradas básicas de cara al éxito de los futuros modelos de programación basados en tareas: rendimiento, facilidad de composición, expresividad e interfaces no permisivos ante fallos.

5. Especificar una API que dé soporte al modelo de programación, así como una implementación *runtime* del mismo aprovechando técnicas y paradigmas soportados en el lenguaje C++ de última generación, e ilustrar su uso, flexibilidad e impacto en el rendimiento a través de ejemplos y casos de uso sencillos.

La metodología que se propugna aboga por una clara y estricta separación de conceptos entre los actores básicos que componen una ejecución paralela: usuario / código y *hardware* subyacente. Este tipo de abstracciones permite delegar el conocimiento experto desde el usuario hacia el *software* de sistema, proporcionando así mecanismos para mecanizar y automatizar su optimización, y adaptar su rendimiento a la arquitectura paralela sobre la que se instanciarán los códigos.

Desde este punto de vista, la tesis diseña, implementa y valida mecanismos para llevar a cabo una *formalización de la complejidad* inherente a la programación paralela heterogénea, clasificando aquellas acciones que en la actualidad se llevan a cabo por parte del usuario en el proceso de desarrollo y optimización de código, y proporcionando un marco de trabajo en el que dicha complejidad puede ser delegada, de forma eficiente y consistente, a un *runtime*.

Against the sneakiest undead lock
and hideous invisible bugs
who hide and never talk.

Through the fire in the chip
and bursts in the heap
that oblige to swap.

Let's stand composed
with strongly typed shields.
Let's relax the parallel demon
coding spells of STEEL.

1 Introduction

1.1 Fundamentals

In a world dominated by Information Technology and Computer Science, computers have been transforming the society since their first physical implementations at the beginning of the previous century, and it is likely that this transformation is not going to end anytime soon. Since the advent of programmable computers, High-performance Parallel Computing (HPC) in particular, and computer simulation in general, have played a crucial role supporting new developments in science and engineering, driving technology and industry toward efficiency, and pushing science to greater levels of knowledge in virtually all disciplines.

However, as it occurs in any technological human discipline, there are still a plethora of open problems in HPC that need to be solved in order to keep extending its applicability to increasingly important scientific and engineering challenges.

1.1.1 Challenges in High-performance Parallel Computing

Since the advent of silicon-based transistors, hardware advances based on physical features –miniaturization, increasing transistor count and higher clock frequencies– and technical advances –branch prediction, deep pipelines, speculative and out-of-order execution, deep cache hierarchies, vector units– [69] managed to keep an exponential growth in microprocessor performance for several decades, while respecting a purely sequential programming paradigm. This trend started to show signs of stagnation when heat dissipation and energy efficiency in microprocessors became an ever-growing problem. This *power wall* [11] problem was circumvented by architects by means of parallelism, rather than clock frequency, which led to the popularization of *parallel* on-chip microprocessors.

However, this *multicore revolution* brought its own set of problems to solve. In contrast to the previous *gold sequential era* in which a two-fold increase of processing performance was just a matter of waiting 18 months [132], parallel processing came with its own limitations with regard to *applicability* and *development productivity*. Specifically, efficient parallel computation trivially requires *(i) application parallelizability* –i.e., the application must inherently expose a certain amount *divisible* work that can be computed independently (quantifiable with the Amdahl's Law [71])–, and *(ii)* parallel programming knowledge –i.e., the programmer must be able to partition and synchronize the workload– needed for the execution performance to scale with the number of processing cores.

There is little that can be done regarding the parallelizability of an application, as it is an

intrinsic property that cannot be circumvented. With respect to the latter requirement it is not a surprise that parallel programming is hard, and generally involves a thorough design that requires an advanced knowledge of *(i)* parallel actors (threads and processes) and *(ii)* synchronization tools (locks, mutexes, condition variables, barriers, latches, futures), to avoid generally hard to debug errors (deadlocks and livelocks) and race conditions.

Moreover, parallel architectures entail an additional (and typically harder to achieve) *parallel performance* requirement, which is not whatsoever guaranteed even though program correctness is attained. Parallel performance tuning typically requires an advanced knowledge about both the underlying parallel architecture and the abstraction layers in between the source code and the actual machine code that is being executed in the processor.

In addition, not only the sheer size of the underlying parallel platform makes the performance problem even harder to attain, but also its complexity in terms of deep, wide and distributed memory hierarchies, and parallel/heterogeneous processors interconnected in complex topologies. In particular, the *memory wall* problem [158] –illustrated by the fact that in many applications the data movement is a few orders of magnitude more expensive (in terms of time and energy) than computation– is aggravated in distributed systems, making it one of the major issues that limit the scalability and performance of modern large-scale applications running on high-end supercomputers.

Soon after the multicore chips became pervasive in practically every device, the increasing hardware advances in GPU technology driven by entertainment industry (gaming and 3D animation), led to the generalization of these previously domain-specific hardware architectures into a more general-purpose-oriented graphics processors. Nowadays, partly thanks to the development of programming models, interfaces and open standards designed to target them, and also thanks to its paramount performance-per-watt efficiency, general-purpose GPUs are already a common component in any parallel system used for HPC purposes. In a broader sense, and in analogy with the *multicore revolution*, the *heterogeneous revolution* was coined as other domain-specific hardware, apart from GPUs, like FPGAs, DSPs and TPUs, gained popularity in HPC applications and systems.

Motivated by the demands of large applications dealing with climate modeling, materials science, energy research, astrophysical research, data analytics, or medical research applied to drug discovery and protein folding, future exa-scale supercomputers aim at delivering a sustained performance peak of 10^{18} floating point operations per second with power demands up to 20 MW [48]. In order to attain such numbers, the so-called exa-scale challenges require hardware and software innovations to enhance *performance/scalability*, *energy efficiency*, *programmability* and *resiliency*. With regard to execution performance, scalability and programmability, exa-scale computers will demand unprecedented levels of *scheduling complexity*: hierarchical and massive levels of parallelism will need to be automatically managed, and data movement will require a coherent orchestration across multiple and deep cache hierarchies. Moreover, the failure rate probability due to soft and hard errors is expected to rise to levels that may cause exa-scale executions not only be expensive, but also impractical. For this reason, *execution resiliency* is being considered as a topic with major research interest [25, 134].

1.1.2 Addressing complexity through abstractions and hierarchies

Abstraction is used in this thesis as a mechanism to hide the complexity by means of *modeling* and *simplification*, aiming at generality at a cost of neglecting the details, when needed. Computer Science is a discipline deeply rooted in abstraction, and its success in society has partly derived from the engineering effort of stacking multiple levels of abstraction layers in deep hierarchies, so that highly abstract concepts in the human mind can be consistently instantiated as purely physical processes that represent those ideas.

Regarding the challenges in parallel programming, it is worth noting that several levels of parallelism are commonly handled in user-oriented applications running on devices such as smart-

phones, tables and laptops. For instance, microprocessor pipelining, vector units, and compiler technology can automatically enable instruction-level and data-level parallelism from a purely sequential program –i.e., the computer user does not need to learn parallel programming to benefit from it through faster and more responsive applications–. Also, modern processors provide support for automatic and efficient memory management mechanisms in a multicore context with deep memory hierarchies, not demanding the user to learn cache-coherent protocols. Similarly, in a higher level, Operating Systems hide system-wide management work to the user while exposing a simplified layer to developers to access some functionalities such as I/O, networking, file operations or memory management. All of these examples show how computers have evolved from expert-oriented machines to pervasive and essential working tools for non computer experts.

From the point of view of application and software developer experts, a plethora of standardized and general-purpose programming languages have been designed according to an abstract computation machine. Downstream, during compilation, a program is translated into more specific microarchitecture-dependent machine instructions. Syntactic rules of a programming language serve as a *contract* to be satisfied by both the programmer and the language implementation in the form of a compiler and possibly an underlying runtime, imposing restrictions to the user so that only syntactically-legal expressions are translated into an actual program. High-level languages are characterized by providing very clear, simple and readable syntactic rules while hiding details of the computer architecture. However, high-abstractions usually come at the price of lack of control and reduced performance.

Finally, from the point of view of parallel application developers, recalling the parallel programming challenges and the role of abstractions, this thesis goes in the same direction to which computer technology is going: *toward greater levels of automatization and abstraction, to drive HPC in particular, and parallel programming in general, toward being a technology usable to non-HPC experts*.

1.2 Programmability in HPC

In essence, Computer Engineering aims at automating some form of *information processing* –or in other words, it addresses the problem of how efficiently automatize *computation* under limited resources–. As mentioned, the percolation of computers into all areas of modern society has not only been based on the exponential growth in computation performance experienced during four decades (i.e., a *hardware* achievement), but also on the simplification of computer interfaces (i.e., a *software* achievement).

In particular, the impact of High-performance Computing has become paramount in many areas of science and engineering. Currently, it is a very specialized field with direct application in a plethora of disciplines. However, HPC users should not be experts in HPC, nor in the specificity of hardware or the underlying software infrastructure; as a technology meant to be used in such a diverse set of disciplines should abstract away the complexities of both aspects. Ideally, focusing on the programmability/portability perspective, a user of a parallel/heterogeneous resource should be exclusively in charge of specifying the input configuration and/or data of the problem and let the system seamlessly map the computation to the underlying parallel platform.

This is far from being the case in general HPC applications, in which computations are typically coupled in complex ways forming irregular parallel patterns, generating load-imbalanced executions, lack of efficiency, poor scaling and portability problems. Modern parallel programming models, languages, patterns and paradigms have been proposed during the last decades to tackle these issues, suggesting runtime-based solutions –in which some of the programming burden is shifted from the user to an automatic runtime system– as potentially ideal to boost performance while reducing development effort.

The following sections expose a non-comprehensive exposition of the characteristics of some programming paradigms, models, systems and runtimes. Some key characteristics, challenges,

appealing trends and limitations are presented to contextualize the developments presented in the following chapters.

1.2.1 Paradigms

In this section, the *imperative* programming paradigm (in which *object-oriented* and *procedural* sub-paradigms are commonly included) is put in contrast with the *functional* paradigm (which is commonly included in the class of *declarative* paradigms), and their appeal and limitations are addressed under a HPC perspective.

1.2.1.1 Functional paradigm

Lambda calculus (or λ-calculus) was developed as an alternative model of computation to the Turing machine model, and it focuses on *function abstraction* to model general computations, in contrast to the *state manipulation* of Turing machines [35, 36]. In relation to programming, λ-calculus is the theoretical cornerstone of the functional programming paradigm, implemented in many industry-standard popular languages such as C++ [138], Go [49], Scala [117], Clojure [70], Rust [106], Haskell [80], ML [110] and Erlang [28]. Also, other mathematical frameworks such as Type Theory [144] and Category Theory [96] have greatly influenced the design and implementation of modern functional programming languages toward a more clear, composable, verifiable and robust development style [109].

From a *closer-to-programming* perspective, there is a set of properties that characterize modern functional programming philosophy (see [42, 151]), which are briefly developed next.

***Pure* function abstraction and referential transparency.** Functions without side-effects are referred as *pure*, so their unique effect is its return value. By favoring a programming style based on function purity, dependency tracking and state encapsulation becomes natural, while code readability is also improved. These characteristics are of great appeal in parallel programming.

Function purity will be repeatedly addressed in the developments of this thesis, also in a *less*-strict form, when considering third-party software interoperability. Specifically, a function may inter-operate with a third-party program or library whose functionality –and state– may be altered. Since its state may be hidden or *not observable* from the caller, it will be assumed that the third-party program is *well-behaved enough* to keep considering *purity* de facto preserved (i.e., the called third-party function is properly and safely encapsulating its state so that it cannot affect whatsoever the state of the caller).

In addition, following λ-calulus ideas, functions are considered *first-class citizens*, in the sense that functions are mere values of a certain type that can be passed to other functions.

Function purity yields *referential transparency* property, which states that a function outputs the same result regardless of the context. This permits the use of *result memoization* in runtime implementations of functional languages, by which the result of a function call is cached to prevent function recomputation.

Strong typing. Strongly-typed languages enforce strict rules in terms of what can be translated into actual machine code. Motivated behind theoretical frameworks of Type Theory and Category Theory, any data object belong to a specific abstract type that encompasses a finite or infinite set of values. Types are also possibly endowed with properties and constraints from which more complex *compositional* rules (to form composite types) and *transformation* rules (that limits the set of all functions able to map values from one type to the other) are derived. Strongly-typed languages favors robust interfaces, facilitates program verification, and enables the application of logic rules to mathematically prove program correctness.

Immutable data objects. Functional programming language implementations encourages –and even enforces– the use of immutable data objects / structures. Immutable data forbids the existence of shared and mutable states across parallel actors (e.g., threads or processes), thus thread-safety is naturally preserved and the chances for unwanted data races are greatly reduced.

The inexistence of mutable data does not necessarily imply that expensive copies of data objects are needed. With this regard, implementations of purely functional languages may employ internal mechanisms to store only the data differences (or *state changes*, or *deltas*) to allow memory-friendly run-time data transformations, ultimately yielding programs with less memory footprint.

Implicit control flow and lazy evaluation. Declarative-like paradigms (in which functional paradigm is commonly included), encourages a style of programming in which the properties of the different agents forming the program (e.g., data objects or functions) are declared. Loosely speaking, it favors a model of execution in which the computational needs are *pulled on demand* as the execution proceeds. In practice, it usually implies a clear separation between what the programmer expresses in the code and what is actually being computed. *Lazy evaluation* refers to function evaluation only if its result is needed, and it also permits to express infinite structures in computer programs regardless of the *finitude* of computers.

In relation to concurrency and parallelism, the previous ideas are reflected into an undefined execution order in general, and a *relaxed* (i.e., opposite to *user-specified*) parallel execution.

Iteration via recursion. Contrary to imperative-like languages that use loops to implement iteration –which requires some form of state in the form of a counter–, functional programming implements iteration by means of *recursion*. In practice, compiler and runtime implementations of functional languages implement *tail recursion* techniques to avoid an overflow of the stack in situations with deep recursions.

Monad pattern. Monads in Category Theory were applied to programming languages with the goal of formalizing general side-effectful computations within a functional programming context [111]. Applied to programming, monadic patterns are in essence a *generic glue* that enables function composition in a scalable and generic way, providing a framework to develop software handling side-effects without sacrificing functional purity. Several functional programming languages provide the most common built-in monads and also let the user to implement more complex or application-specific monadic patterns.

1.2.1.2 Imperative paradigm

Imperative languages were first implemented matching actual hardware architecture implementations. In essence, the programmer writes commands that result into a program state change, which directly influences state change of the physical machine –i.e., the program commands closely resemble what the hardware actually executes–. Contrary to the functional paradigm, modularization, composability and separation of concerns are important *but not primary*, hence favoring *full control* of what the underlying hardware actually does. Some of the features of the imperative-like paradigm (specially in relation to low-level and non-interpreted imperative languages) are summarized next.

Non-strict encapsulation for full control of program state. Motivated by the need of modularization and composability requirements, mandatory in any mid or large-scale software projects, a set of *subparadigms* like *structured*, *procedural*, *modular* and *object-oriented* were distinguished under the umbrella of the *imperative* paradigm, and they differentiate with respect to the different

ways in which expressed computation is encapsulated (i.e., in terms of subroutines, modules, objects, etc.). In particular, object-oriented programming aroused as an approach for enhanced code reusability and modularization over structured and procedural paradigms, by means of encapsulation, inheritance and polymorphism.

In essence, imperative languages provide language semantics encouraging some form of *mild* encapsulation without sacrificing control, in the sense that high levels of modularization and separation of concerns can be achieved, yet full control over the program state is still available to the programmer.

Execution is mainly user-driven, control flow is explicit. Contrary to the declarative-style of the functional paradigm, the imperative paradigm conforms a programming style based on user-specified *commands* that explicitly define how the underlying execution is performed at run-time. Despite its proximity to the actual hardware architecture, there are still several abstraction layers that separate the computer code and the actual runtime execution. One of these layers is the compilation process, and some knowledge regarding compilation is usually mandatory to achieve high-performance executions.

With this regard, imperative languages are usually easier to learn and less abstract than functional languages, and compiler technology has been greatly improved to close the gap between simple and high-level imperative codes and high-performance runtime executions. Compiler-targeted commands aiming at performance such as inlining, loop unrolling, pre-compiled tables, and cache-friendly data layouts / memory allocations, are frequently employed by implementers in compiled / non-interpreted languages like C or Fortran.

Fine-grained memory management. Memory management – in the form of explicit memory allocations/deallocations and alignment, data layout design, and cache-aware data structures, is usually required when programming high-performance applications in imperative and low-level programming languages. Languages of this kind may also provide a memory model that abstracts the memory model implemented in hardware, so it is usually desirable for the programmer to know the specifications of this model, specially in concurrent and parallel applications meant to run on cache-coherent multiprocessing architectures.

1.2.1.3 Generic paradigm

Generic programming patterns are applicable to either functional [110] and imperative paradigms. This programming paradigm favors a style in which types are first abstracted (or relaxed) and only instantiated (or specified) when needed. The goal is to provide generic algorithms which are parametrized by a set of allowed types, so that maintainability and programming productivity are enhanced. In languages supporting an imperative style like C or C++, genericity is typically achieved by means of macros and templates, respectively. In functional languages like Haskell, full genericity is a built-in feature (its type system *Hask* could be even viewed from a strict Category-theoretic approach [109]). With this regard, all types (including functions) are essentially generic by default and are implicitly parametrized, which can also be constrained and composed to form hierarchical and richer type structures without sacrificing full genericity.

1.2.1.4 Appealing characteristics and limitations from a HPC context

In this section, functional and imperative paradigms are compared in the context of HPC. In particular, the great abstraction power of the functional paradigm (desirable for application scalability and correctness) is compared with the great platform *control* and *observability*, easier to be achieved from a imperative paradigm (desirable for execution performance).

Few observations are made next, which reflect that functional languages offer great abstraction power that encourages safer and verifiable programs, but at a price of sacrificing two aspects crucial in HPC environments: *(1)* lack of execution control and *(2)* lack of platform monitoring.

Program correctness and safety. The functional paradigm offers an advantage with regard to program verification and correctness. Specifically, program development is improved if errors are caught as soon as possible (e.g., better at compile-time rather than at run-time). From the compilation and runtime perspectives, the strong-typing characteristics of many functional-based languages naturally enable the possibility of static analysis and mathematical correctness (programs can be thought as theorems).

Application development. From the developer perspective, functional programming languages tend to favor syntactic terseness and facilitates program analysis through simpler *equational reasoning* –i.e., in functional languages, *equal* means *equal*, not *assignment*, so expressions can be replaced by mere mathematical substitution–. In contrast, readability in imperative-like languages may get obscured due to side-effectful assignments and *operational reasoning*, which demands the developer to keep track of the eventual side-effects of every expression.

Regarding challenges in concurrency and parallelism, explicit synchronization mechanisms to manage shared states in imperative languages typically harm developer productivity and program correctness. Normally, keeping data invariants across threads in order to maintain thread-safety while keeping the granularity of the shared data structures not too coarse (to get an acceptable parallel performance), is a challenging task.

The inherent *side-effect-free* characteristic of purely functional languages opens the door toward models in which the parallelism is automatically derived, not requiring developers to actually specify it and manage thread safety.

The functional paradigm also favors a more scalable development: as composability, reusability and separation of concerns are built-in features, by means function composition generalization through monadic patterns. However, despite the richness and elegance of monadic abstractions, they do require a certain amount of learning effort, and a considerable level of expertise is needed even for mildly complex programs, in relation to the lesser development effort of equivalent imperative programs.

Previous observations are neatly summarized in the following paragraphs extracted from [109]:

> *"One of the forces that are driving the big change is the multicore revolution. The prevailing programming paradigm, object oriented programming, doesn't buy you anything in the realm of concurrency and parallelism, and instead encourages dangerous and buggy design. Data hiding, the basic premise of object orientation, when combined with sharing and mutation, becomes a recipe for data races. The idea of combining a mutex with the data it protects is nice but, unfortunately, locks don't compose, and lock hiding makes deadlocks more likely and harder to debug."*
>
> . . .
>
> *"But even in the absence of concurrency, the growing complexity of software systems is testing the limits of scalability of the imperative paradigm. To put it simply, side effects are getting out of hand. Granted, functions that have side effects are often convenient and easy to write. Their effects can in principle be encoded in their names and in the comments. A function called SetPassword or WriteFile is obviously mutating some state and generating side effects, and we are used to dealing with that. It's only when we start composing functions that have side effects on top of other functions that have side effects, and so on, that things start getting hairy. It's not that side effects are inherently bad — it's the fact that they are hidden from view that makes*

them impossible to manage at larger scales. Side effects don't scale, and imperative programming is all about side effects."

Execution control. Despite composability and modularity characteristics of functional-like patterns may improve *software scalability*, it is not usually followed by *performance scalability*. Runtime implementations of functional languages may hide from the user some performance-critical data management operations (e.g., non-cache-friendly data manipulation, garbage collection) that may greatly harm execution performance. In particular, note that data-persistence / immutability restriction imposed in some functional languages may require alternate and *less natural* data layouts and data structures representations, different from their *imperative language* representations, which could incur into prohibitive performance penalties. On the contrary, low-level imperative languages typically give access to the developer for finer-grained control of the hardware resources, exposing user-defined data movement operations and explicit use of parallel synchronization primitives for high-performance tuning. These observations reflect the fact that programs written in purely functional languages are not competitive in resource-constrained scenarios demanding high-end performance (e.g., real-time systems and high-performance computers).

Platform observability for state-driven execution. Note that all programs ultimately run on stateful physical machines (whose state does not exist in the abstract functional program). Hence, the concept of *purity* is valid or *applicable* only within the program semantics. This observation poses serious limitations into the applicability of *functional purity*-based ideas in High-performance Parallel Computing, in which the execution performance of a massively parallel program is highly context-dependent. Specifically, any HPC-oriented runtime system must operate with a consistent monitoring of the underlying hardware system state to mitigate usual load-balancing, contention, starvation and data locality problems arising at run-time. In this sense, modeling context-dependent computations from a functional approach is currently a research topic [121, 118]. On the contrary, the *close-to-the-hardware* nature of imperative-like languages and their easier learning curve have greatly favor the development of context-aware programming models, runtime systems and high-performance parallel applications.

1.2.2 Models, systems and runtimes

Parallel programming models are commonly conceptualized as *bridges* between applications – that is, what the user intends to compute–, and hardware –the computing platform– (see [11]). Essentially, they provide abstractions of a parallel computing platform, and they can be evaluated based on *(i)* its capacity to achieve high-end performance (out-of-the-box, or after a performance tuning stage); *(ii)* its generality of use for a range of applications and platforms; and *(iii)* its abstraction power, or its ability to hide the details of the underlying platform [15]. In this thesis, the concept *programming model* is used to indistinguishably refer to either *programming styles* (usually referred to the different ways in which concurrency and parallelism can be expressed) and *programming systems* (which refer to the actual programming interfaces by which applications are expressed). With this regard, the *programming model* concept usually encapsulates a programming style *and* a programming interface, as it is the case of the programming model presented in this thesis.

According to the *Berkeley Bridge* idea [11], writing high-performance parallel applications should be no more difficult than writing sequential applications. The work of this thesis departs from the same hypothesis, arguing that the success of a programming model requires to consider programming productivity, portability, maintainability and scalability on equal footing with execution performance and efficiency.

With regard to existing parallel programming models, OpenMP [30] is probably one the most popular models targeting shared memory parallel processors. Designed to target C and Fortran programs, OpenMP is an open standard whose implementations usually achieve a considerable

out-of-the-box performance by simple loop parallelization through compiler *pragmas*, following a fork-join and vector Single Instruction - Multiple Data (SIMD) model. Thanks to its popularity, the simplicity of compiler directives and its design based on open standards, OpenMP was considered as a reference for other models and frameworks targeting task parallelism and accelerators such as OpenACC [156], OpenHMPP [10], HOMPI [47], XKAAPI [58] and OmpSs [51]. Interestingly, the latter has influenced the incorporation of task-parallelism style into the OpenMP standard.

Other programming models targeting multicores were developed as the multicore architectures and distributed platforms gained attention, for example Cilk [131], Loci [104], Chapel [29], Charm++ [86], UPC [26], Co-array Fortran [112], OpenSHMEM [31] and X10 [32]. Those targeting distributed architectures typically employ a PGAS abstraction layer and internal MPI-based [62] mechanisms for inter-node communication. In particular, MPI is still the most popular model for distributed computing and Single Program - Multiple Data (SPMD) parallelism, and it is commonly used orthogonally with other models in large-scale applications, forming other (MPI + X) *hybrid* models where X is a model targeting intra-node computation (commonly OpenMP).

The so-called *heterogeneity revolution* was pioneered by proprietary NVIDIA CUDA programming model [115]. The simplicity and great applicability of data-parallel patterns in many scientific and data-centric applications, together with the unprecedented parallelism and performance delivered by Single Instruction - Multiple Thread (SIMT)-based GPU architectures, seduced the HPC community soon after the first general-purpose GPUs were put on sale.

NVIDIA CUDA packs a vendor-specific API, a compiler and a runtime system highly tuned to squeeze the maximum performance of their GPUs. Also, NVIDIA GPUs and CUDA (API and runtime) are regularly improved and updated, and new hardware features of new GPUs can be readily exploitable via new software updates timely delivered. In addition, CUDA is originally envisioned to support heterogeneity: its compiler is able to output x86 code, and it will provide full support for either x86, ARM and POWER architectures [41].

With equivalent ambitions with regard to heterogeneity, but aiming at open standards, Khronos Group industry consortuim released OpenCL [137]. Over the last years, OpenCL has been implemented for a wide variety of architectures, also demonstrating its appeal to a great number of hardware vendors willing to leverage a stable and open-standardized language to develop applications running on their devices. OpenCL is designed to operate in C programs and its implementations are equipped with a Just-in-time (JIT) compiler for computational kernels and a runtime system.

CUDA and OpenCL are currently the main programming models used in GPU kernel development. However, the programming and portability challenges derived from the need of efficient orchestration of a heterogeneous computation –in terms of data movement across memory spaces, heterogeneous kernel dispatching, and the need for data-computation overlap–, motivated the development of other (mainly task-based) programming models and runtime systems on top of CUDA and / or OpenCL, which could perform this kind of management transparently. For instance, the already mentioned directive-based approaches [10, 47, 51, 156, 20, 58] also offered support for GPU computing, while others like SGPU2 [119], StarPU [12], Qilin [103] and Harmony [46] enabled heterogeneous workload dispatching with (non-directive-based) custom task definitions.

Some of the models just mentioned adopted a task-based style, arisen in response to the limitations of imperative-like programming models based on coarse-grain, and bulk-synchronization (employing fork-join and SPMD programming styles). The benefits of asynchrony and fine-grain synchronization over previous styles favored computation expression in terms of a *tasking / data-flow computation / data-oriented workflow* style, in which the execution is *driven* from data dependencies that bind tasks, defined in a *declarative* way (unsurprisingly, the *data-flow* style is natural in the functional programming paradigm [4]).

With this regard, HPX model [85] addresses the following issues that dampens performance scalability in modern HPC systems, namely (1) *processor starvation*, (2) *latencies* due to access to remote resources, (3) *runtime overhead* due to parallel synchronization, and (4) contention

resolution. These problems are partially inherent to other classic parallel bulk-synchronous patterns, based in global synchronization, message passing and fixed data distribution. HPX runtime implementation revolves around several ideas taken from asynchronous, adaptive, and data-flow patterns, pursuing executions driven by data locality, fine-grain synchronization of tasks derived from dependency resolution, and hierarchical parallelism.

In particular, in fork-join and bulk-synchronous models, hierarchical or *nested* parallelism do not easily scale due to the explicit use of synchronization barriers programmed statically –e.g., all actors need to wait for the slowest actor at every barrier–. In response to the limitations of these models, the expression of the computation in the form of asynchronous tasks generated as data dependencies are resolved, provides an on-demand and fine-grain synchronization of parallel actors, which in turn yields a much more flexible orchestration of the workloads able to adapt to dynamic changes in the parallel platform state.

Note that the task-asynchronous execution model also inherits some of the ideas from the functional paradigm, in the sense that the *when*, *where* and *how* the computation is performed, are *loosen* or *relaxed* from the perspective of the *caller* (i.e., the dispatcher of the asynchronous computation). Specifically, the *when* is relaxed by means of *asynchrony* itself, while the *where* and the *how* are decided by the recipient of the asynchronous call, possibly accounting for some form of runtime state observation and workload characteristics. In other words, execution control flow is implicit and uncertain for the caller, and resolved only when needed.

Moreover, this execution model naturally favors data-computation overlap and task parallelism, that ultimately helps to reduce processor starvation and hide data transfer latencies. Also, from a programmability perspective, task-asynchrony driven by data dependency detection favor a sequential-style and less error-prone programming that enhances development productivity and code readability (i.e., *taskified* code looks sequential and data dependencies can be automatically detected at run-time).

1.2.2.1 ISO C++ 11/14/17

C++ is a language developed and standardized in the eighties that first endowed C language with object-oriented and generic programming features. Since its inception, it has achieved great success in science and industry, both as a system-software and applications language, and also as basis of many of the programming languages, libraries and runtime systems aforementioned in this chapter. For instance, in relation to current challenges in parallel programming models, SYCL [90, 88], Kokkos [52], PACXX [65] and Raja [74, 91] have adopted the benefits of C++ departed from high-level C++ abstraction layers to aim at more performance-portable computational kernels.

Latest 2011, 2014 and 2017 ISO[1] C++ standards (also referred as *modern* C++) have dramatically improved the language by enhancing its expressivity and its capabilities in terms of generic and functional programming patterns. Moreover, since C++11 standard, the language incorporated a memory model that provides native support for concurrency. Not only new syntax has been added to it, but development guidelines of modern C++ are progressively discouraging (although still supporting) classic object-oriented patterns in favor of compile-time polymorphism jointly with generic and functional styles. Also, latest modern C++ 17 template meta-programming [2] –through expanded *constexpr* syntax and fold expressions– have also radically reduced the compile-time programming efforts and let the developers to yield very robust type-safe runtime systems usable by means of high-level, expressive, and hard-to-misuse interfaces.

In particular, some of the guidelines and principles encouraged by modern C++ programming are:

- *Favor zero-cost, easy-to-read and high-level abstractions.*

[1] International Organization for Standardization.

- *Don't pay for what you don't use.*
- *Do computation at compile-time better that at run-time as much as possible.*
- *Check for errors at compile-time better that at run-time as much as possible.*
- *Minimize the possibility of resource leaks via ownership semantics.*

Specifically, *high-level* yet *zero-cost* abstractions are provided through compile-time computations and highly expressive *meta-programming* idioms. By *zero-cost* it is meant that abstractions based on compile-time computations do not necessarily add any cost *at run-time*. In particular, *template meta-programming* is itself a Turing-complete functional language interpreted by a C++-compliant compiler. It is designed to express sophisticated generic programs and to build higher-level domain-specific languages leveraged on transparent compile-time computations. The resulting executables are meant to deliver excellent run-time performance. Apart from its multi-paradigm functionalities, modern C++ can be thought as a *high-performance* and *multi-layered* language: multiple high-level zero-cost abstractions can be stacked and integrated with lower-level C-specific constructs.

1.3 Motivation

Based on the previously exposed programming paradigms, models, engines and frameworks, next sections elaborate the departing hypothesis from which the developments of this thesis are derived, in particular influenced by the need of enhanced *programmability*, higher *abstractions*, *automatization*, *expressiveness* and *portability*.

1.3.1 HPC programmability is still unsolved

Regarding parallel programming, the tradeoff *performance-and-control* vs. *programmability-and-abstraction* is currently very much present in programming languages and models. However, this dychotomy is not a natural law, but just a manifestation of the fact that parallel programming in general, and HPC in particular are essentially *unsolved technical problems*. They will be considered *solved* when developers without expertise in parallel programming are able to create programs delivering near-optimal parallel performance. In other words, HPC software development will be considered *solved* when there is no need for having HPC software experts.

The history of technology is full of examples in which human jobs are deprecated not because machines could simply *do the job*, but also because they could do it *better* –i.e., *faster*, *safer* and *cheaper*–. For instance, in the context of the automotive industry, driving will (soon) be considered *solved* in the moment in which human drivers become not only *unnecessary*, but also *forbidden* for safety and performance (with respect to road congestion) reasons.

From a programming and software standpoint, the *HPC-programming solvability* will be the result from joint *automatization* and *abstraction* efforts able to dilute the mentioned *performance vs. abstraction* tradeoff, thus fully decoupling the user *problem* –i.e., the application intended to be solved by means of parallelism– from the *solution* or *tool* –i.e., the parallel platform–. This is far from being the case in current programming models: the efficient use of *the tool* is itself an unsolved problem, as in general it requires a considerable amount of expertise.

Programming models exposed in Section 1.2.2 have demonstrated the viability of promising programming patterns originally devised in the functional world, such as data-flow, task-asynchrony, continuations, strong-typing and generic algorithms. However, there is not a *sufficiently-generic* yet *high-level* programming model that permits the application developer to seamlessly and efficiently exploit parallel resources without having any parallel programming knowledge. In many of these models, parallel programming knowledge is still needed to ensure both correctness and performance even for applications exposing moderately irregular parallel patterns. Moreover,

most of the mentioned *library-based* programming models expose a considerable big API, which introduces an additional programming effort.

With this in mind, this thesis aims at providing generality without sacrificing programmability or potential for performance. The presented approach goes in the direction of *generality* also inspired by the success of other *non-generic* / *application-based* programming models and libraries (targeting linear algebra, machine learning and data analytics applications), such as MAGMA and PLASMA [5], TensorFlow [1] or Hadoop [155]. These *application-tailored* frameworks are able to target heterogeneous and distributed parallel platforms, delivering high parallel performances *out-of-the-box* without requiring the user to have any parallel programming background.

The success or failure of generic-oriented models in the road toward exascale will ultimately depend on the mechanisms provided to the users to simply expose new application characteristics that permit the runtime to choose and exploit them according to the underlying hardware characteristics. Under these premises, the term programmability should be redefined or expanded, not necessarily focusing on metrics like lines of code or development time, but rather on the amount, variety and wealth of information that the user is able to expose and communicate to the underlying runtime. As an example of current model limitations, given a parallel application instance and a parallel platform (most likely with some degree of heterogeneity), it is straightforward for the user to expose tasks to the runtime in which the application can be decomposed. However, it is still unclear how to optimally divide and distribute the computation so that performance is maximized. For this reason, the user is still in charge of finding critical pararameters such as task granularity (that ultimately determines potential inter-task parallelism) or intra-task threading granularity, whose optimal value depends on the application size and obviously target platform. This limitation not only has an impact on the development effort, but also implies a clear drawback in terms of performance and/or portability: if the underlying platform changes, or the application size is modified, a re-calibration of these manually-tuned parameters becomes mandatory, hence the programmability is reduced.

The main user concern should be to execute the application as fast or efficient as possible with minimal development effort. From this perspective, when facing a programming decision that might have an impact on current or future executions with different input parameters, or future executions in future platforms, a programming model should provide an interface by which the user could expose ways to declare different execution possibilities that the runtime system might take. In a sense, this is an effort toward *(i) complexity formalization* (complex development decisions are systematically defined and delegated) and *(ii) execution abstraction* (how the execution proceeds is abstracted away from user intent).

Execution relaxations refer to user-defined expressions aiming at expanding the actions that the runtime system can take. The goal is to propose a programming paradigm in which the user can define particular execution relaxations when *application*- and / or *platform*-sensitive decisions may affect the performance and / or the portability. In summary, the fundamental idea is to provide ways in which the user can systematically shift the *cost of decision* during development time to an automatic system (i.e., the runtime), with the objective of decoupling the *problem domain* –i.e., the user application– from the *solution domain* –i.e., the parallel resources–, to ultimately *push* the *solution-domain*-specific decisions to an automatic system. Noting that the user is *extending* or *expanding* the space of possibilities to be taken at run-time, a new parallel programming paradigm by which *user-defined execution relaxations* yield *runtime-driven expanded executions*, is proposed in this thesis.

In the following sections, a set of possible actions or operations that are (or could be) taken at run-time are grouped in two categories. The first category (Section 1.3.2) encompasses a set of *operations*, *actions* or *mechanisms* that some of the current programming models, runtimes, engines or libraries exposed in Section 1.2.2 already implement and perform at run-time –i.e., actions are taken dynamically, possibly considering dynamic information–. On the contrary, the second category (Section 1.3.3) encompass another set of *operations*, *actions* or *mechanisms* that are *not*

systematically managed in a runtime fashion in general-purpose parallel programming models, but are typically managed solely by the developer in a manual or *static* fashion. This exposition motivates the management of these mechanisms from a purely automatic runtime perspective, treated jointly or in an isolated way.

1.3.2 Currently runtime-driven work

1.3.2.1 Task-to-processor mapping

The emergence of heterogeneous architectures in the last decade has motivated the development of runtime systems supporting heterogeneous execution contexts. From the *execution relaxations* rationale, this is the response given by HPC runtime system developers to the ambiguity that the application developer has to face when multiple processing devices are available to run a given application (or parts of it). With this regard, the exposition from the user side of several kernel implementations for a task, targeting different architectures would be a kind of execution relaxation, as the specific target device in which the task will be computed is *relaxed* and delegated to the runtime.

From the performance point of view, task-to-processor mapping is commonly addressed from the perspective of the *task scheduling problem* [34] under two different approaches: *run-time / online / dynamic* vs. *offline / static* methods. From an *offline* or *static* point of view it is classified as a NP-hard combinatorial optimization problem, which it is in practice *exactly solvable* only for relatively few tasks running on relatively few processors. More realistic *run-time* task scheduling scenarios require however *online* or *dynamic* scheduling approaches, in which tasks are scheduled as they are generated. Even if the task Directed Acyclic Graph (DAG) representing the whole application is fully known in advance (which is uncommon), fixing a static task-to-device schedule in the application code would constraint too much the execution. Even if this assignment is optimal for a certain problem size and given platform, and assuming the tasks have been perfectly profiled for every target processor, those schedules may become too rigid to be successfully applicable in a run-time context, as they may not be able to adapt to the intrinsic uncertainty of run-time conditions. On the contrary, dynamic task scheduling policies, even though suboptimal and short-sighted (task DAG is usually not known in advance), are preferred over static algorithms in realistic scenarios (a great number of tasks running on a massively parallel computer) due its low complexity and run-time adaptability.

1.3.2.2 Transparent data and memory management

In a context with complex memory hierarchies and a multi-processing environment, it is commonly not practical for the programmer to imperatively specify the data movements across memory spaces. Automatic mechanisms such as cache policies, data coherency protocols, and memory models based on virtual unified spaces that abstract distributed physical spaces (e.g., CUDA Unified Memory [115] and PGAS-based models [31]), let the user abstract away from data movement decisions. Automatic memory allocation and reclamation policies are also examples that relieve the programmer from the burden of dealing with memory management.

From the *execution relaxations* perspective, this is the response given by system developers and library implementers to abstract away the complexity behind data movement and memory management in architectures with several memory spaces. In this context, *where* the data is placed and *when* is it fetched from a certain memory space is no longer important from the user application point of view, as long as there is a well-implemented coherence protocol able to transparently satisfy data visibility requests at user's will respecting reasonable latency requirements. In other words, *data placement* is *relaxed* and transparently managed at run-time.

1.3.2.3 Out-of-order task execution and priorization

The *computation re-ordering* or *out-of-order execution* is not only considered from the task-parallel perspective, but it has been implemented at microarchitectural level decades ago, as microprocessors started to be equipped with heterogeneous processing units and long instruction pipelines. From a data-dependency scope, out-of-order considerations are motivated from the fact that ordered computations expressed in the user code which are not ready to be done at a given time should not block others that are ready. Automatic re-ordering has potentially a big impact in instruction / task processing throughput depending on the amount of instruction / task parallelism. Also, considerations about task importance from critical-path analysis [159] motivate the prioritization of some instructions / tasks via reordering.

From the *execution relaxations* perspective, automatic re-ordering can be viewed as if the user-defined order (of instructions / tasks, derived from source code) is *relaxed*, so that it can be automatically managed at run-time aiming at latency hiding and performance improvement.

1.3.2.4 Task-wise variable voltage-frequency scaling

In a context in which tasks are sharing a computing resource and performance is the optimization objective, an increase of clock frequency or voltage (see DVFS [153]) on the cores in which the task is running may be beneficial –according to some application performance measure– for those tasks that are more *critical* than others. Similarly, if energy efficiency and / or power capping constraints enter into consideration, the decrease in clock frequency could be used to satisfy a given energy-efficiency and / or power constraint.

DVFS is implemented in modern hardware and many Operating Systems (OS) as a mechanism to save power and to prioritize the execution of applications, being also a popular research field in energy- and power-aware HPC contexts. From the *execution relaxations* perspective, DVFS actions are properly abstracted away from the user side, as they are not related to the user intent (i.e., the *problem domain*) and are automatically taken at run-time. Hence, core voltage and frequency could be contemplated as *relaxed* from the user point of view (although this relaxation may *not* be considered *user-defined* if it is implicitly equipped at OS-level).

1.3.3 Potentially runtime-delegable work

This section encompases instances that could be seen as possible execution relaxations, but are not currently implemented in *general-purpose*-oriented parallel programming models.

1.3.3.1 Task-wise variable thread scheduling

Many of the aforementioned programming models consider each individual task as the minimum scheduling unit, and an individual core as the basic execution target for ready tasks. Thus, task codes are internally executed in a sequential fashion, and the runtime extracts and exploits parallelism by executing sequential tasks in parallel as data dependencies are satisfied. However, if tasks are internally *threadable*, the classic task scheduling problem can be extended with this additional degree of freedom [147, 99]. Armed with this extension, the runtime would not only be in charge of *mapping* ready tasks to available cores, exploiting *inter-task* parallelism, but it would also decide the degree of *intra-task* parallelism depending on the architectural and runtime states, and on the specific features of each individual task (e.g., granularity, criticality).

From a theoretical perspective, the task scheduling problem belongs to the NP-hard class and has been one of the most studied combinatorial optimization problems in the last decades, while the *moldable-/malleable*-task variants have been lately studied from theoretic grounds [99] and application-oriented research [105]. In particular, the problem of scheduling *moldable* tasks has been considered from the static perspective as a combinatorial optimization problem.

When a task is internally parallelizable –e.g., by host OS threads or *lightweight* GPU SIMT-threads– a similar discussion regarding internal task performance can be exposed depending on thread granularity: for a given task grain size there will be a specific amount of threads or thread-granularity that will provide optimal performance. Specifically, in a DAG task-scheduling context, tasks do not only exhibit data dependencies but they also share limited resources.

The problem of variable intra-task parallelism at run-time is relevant in general scenarios in which a heterogeneous set of threadable tasks exhibit data dependencies between them. In these applications, considerations in terms of task criticality and critical path analysis arise from two sources: *(1)* tasks are heterogeneous in terms of computations and data granularity, and *(2)* task inter-dependencies introduce limitations in potential task parallelism. Heuristically, a greater amount of cores should be assigned to those critical and coarse-grained tasks in favor of less critical and finer-grained tasks in order optimize time-to-solution. These kind of decisions, however, are still taken by the user / developer, that needs to decide the optimal value for a given operation and architecture; in the case of delegating them to the runtime task scheduler, the decision is usually static and fixed for the overall computation. In both cases, an incorrect decision could imply a dramatic impact on performance and / or energy efficiency.

Based on these observations, from the *execution relaxations* perspective it could be beneficial to let the user *relax* the internal parallelism of the tasks, so that it can be automatically resolved at run-time depending on the aforementioned factors.

1.3.3.2 Task partitioning

Heterogeneous architectures do not only expose *task-to-device* mapping ambiguity (which itself motivated *task-to-processor* relaxations in Section 1.3.2.1), but they also motivate issues regarding *data granularity*. When the computation is (or can be) partitioned, the granularity of the tasks in which the original task has been partitioned (i.e., *children* tasks) typically affects the amount of available task parallelism. Also, it may also dampen the performance of the children tasks depending on the devices to which they are maapped. From a theoretical approach, Divisible Load Theory [149] emerged as a tool to model the problem of partitioning independent data into a set of distributed processors with the goal of finding the best data distribution. Task scheduling on heterogeneous architectures has explored –either from theoretic, simulation and runtime scopes– the impact of setting different task granularities adapted to the underlying characteristics of processors –e.g. massively parallel GPUs demand a coarse data stream in order to maintain its SIMD processors busy, while more general purpose CPUs achieve its peak performance for finer granularities–. However, these studies tend to focus on a specific application targeting a specific parallel system, so little can be inferred in terms of generic solutions to the problem.

Following the *execution relaxations* rationale, the user should be able to *relax* data partitions by specifying a set of *granularities* for certain applications and input sizes on certain platforms. According to this philosophy, the user should be able to push these *granularity-related* ambiguities to the runtime when there is no straightforward decision in this sense, and also when the same code is supposed to be compiled and run in other platform (in which optimal performance could require other granularities).

From a programmability perspective, some of the programming models presented in Section 1.2.2 provide API calls to partition data (e.g., [12]) or support for explicit data-layout specification (e.g., [52, 104]). However, none of them provide the sufficient expressivity to let the user relax data granularities, so that these decisions can be delegated to the runtime system for generic task- and device-wise dynamic (and possibly hierarchical) data partitioning.

1.3.3.3 Task scheduling on hardware-specific extensions

The popularity of SIMD parallel patterns in many algorithms and HPC applications have encouraged hardware developers to extend the Instruction Set Architecture of microprocessors to include

additional specialized vector processing units. In this regard, SSE, AVX2 and AVX512 instruction extensions implemented in CISC-based x86 architectures, have proven to be efficient mechanisms to exploit SIMD parallelism, showing to be a competitive approach in relation to fork-join patterns or GPU-based SIMT parallelism. Vector instructions have also became popular in simpler RISC-based systems such as ARM –in the form of NEON and SVE [136]–, and POWER microarchitectures. Concerning general-purpose GPU processors, modern NVIDIA Volta GPU architecture is equipped with *Tensor cores*, which are specialized execution units able to perform very fast mixed-precision matrix-multiplications in few cycles.

However, in these modern architectures –either RISC, CISC or GPUs–, due to the power demands, scarcity of these units and the temperature and power constraints of the chip, it may not be possible to use all of these processing units available in the chip simultaneously at design clock frequency. Therefore, these specialized units are themselves a scarce resource whose concurrent and efficient utilization could be contemplated as a scheduling problem. With this in mind, the context in which a set of tasks is competing for processing resources –in this case vector or tensor units–, opens the question about *for which tasks should the user enable these processing resources*, so that the overall execution of these concurrent tasks is globally optimized. Moreover, noting that incoming ARM SVE extensions admit variable vector length instructions at run-time, the previous question is widened (maybe even addressable under the mentioned *moldable task scheduling problem*), and the programming ambiguity is translated to the user in the form of a more complex *task-to-vector-width* scheduling decision. Hence, with regard to the *execution relaxations* point of view, either this *vector-width* decision, together with the previous scheduling decisions –e.g., whether to enable tensor cores or not for a given task–, could be considered as possible *execution relaxations* that the user could define and delegate to the runtime for automatic resolution.

1.3.3.4 Mixed precision computing

Mixed precision computations on heterogeneous and GPU platforms have shown practicality in many applications (e.g., linear algebra [13] and computational physics [60, 38]) to speed-up computations.

From the application developer perspective (who would ideally just aim at getting the result from a computer simulation as soon as possible, using a parallel computer), and assuming that *how the computation is performed* is a question that belongs to the *solution domain* (thus solvable under the automatic runtime-oriented side), then *task precision casting* could be viewed as a possible *execution relaxation*. After compiling user-defined *task-to-precision* relaxations, in which specific tasks could be annotated with a set of specific floating-point precisions on which they could operate, final task-to-precision decisions could be automatically and dynamically taken by the runtime, accounting for some pre-defined optimization goal(s).

1.3.3.5 Task-wise algorithmic reimplementation

A task reimplementation could be considered as a possible user-defined *execution relaxation* that could be systematically delegated to the runtime. An algorithmic reimplementation could be worth considering for those applications that expose a kind of *algorithmic freedom*. For example, an array sorting may admit multiple implementations such as *merge-sort*, *bubble sort*, *quick sort*, *radix sort*, etc. Each implementation exposes different characteristics in terms of average performance, amount of parallelism or memory footprint, each of which may be more appealing that the others depending on the underlying execution context and the problem size. This is also the case in many linear algebra problems, in which the requirement (or the application *problem domain*) is simple –e.g., solve a linear system–, but the algorithm yielding the best execution according to some metric is uncertain (i.e., the application *solution domain* presents ambiguities). For instance, depending on the problem size, sparsity of the matrices, condition number, required precision, and underlying parallel architecture, it is not straightforward to anticipate the algorithm which would

perform the system resolution in the fastest and / or most energy-efficient way.

1.4 Summary

The general objectives of this thesis and the chapter structure overview are exposed next. The core hypothesis of this work is that the mechanisms just exposed, that are normally handled statically by the user, can be *delegated* (and integrated with the already-automatized work) to an automatic runtime system from a general-purpose programming model standpoint.

1.4.1 Objectives

The previous exposition of possible *execution relaxation* instances in Sections 1.3.2 and 1.3.3, together with the current limitations regarding programming paradigms and parallel programming models presented in Section 1.2, motivate the developments presented in this thesis.

The departing point is a set of appealing ideas from imperative and functional paradigms, together with already successfully proven patterns of modern parallel programming models, based on *task-asynchrony*, *data-flow*, and high-level language abstractions. In summary, the general goal is to address the current limitations regarding HPC development, aiming at enhanced programmability and performance portability. Specifically, this thesis proposes a new programming paradigm or framework in which the execution complexity can be expressed by the user for consequent delegation to an automatic system (i.e., the runtime), not letting this expressivity to harm or limit the attainable performance.

This objective can be further particularized into three specific steps, namely:

1. Aim at highest-level abstractions regarding interfaces, encouraging composability, modularity and reuse over monolithic designs, while also strictly respecting a full sequential development style –i.e., without requiring the explicit use of any *concurrent-* or *parallel-specific* actor, synchronization tool or expression– from the user side.

2. Propose new ways in which rich and high-performance computations can be attained without the need of sacrificing high-level abstractions and sequential programming style.

3. Implement previous ideas in the form of an usable runtime framework and extract experimental results that validate the proposed ideas.

1.4.2 Thesis organization

Chapter 2. Static relaxed execution. Motivation and limitations. This chapter motivates the use of *relaxed execution* approaches from a static standpoint. Specifically, it explores the potential performance and energy consumption gains derived from hierarchical task partitioning and variable intra-task threading for a task-based implementation. The obtained results motivate via simulation the relaxed execution approach and provide estimations for potential gains. The chapter, however, ends by listing the drawbacks of the static execution, providing a starting point from which a dynamic model can be developed.

Chapter 3. Dynamic relaxed execution. A programming model architecture. This chapter introduces a programming model architecture for the relaxed execution paradigm. It proceeds by introducing a new task-executor taxonomy, in which the abstract concept of *task*, *executor* and *allocator* are defined and combined in a systematic fashion. The chapter advocates a new programming model paradigm following the STSE model and exposes how it can be leveraged by means of *scheduling execution expansions* at run-time, in order to boost a pre-defined output metric (e.g., performance).

Chapter 4. STEEL. Design principles and implementation. This chapter exposes a detailed description of the STEEL programming model implementation, together with use cases of the C++ API provided to the user to exploit the STSE model, including executor deployment.

Chapter 5. Use cases and experimental results. This chapter extends the description of the API in terms of specific examples and use cases, and provide qualitative and quantitative results for a number of applications in order to illustrate how changes in STEEL-leveraged codes are translated and exploited into changes in execution paths by means of the underlying runtime scheduler. These use cases stress specific aspects such as data management, task execution management and executor hierarchy deployment.

Chapter 6. STEEL as a functional model. This chapter re-exposes Chapter 3 and Chapter 4 ideas in the form of a model written in functional language, showing the benefits –regarding *abstraction*, *composability* and *side-effect encapsulation*– and limitations –regarding *context-aware executions*– of an eventual purely functional implementation of the STEEL model.

Chapter 7. Conclusions. This chapter summarizes the contributions of this thesis, exposes future enhancements to the current runtime implementation and motivates future research topics.

...grab Reality, take a slice.
The details, to be removed.
Build the model, smoothed.

If Nature throws her dice,
exposed is the fact
and rigor is attacked,
grab Reality, take a slice...

2

Static relaxed execution. Motivation and limitations

This chapter develops *relaxed execution* ideas from a static perspective, using a well-known benchmark –the Cholesky factorization– as a driving and motivating example. The first goal of the chapter is to state the problem of lack of task parallelism and processor starvation and to propose two different solutions from combined optimization approaches under the light of the *relaxed execution* paradigm. Secondly, these approaches are analyzed under a set of criteria, such as performance, energy minimization, power constraints, and programmability.

Specifically, after introducing the motivation of the static approaches and the use case in Section 2.1, two models are proposed and solved using two different approaches (approximate and exact) in Sections 2.2 and 2.3, respectively. Finally, Section 2.4 summarizes the exposed results and highlights the inherent limitations of the introduced static-based strategies, motivating its application to future programming models and runtime systems.

2.1 Introduction

Although the ultimate goal of the thesis is to motivate a new parallel programming model meant to be implemented for dynamic task scheduling scenarios, the following *simulation-based* approach seeks to model these scenarios introducing simplifications in order to prioritize determinism and result repeatability of simulated, cheaply-generated and high-quality schedules. With this regard, data transfer and task execution delays are profiled and modeled as fixed –not subject to any noise or variation–, neglecting any runtime-intrinsic stochasticity and parallel synchronization overheads.

These simplifications will prove to be very useful in order to incorporate *granularity considerations* –in terms of task grain size and task intrinsic thread parallelism– jointly with the *classic* task-based scheduling decisions, such as task-to-processor mapping and out-of-order task execution. Note that some general-purpose task-based runtime systems already support heterogeneous task-to-processor mapping and out-of-order task execution (see Section 1.2.2); however, it is still unclear how to estimate the potential benefits –in terms of performance and / or energy consumption, for example– expected when some of the *actions* exposed in the previous chapter are *combined*, in particular task-to-processor mapping (Section 1.3.2.1), task-to-threading (Section 1.3.3.1), task partitioning (Section 1.3.3.2) and out-of-order task execution (Section 1.3.2.3).

In the following, departing from a well-known linear algebra use case presented in Sec-

tion 2.1.2, Sections 2.2 and 2.3 expose two simulation models designed to provide estimations for the potential benefits of future task-parallel runtime schedulers able to systematically make task-aware partitioning and threading decisions at run-time.

2.1.1 Extensions to the classic task scheduling problem

In the following, two different scheduling extensions are considered, incorporating task granularity and task thread parallelism as additional parameters into the *classic* task scheduling problem (see Section 1.3.2.1). Specifically, Table 2.1 summarizes the characteristics of the analysis exposed in the following sections. The goal of this analysis is to give estimations that quantify the potential benefits of an hypothetical runtime task scheduler able to perform efficient (joint or independent) granularity-aware and threading-aware task scheduling decisions.

	Section 2.2	Section 2.3
Extension	Variable data granularity	Variable threading
Test platform	Heterogeneous SMP-GPU	SMP
Approach	Approximate iterative	Exact MILP
Measurements	Performance	Power-constrained performance & energy

Table 2.1: Summary of the analysis for relaxed granularity.

2.1.1.1 Relaxing task granularity, heterogeneous scheduling and out-of-order execution

In Section 2.2, task granularity is considered as an additional *degree of freedom* appended to the task scheduling problem –i.e., in this model, task partitioning decisions (see Section 1.3.3.2) are considered jointly with task scheduling decisions (Section 1.3.2.1) and out-of-order task execution (Section 1.3.2.3) in a heterogeneous processing context. Thus, task granularity is a tunable knob by means of *parametrized task partitioners*: functions that, given a numeric value, decompose the computation into a set of inter-dependent tasks whose execution yield an equivalent result. The hierarchical characteristic of this *extended scheduling-partitioning* problem makes its formulation as a linear model not practical (if not impossible); for this reason, an approximate iterative solver is proposed, implemented and applied to it in order to provide performance estimations for this extended task scheduling model.

2.1.1.2 Relaxing task threading and out-of-order execution

Similarly, in Section 2.3, the number of threads assigned to a task is considered as an additional degree of freedom included to the task scheduling problem. As such, *task threading* decisions (see Section 1.3.3.1) are jointly considered together with task scheduling decisions and out-of-order task execution. In this approach, a linear model (or a *linear program* – 'LP') is proposed and solved using a state-of-the-art solver engine, targeting power-constrained performance and energy optimization.

2.1.2 A driving example: Cholesky factorization

The blocked Cholesky factorization decomposes an $n \times n$ symmetric positive definite matrix A stored by $s \times s$ blocks of dimension $b \times b$ each, into $A = LL^T$ where L is a lower triangular matrix. At run-time, the outer loop in the code depicted in Listing 2.1 that calculates the Cholesky factorization divides the operation into a number of sub-tasks that, when executed under a task-parallel paradigm, generate a task DAG as that shown in Figure 2.1. In the task DAG, nodes correspond to different tasks, and edges denote data dependencies between them.

Listing 2.1: C implementation of the blocked Cholesky factorization.

```
1  void cholesky (double *A[s][s], int b, int s) {
2    for (int k = 0; k < s; k++) {
3      chol (A[k][k], b, b);              // Cholesky factor. (diag. block)
4
5      for (int j = k + 1; j < s; j++)
6        trsm (A[k][k], A[k][j], b, b);  // Triangular solve
7
8      for (int i = k + 1; i < s; i++) {
9        for (int j = i + 1; j < s; j++)
10         gemm (A[k][i], A[k][j],       // Matrix multiplication
11               A[i][j], b, b);
12
13       syrk (A[k][i], A[i][i], b, b);  // Symmetric rank-b update
14 } } }
```

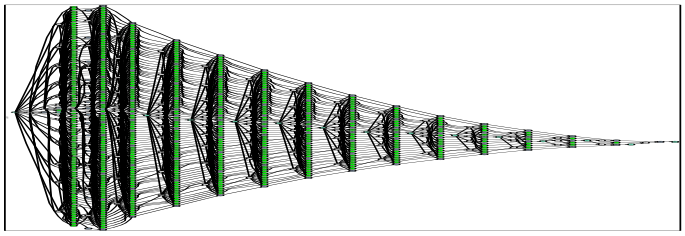


Figure 2.1: Fine-grain Cholesky task DAG.

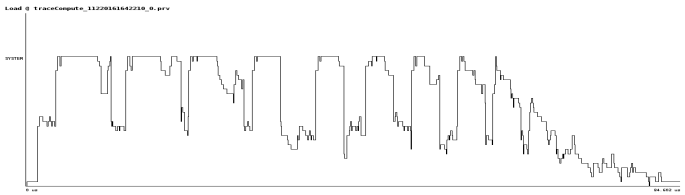


Figure 2.2: Compute load trace (number of ongoing tasks) for the DAG in Figure 2.1.

Figure 2.3: (a) Task DAG in which the computation evolves from left to right, and (b) compute load trace generated by the Cholesky factorization in Figure 2.1 in terms of number of tasks in execution, for a problem size $n = 16384$, and block size $b = 1024$.

The Cholesky factorization is an appealing example for the purposes of this chapter: (i) it exhibits different sub-task types (CHOL, SYRK, GEMM and TRSM) and complex data dependencies among them, and (ii) it features different degrees of parallelism as the factorization evolves. Consider, for example, how the DAG depicted in Figure 2.1 reduces the available task parallelism (that is, the number of tasks that can be executed in parallel, typically related with the width of the DAG) at the first stages of the factorization, and (in a much larger extent) at the last stages. This is usually translated into processor load patterns like that shown in Figure 2.2, that represents a timeline of an execution of the Cholesky factorization on a highly heterogeneous platform, composed by 28 Intel Xeon cores and 3 different GPUs. The plot represents the number of active processors as the execution proceeds. Areas with reduced load illustrate *workload imbalance*, which can be caused by two different factors: different processing capabilities of each processor type, and lack of potential parallelism on specific stages of the execution. The first can be alleviated by *scheduling* heuristics (e.g., mapping tasks in the critical path to fast processors), but the second is inherent to the algorithm, and can be alleviated (as long as tasks are internally parallelizable) by non-uniform task *partitioning* or variable intra-task threading in order to expose additional parallelism at run-time.

Data block (or *tile*) size is a crucial parameter in task-parallel executions, as it ultimately determines the amount of available parallelism, and the efficiency of each individual task execution.

In Figure 2.1, block size is determined by `b`; note that, typically, larger block sizes usually imply higher performance per individual task, and smaller block sizes tend to expose higher degrees of parallelism, which naturally drives to better processor occupation. In addition, different block (task) sizes are desired for different architectures, and even for different problem dimensions in the same architecture.

In general, the blocked Cholesky factorization is a particular case of a more general situation in which a set of heterogeneous and interdependent tasks require shared computational resources. With this regard, execution performance may be seriously limited due to a complex interplay between different factors, such as task granularity, task criticality, problem size, application inter-task and intra-task parallelism, and available hardware concurrency.

Altogether, these observations motivate *(i)* the exploration in Section 2.2 of new techniques that explore the impact of *heterogeneous* or *non-uniform* task partitioning on the performance and resource occupation of heterogeneous architectures and *(ii)* the exploration in Section 2.3 of the effect of *per-task variable threading* in terms of power-constrained performance and energy optimization in a homogeneous parallel processor.

2.2 Heterogeneous scheduling-partitioning with HeSP framework

A simulation framework named Heterogeneous Scheduler-Partitioner (HeSP) that jointly addresses the *task scheduling* and *partitioning* problems in a simultaneous fashion is presented next. Based on per-task and data transfers performance models, HeSP adds an additional degree of freedom to typical task scheduling policies by considering a joint task partitioning / scheduling approach. The framework proceeds by finding a set of task partitions that divides the initial workload into a number of sub-tasks with different granularity, that better fit to the underlying hardware resources at a given execution point. The approach drives to considerable performance improvements and more efficient resource utilization.

HeSP is a simulation framework that *approximately* solves a *task scheduling-partitioning problem* targeting heterogeneous architectures. At a glance, the input to this problem is *(i)* a hardware platform description where several finite-size memory spaces are connected according to a certain network topology, together with a (possibly heterogeneous) set of processors associated with them; and *(ii)* a single task to be computed in that platform. A solution to this problem consists of *(i)* a set of tasks –presumably with different granularity–, related by arbitrary data dependencies and equivalent to the input task, and *(ii)* a task-to-processor mapping, considering *performance* as the optimization goal.

2.2.1 Features of the scheduling-partitioning simulation framework

Besides supporting recursive task partitioning, HeSP is designed to be a realistic framework that simulates not only current heterogeneous architectures, but also state-of-the-art scheduling and data management policies on task-parallel executions.

2.2.1.1 Task and data scheduling heuristics

HeSP implements different heuristics for task-to-processor assignments. Random Processor (R-P) and Fastest Processor (F-P) selection policies consider such processor choices among idle processors at the time in which task is ready (or *release*) to be executed. The Earliest Idle Time Processor (EIT-P) and Earliest Finish Time Processor (EFT-P) policies select the processor becoming idle first, and the processor finishing first if that task is assigned to it, respectively. EFT-P estimates the finishing time accounting for eventual data transfers if needed. Task scheduling order is specified by choosing between First-come First-served (FCFS) or Priority-List (PL) choices. In PL, a priority list is built by sorting tasks by their critical times in decreasing order. Critical times are computed by averaging task processing time for all processors, and propagating them throughout

the task DAG by a backflow algorithm. The combination of PL and EFT-P heuristics is practically identical to the well-known *HEFT* scheduling algorithm [146].

2.2.1.2 Simulation of complex memory spaces

When several independent memory spaces are present, HeSP considers data movement for scheduling decisions, considering individual memory spaces of each accelerator as software caches of a main memory space, typically tied to CPUs. Common caching policies like *write-through* (WT), *write-back* (WB) or *write-around* (WA) are used. When a task is about to be scheduled to a processor and its data dependencies are not present in the local memory space, the required data transfers are issued from the source memory space to the local memory space using prefetching schemes.

2.2.1.3 Performance and data transfer models

HeSP estimates computing or transfer times relying on analytical models extracted a priori for each task / data type and size mapped to any existing processor / interconnect in the system. These estimations are considered when making both scheduling and partitioning decisions, jointly or in an isolated fashion. The quality of these models will ultimately determine the accuracy of the simulated scheduling results.

2.2.1.4 Recursive task partitioners

Task *partitioners*, specified for each partitionable task type, are just blocked algorithms (see, for example, Figure 2.1 for the specific case of the Cholesky factorization) with an input parameter that specifies the data granularity / degree of parallelism of the following partition. On a partitioner invocation, the corresponding emergent sub-tasks are managed by HeSP by introducing them in the respective task DAG which the partitioned task belongs to. Figure 2.4, starting from an initial CHOL task –Cholesky factorization–, illustrates how three successive task partitions –corresponding to respective CHOL, TRSM and SYRK blocked algorithms– affect the prior task DAG, and the corresponding data partitions they induce.

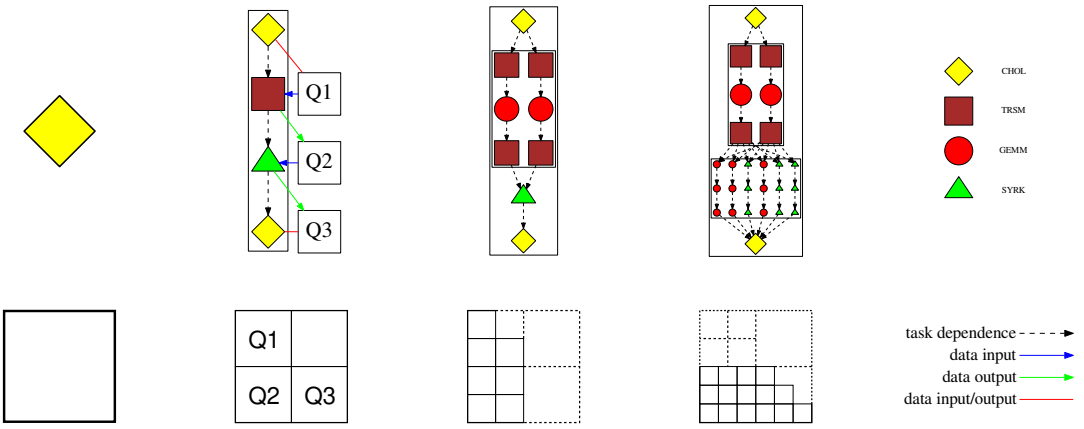


Figure 2.4: Three successive task partitions and corresponding partitioned data blocks. Tasks and their related data dependencies –Q1, Q2 and Q3 quadrants– are shown only for the first partition for the sake of clarity.

Note that any task can be partitioned again as long as its dependent data blocks can be divided consistently, so an extremely hierarchical task DAG can be constructed by recursively partitioning

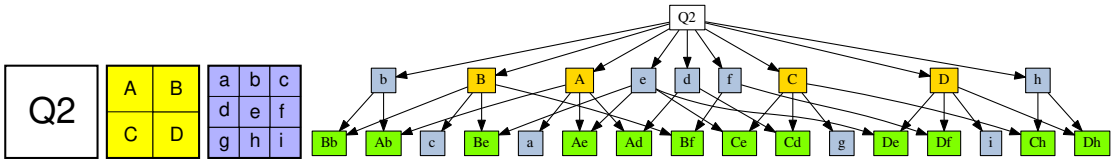


Figure 2.5: A data block (Q2 quadrant) can be simultaneously divided according to different tilings (yellow and blue tilings, corresponding to TRSM and SYRK task partitions in Figure 2.4). Additional data block descriptors (green) are constructed to represent partial overlaps between not nested data blocks.

its tasks. A *task cluster* is a set of tasks generated from a single task partitioning, being the source task their *parent*. The *DAG / graph depth* indicates the maximum number of nested task clusters, and the *DAG / graph width* refers to the maximum number of tasks that can be executed in parallel. For instance, in the four task DAGs in Figure 2.4, the corresponding depths are 0, 1, 2 and 2, and their widths are 1, 1, 2 and 6. Dependencies between tasks, shown as dashed arrows, represent RaW, WaR and WaW constraints.

2.2.1.5 Recursive data partitioning and data coherence management

New tasks generated after a partition reference to finer-grain input and output data dependencies, which are partitions of the initial data block(s) of the parent task (see Figure 2.4). HeSP implements validate / invalidate mechanisms to ensure data coherency among different memory spaces while handling asynchronous memory transfers. Since recursive task partitions induce corresponding recursive data block partitions, the existing partitioned data blocks are organized in a directed acyclic graph structure (data DAG onwards) in which nodes represent data blocks and directed links represent nesting relations between them; for example, $A \longrightarrow B$ means B is fully contained in A and A is bigger than B.

Armed with the data DAG, validations and invalidations are propagated by top-bottom and bottom-top mechanisms throughout this graph to maintain coherence. For instance, to ensure that a task can start its computation and store the result in an output block OB, not only the block OB must be invalidated on the remaining memory spaces in which the block may be present, but the hypothetical data block partitions contained in OB and the bigger blocks in which OB may be contained must be invalidated as well. Similarly, after a certain task has finished its computation updating OB, both OB and all the blocks within OB must be validated in the memory space corresponding to the processor assigned to that task.

In general, these data block partitions induced by task partitions form tree structures. However, it is possible to have a pair of blocks which intersect partially, nested within a common bigger parent block. This case shows up when two partitions of non-divisible grain sizes are applied to the same data block (for example, quadrant $Q2$ in Figure 2.4). In this case, a new data block descriptor which refers to its intersection is introduced in the data DAG as a common child node of two intersecting blocks (see Figure 2.5). With this mechanism, together with the validate / invalidate propagation mechanisms, data coherency is ensured for all possible data partitions and hierarchical data graphs.

2.2.1.6 *Approximate* iterative solver

HeSP solves the scheduling-partitioning problem by iteratively searching for those hierarchical task DAGs which best fit to a heterogeneous processing platform –according to performance optimization– given a specific combination of the aforementioned scheduling heuristics –processor selection heuristic and task ordering–. A schedule stage is followed by a partition stage for each iteration, being the number of iterations a user-defined parameter.

At the partition stage, HeSP chooses a candidate task to be partitioned or a candidate task cluster to be merged back / re-partitioned with a different granularity. A global analysis of the schedule-partition done in the previous iteration can provide useful information –i.e., bottleneck identification, number of idle resources, or too fine grained tasks– to help the iterative process to converge toward a better overall schedule-partition.

The partition procedure is based on two stages: *(i)* task selection to build the candidate list, and *(ii)*, sampling type to choose the final candidate. For *(i)*, HeSP implements three different policies: *All*, *CP* and *Shallow*. *All* selects all tasks of the previous step, *CP* selects only tasks belonging to the critical path, and *Shallow* selects those tasks whose depth (i.e., number of task clusters that contain it) is minimal. All existing task clusters are candidates to be merged back or repartitioned. For each added candidate, a positive score is computed by subtracting the current cost delay by an estimated cost after its eventual partition or merge, being this estimation based on the available parallelism at its scheduling time of the previous step. For each candidate whose data dependencies have a characteristic size d, a partition parameter $p \in (0, 1]$ is chosen such that new tasks created after the eventual partition will depend on data blocks of size $b = p \times d$[1]. The more available parallelism is exposed, the smaller p is set in order to generate a higher amount of parallel finer-grained tasks.

In the second stage *(ii)*, a final selection among all candidates is done according to *Hard* or *Soft* procedures. In *Hard*, the candidate with the maximum score is chosen; in *Soft*, the candidate is randomly selected such that the selection probability equals the score divided by the sum of all scores.

2.2.2 Performance results on heterogeneous architectures

In the following, HeSP is fed with model data and a heterogeneous platform description (MACHINE1, see B.1), from which a Cholesky factorization execution in single precision is simulated.

2.2.2.1 Framework validation and evaluation of scheduling heuristics

The goal of the following first set of experiments is two-fold: *(i)* to validate the results extracted from HeSP by comparing them with an equivalent execution using a real task scheduler (OmpSs [51]) and *(ii)* to illustrate the impact in performance of several scheduling policies in HeSP when *homogeneous* or *uniform* task partitions are employed.

Each point in the line labelled as `OmpSs` in Figure 2.6 (left) corresponds to the best scheduling performance out of 20 OmpSs executions for each grain size. These 20 trials were set to let OmpSs *Versioning* scheduler [122] improve itself by gathering enough task execution delay samples for each task type / size and processor. The other two curves –HESP-REPLICA-PM and HESP-REPLICA-RD– denote the performance attained by HeSP when applying the same task-to-processor mapping extracted from the best OmpSs trial, using previously extracted performance models (PM) and the real OmpSs task delays (RD), respectively, for each uniform tiling.

Differences in performance between HESP-REPLICA-RD and OMPSS points are a measure of the OmpSs runtime overhead, while the differences between HESP-REPLICA-PM and HESP-REPLICA-RD are mainly due to *(i)* the accuracy of the extracted performance models fed to HeSP, and *(ii)* possible differences between OmpSs-internall task delay instrumentation module and the instrumentation used to extract the performance models. In summary, the differences between the replicated schedules are small enough and easily explainable to assert the validity of the following results. In general, these observations reveal a qualitative matching between real and simulated workloads for all problem sizes, with deviations that can be easily explained and should not affect the quality of the following observations.

To introduce the context in which the *heterogeneous* or *non-uniform* partitioning approach takes place and its potential benefits, Figure 2.6 (right) reports the performance obtained by run-

[1] A task cluster is a candidate to be merged if $p = 1$ or repartitioned if $p < 1$.

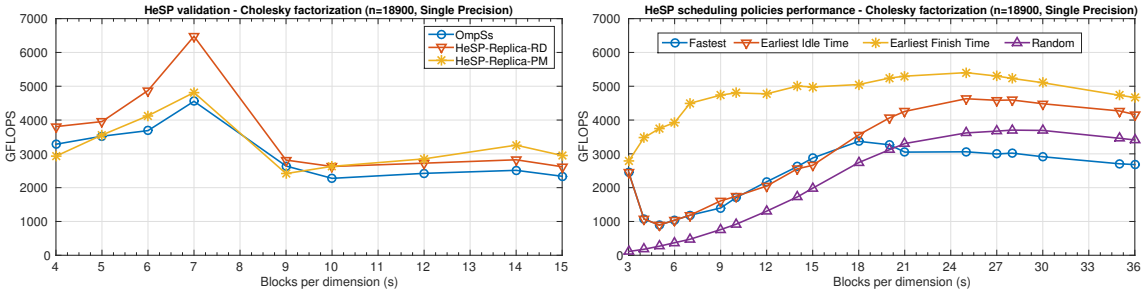


Figure 2.6: Left: Comparison between OmpSs and their simulated schedules. Right: Comparison between different scheduling policies and block sizes in HeSP.

Table 2.2: Performance comparison for MACHINE1.

MACHINE1 ($32,768 \times 32,768$ Cholesky factorization in single precision)								
	Best Uniform			**Best Found Non-uniform**				
Config.	Perf. (GFLOPS)	Avg. load (%)	Block size	Perf. (GFLOPS)	Improve. (%)	Avg. load (%)	Avg. block size	DAG depth
FCFS/R-P	3453.91	75.3	1024	4189.17	21.29	82.3	991.23	2
PL/R-P	4460.30	88.4	1024	4752.43	6.55	89.4	978.33	2
FCFS/F-P	2846.78	53.4	2048	3687.93	29.55	63.6	446.52	3
PL/F-P	3381.76	68.4	2048	3614.28	6.88	66.2	1165.70	3
FCFS/EIT-P	5650.10	91.3	1024	5747.87	1.73	92.3	1002.26	2
PL/EIT-P	6096.91	93.9	1024	6206.55	1.80	95.4	1009.91	2
FCFS/EFT-P	6581.96	23.3	2048	7569.34	15.00	63.9	412.15	5
PL/EFT-P (*)	7046.87	55.9	2048	8030.50	13.96	86.9	407.41	4

ning HeSP simulations using different scheduling policies for different *uniform* task partitions. It is worth noting that: *(i)* the optimal tile size does not only depend on the underlying architecture and problem size, but also on the selected scheduling policy; *(ii)* for every policy, performance curves follow a similar pattern, exhibiting a peak performance in a *trade-off tile size* that best balances potential parallelism and optimal individual task performance; and *(iii)* differences in performance are relevant depending on the selected scheduling policy, being even more dramatic for large tile sizes. These obvervations motivate the simulation of *non-uniform* partitioning schemes to estimate their potential for performance maximization.

2.2.2.2 Impact of non-uniform partitioning in performance

This section illustrates the main performance improvements obtained with HeSP using *All / Soft* configurations for task partitioning selection. Table 2.2 reports performance values on MACHINE1 using the best uniform and non-uniform partitions found by HeSP for different scheduling policies[2], together with additional metrics that clarify many of the concepts exposed hereafter, including average processor load, optimal / average block size and task DAG depth. The first observation worth noting is the overall improvement attained for all non-uniform task partitions found by HeSP and the overall reduction in the optimal average block size on non-uniform partitions.

Note the direct relation between the average processor occupancy and the improvements of the non-uniform partitions. For example, EIT-P with uniform partitioning yields high processor occupancy (between 91% and 98.5%), so the potential benefit expected from additional extracted parallelism is poor, ranging between 0.76% and 2.02%. Contrary, uniform partitions on EFT-P schedules yield better performance than EIT-P ones while still leaving more room for potential parallelism. Although the quality of EFT-P uniform schedules could actually leave little

[2]In all cases, Write-Back caching is employed.

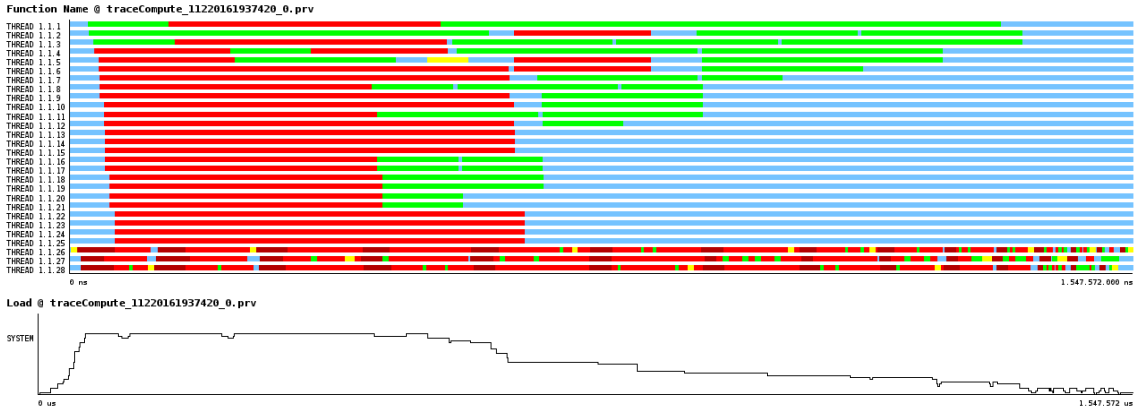


Figure 2.7: Execution traces of best uniform partitioning on MACHINE1. Task scheduling (top) and compute load (bottom).

room for additional improvements, the greater processor availability they offer permits the iterative scheduler-partitioner to find finer-grain partitions (compare Figures 2.7 and 2.8) both in terms of idle time (marked in light blue in the traces) and compute load, attaining remarkable net improvements for MACHINE1 (between 13.96% and 15%). Note also that bigger performance improvements do not only correspond with lower processor occupancy, but also with higher task DAG depths (up to 5 in MACHINE1). This observation reinforces the importance of managing multiple levels of task granularity, extending the idea of using only two degrees of granularity for two types of processors introduced in other works [157].

Note the even better improvements, with simpler –i.e. less deep– partitions, attained by this scheme when jointly applied with simpler schedulers –R-P/F-P– and naive FCFS task ordering. Since bad scheduling decisions exhibit a smaller worsening global impact when applied to a bigger set of smaller tasks, task partitions cooperating with a simple scheduler might alleviate its poor performance: under highly heterogeneous scenarios and available resources, it could be safer to partition a task rather than taking the risk of assigning it to the wrong processor.

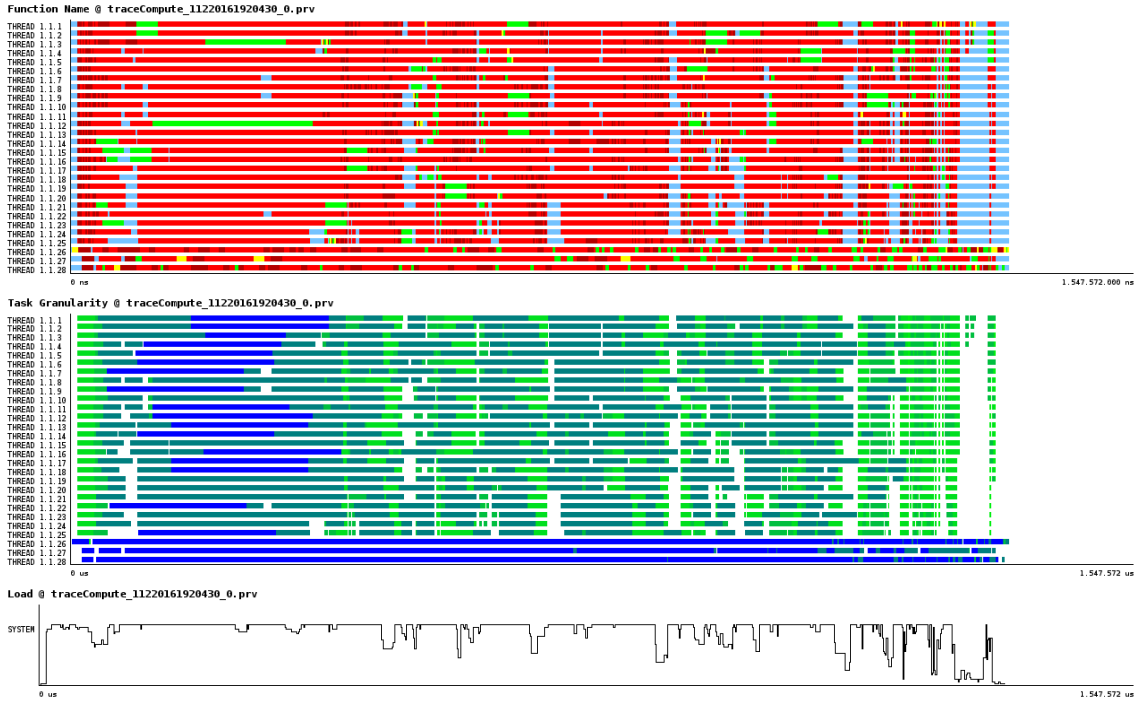


Figure 2.8: Execution traces of best non-uniform partitioning on MACHINE1. Task scheduling (top), granularity (middle) and compute load (bottom). Task granularity ranges from finer (light green) to coarser (dark blue).

2.3 A MILP framework for power-constrained variable threading

As explained in Section 1.3.2.1, the task scheduling problem belongs to the *NP-hard* class and has been one of the most studied combinatorial optimization problems in the last decades, while the moldable- malleable-task variants have been lately studied from theoretical grounds [147], [23]. In contrast, the following exposition exposes a Mixed-Integer Linear Program (MILP) model applied to the same Cholesky application, in which an arbitrary number of threadable or *moldable* tasks with dependencies and no convexity requirements are scheduled on a multiprocessor.

As in the previous section, this study ultimately targets modern task-based parallel programming models and runtime task schedulers, mainly motivated by existing challenges in terms of execution performance, programmability and developer productivity. The exposed model and its implementation targeting a state of the art solver (Gurobi [63]) are able to return *exact* solutions or high-quality bounds, thus providing useful information to runtime-scheduling developers when tuning *task-to-threading* policies in future task-based programming models.

Section 2.3.1, exposes a model and a MILP formulation that considers both makespan and energy minimization[3] under a power constraint. In Section 2.3.2, different execution instances of Cholesky factorization are simulated and solved on a modern 20-core processor as a test case, exploring six different constrained optimization scenarios targeting power-limited execution performance and energy. Finally, the improvements and execution trace characteristics of solutions with constant and variable task-to-core(s) assignments are compared.

2.3.1 Model

This section exposes a model that simulates the allocation of a set of interdependent tasks structured in a DAG on a multiprocessor with N_{max} identical cores, where each task is internally threadable, and the goal is to find a particular number of cores or *thread configuration* for each

[3]For the tested application, *makespan minimization* is equivalent to *performance maximization*.

one, such that a given global metric –either makespan or consumed energy– is minimized. In this context, each core can only process one simultaneous task, although several cores can co-operate to process a single task. Cores are modeled as indistinguishable computational units, in the sense that core identifiers or core placement in the die are not considered. With this abstraction, a task allocation is only defined by the execution start time and the number of cores / thread configuration employed, not taking into account any specific core binding. This *core-independence* assumption also requires that the execution performances of any pair of subsets of cores –each subset running a different task– are independent, so eventual inter-core coupled effects such as Last Level Cache (LLC) contention or thermal-aware power capping policies are not considered. Eventual LLC access conflicts could in principle be detrimental to this assumption, but appropriate orchestration of data placement in concurrent kernels to exploit shared caches could alleviate that. Nevertheless, a detailed analysis of this problem is out of the scope of this study for the sake of simplicity.

For each specific task and threading configuration pair, the number of cores and *dynamic* power consumption is fixed. A constant static power W_{idle} is also modeled, which contributes to the overall energy consumption in the background as the execution evolves, together with the dynamic power due to task allocations. The power at any given moment is the aggregated power of W_{idle} plus the dynamic power of all the tasks that are being executed at that moment. With these assumptions, execution delay and dynamic power of a given task instance are fully characterized by the threading configuration assigned to it.

2.3.1.1 Mixed-Integer Linear Programming formulation

Based on these assumptions, a MILP model is exposed next, in which solutions are defined by a set of binary variables representing *task-to-threading-configuration* assignments and a set of continuous variables representing task execution start times. The model **definitions** are the following.

Input parameters $N_{max}, W_{max}, W_{idle}, M$**:** The target multiprocessor is characterized by N_{max} cores and a static power of $W_{idle} \in \mathbb{R}^+$. Additionally, $W_{max} \in \mathbb{R}^+$ denotes the power budget and $M \in \mathbb{R}^+$ is an upper bound for the makespan.

Input sets T, H, P, Q, O, D, N, W**:** The application is represented by the set of tasks T to be allocated and H is the allowed threading configurations. For all pairs $(t, u) \in T$, the parameter P is the precedence matrix, being $P_{t,u} = 1$ if u computation must be preceded by t and $P_{t,u} = 0$ otherwise. Q denotes the full precedence matrix, being $Q_{t,u} = 1$ if t precedes u anytime in the DAG and $Q_{t,u} = 0$ otherwise. Note that Q defines a partial order, and in particular, if $Q_{t,u} = 0$ and $Q_{u,t} = 0$ then the executions of t and u can overlap.

In runtime scenarios, tasks are usually generated following a specific ordering, although out-of-order execution is assumed to happen in general. The set O establishes a total order such that for each $(t, u) \in T$, $O_t < O_u$ if task t was generated before u. From this total order, in-order constrained execution can be imposed. Note that for application execution correctness, O inherits the partial order of Q: $Q_{t,u} = 1 \Rightarrow O_t < O_u$.

For all $t \in T$ and $h \in H$ the parameter $D_{t,h} \in \mathbb{R}^+$ indicates the execution delay of task t when it is scheduled using threading configuration h. The value N_h indicates the number of cores required for the threading configuration $h \in H$. $W_{t,h} \in \mathbb{R}^+$ indicates the required dynamic power when t is scheduled with h.

Decision variables X, S**:** For all $t \in T$ and $h \in H$, binary variables $X_{t,h} \in \{0, 1\}$ indicate whether task t is scheduled with threading configuration h or not. Also, for all $t \in T$, $S_t \in \mathbb{R}_{\geq 0}$ denotes the scheduled start time of task t.

Auxiliary variables $\Delta, \Omega, \Phi, C_{max}$**:** For all $(t,u) \in T$, $\Delta_{t,u} = 1$ if the start time of task t is greater or equal than the start time of task u, and $\Delta_{t,u} = 0$ otherwise. Similarly, $\Omega_{t,u} = 1$ if the start time of task t is strictly less than the end time of task u, and $\Omega_{t,u} = 0$ otherwise. $\Phi_{t,u,h}$ is a representation of the cubic term $\Delta_{t,u}\Omega_{t,u}X_{u,h}$, and $C_{max} \in \mathbb{R}^+$ denotes the makespan or *time-to-solution*.

The model **constraints** are the following.

The makespan C_{max} is constrained by task end times:

$$C_{max} \geq S_t + \sum_{h \in H} D_{t,h} X_{t,h}, \quad \forall t \in T. \tag{2.1}$$

There is only one threading configuration $h \in H$ to be assigned for each task:

$$\sum_{h \in H} X_{t,h} = 1, \quad \forall t \in T. \tag{2.2}$$

Tasks depending on other tasks must wait for them to start:

$$S_u \geq S_t + \sum_{h \in H} D_{t,h} X_{t,h}, \quad \forall (t,u) \in T \mid P_{t,u} = 1. \tag{2.3}$$

Auxiliary variables $\Delta_{t,u}$ and $\Omega_{t,u}$ are set following the definition in section 2.3.1.1 and together with $\Phi_{t,u,h}$ are required to satisfy processor occupation and power budget constraints. Defined from constraints (2.4), (2.6) and (2.8), they are set to 0 if task pair t, u cannot overlap –Expr. (2.5) and (2.7)–.

$$\begin{aligned} S_u &> S_t - M\Delta_{t,u}, \\ S_u &\leq S_t + M(1 - \Delta_{t,u}), \\ \Delta_{t,u} \in \{0,1\}, &\quad \forall (t,u) \in T \mid Q_{t,u} = 0 \wedge Q_{u,t} = 0. \end{aligned} \tag{2.4}$$

$$\Delta_{t,u} = 0, \quad \forall (t,u) \in T \mid Q_{t,u} = 1 \vee Q_{u,t} = 1. \tag{2.5}$$

$$\begin{aligned} S_t &\geq S_u + \sum_{h \in H} D_{u,h} X_{u,h} - M\Omega_{t,u}, \\ S_t &< S_u + \sum_{h \in H} D_{u,h} X_{u,h} + M(1 - \Omega_{t,u}), \\ \Omega_{t,u} \in \{0,1\}, &\quad \forall (t,u) \in T \mid Q_{t,u} = 0 \wedge Q_{u,t} = 0. \end{aligned} \tag{2.6}$$

$$\Omega_{t,u} = 0, \quad \forall (t,u) \in T \mid Q_{t,u} = 1 \vee Q_{u,t} = 1. \tag{2.7}$$

The auxiliary binary variables $\Phi_{t,u,h}$ are a representation of the product $\Delta_{t,u}\Omega_{t,u}X_{u,h}$:

$$\begin{aligned} \Phi_{t,u,h} &\geq \Delta_{t,u} + \Omega_{t,u} + X_{u,h} - 2, \\ \Phi_{t,u,h} \leq \Delta_{t,u}, &\quad \Phi_{t,u,h} \leq \Omega_{t,u}, \quad \Phi_{t,u,h} \leq X_{u,h}, \\ \Phi_{t,u,h} \in \{0,1\}, &\quad \forall h \in H, \quad \forall (t,u) \in T. \end{aligned} \tag{2.8}$$

The number of cores required at each time S_t must not exceed N_{max}:

$$\overbrace{\sum_{h\in H} N_h X_{t,h}}^{t\ occupancy} + \overbrace{\sum_{\substack{u\in T\\ u\neq t}}\sum_{h\in H} \Phi_{t,u,h} N_h}^{overlap\ occupancy\ at\ S_t} \leq N_{max}, \quad \forall t \in T. \tag{2.9}$$

In constraint (2.9), the occupancy due to task t at time S_t is denoted by the first term, while the second term represents the aggregated occupancy due to other tasks being run at the same time.

Similarly, the aggregated power at each instant S_t cannot exceed W_{max}:

$$\overbrace{\sum_{h\in H} W_{t,h} X_{t,h}}^{t\ power} + \overbrace{\sum_{\substack{u\in T\\ u\neq t}}\sum_{h\in H} \Phi_{t,u,h} W_{u,h}}^{overlap\ power\ at\ S_t} + W_{idle} \leq W_{max}, \quad \forall t \in T. \tag{2.10}$$

Previous constraints assume no execution order for those tasks that can run concurrently. However, in-order execution can be enforced by setting

$$S_t \geq S_u, \quad \forall (t,u) \in T \mid O_t > O_u. \tag{2.11}$$

Finally, the **Optimization targets** are the following.

Time-to-solution minimization (performance maximization) is defined as:

$$\min(C_{max}), \tag{2.12}$$

while *energy-to-solution* minimization is defined as:

$$\min\left(\sum_{t\in T}\sum_{h\in H} W_{t,h} D_{t,h} X_{t,h} + C_{max} W_{idle}\right). \tag{2.13}$$

2.3.1.2 Implementation

The previous MILP model has been implemented using Gurobi Optimization software v8.0 [63]. Some numerical considerations were taken: an *epsilon* is set to 10^{-6} to enforce strict inequalities in expressions (2.4) and (2.6), an integer feasibility tolerance to 10^{-9}, and fixed $M = |T| \max_{t,h}(D_{t,h})$, being $|T|$ the number of tasks. External data in matrices D and W is set from profiled executions of the task instances to be simulated. Similarly, dependencies represented in matrices P and Q are derived from the read / write requirements of the tasks.

Threading configuration In general, the concept of *threading configuration* encompasses a pair of parameters: (*A*) core occupation –represented as N_h above– and (*B*) number of threads per core. This concept can be particularized depending on the architecture and the application. For instance, in Intel MIC architectures the *scatter* and *compact* parameters, or the number of threads per core –up to 4– can be represented as the *B*-parameter of a threading configuration. In the following, the terms *threading configuration* or *core occupation* are used indistinguishably, as all tested threading configurations in this particular test case consider a single thread per core.

2.3.2 Experimental results

This section exposes the execution simulation results of a blocked Cholesky dense factorization in double precision running on a high-end 20-core Intel®Xeon Gold 6138 @2GHz, with *Thermal Design Power* (TDP) of 125 watts, featuring AVX512 vector extensions.

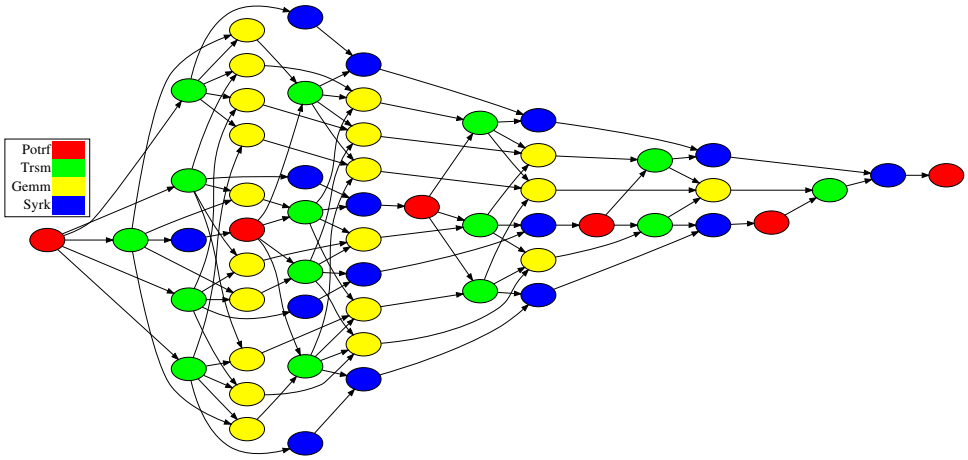


Figure 2.9: DAG of a blocked Cholesky factorization induced from a 6×6 tiling.

2.3.2.1 Optimization cases

Six optimization cases optimizing time and energy are simulated –Expr. (2.12) and (2.13)– with power budgets –Expr. (2.10)–, 70 w, 100 w, and unconstrained ($\approx$ TDP).

The factorization is performed by a blocked algorithm running on 6×6 sub-matrices that evenly divides the target matrix, corresponding to a task dependency graph with 56 tasks (Figure 2.9). As in the previous HeSP simulation, these tasks are instances of four computations or *kernels*, namely `potrf` (Cholesky factorization), `gemm` (general matrix-matrix multiplication), `syrk` (symmetric rank-k update) and `trsm` (triangular solve with multiple right-hand sides). Three problem sizes of the Cholesky factorization are tested and a set of instances for each size are solved. Homogeneous threading schedules are used as a baseline and they correspond to schedules in which there is only one allowed core occupation for each task –using the full `potrf` task as the baseline run for the 20-core case–. In heterogeneous threading schedules, several core occupations are allowed in the same schedule. Due to model limitations in the heterogeneous cases, only four different core occupations (A/B/C/D) are allowed for each instance, in contrast to a single and more general heterogeneous set (e.g. 1/2/3/.../20), in order to limit the search space and resolution time of the linear program. Other combinations could be tested as well but only four heterogeneous sets were chosen for the sake of simplicity. The particular values of the allowed core occupations in both homogeneous and heterogeneous cases were chosen to be divisors of 20 –the full core availability of the CPU– to facilitate the packing of concurrent tasks.

Note that the amount of tasks and the DAG parallelism result from the tiling divisions of the target matrix. Due to time limitations –the solver is exact and its complexity is not polynomially-bounded–, the mentioned subdivision was chosen, from which most of the homogeneous-threading solutions can be solved optimally under two hours. A further study exploring the trade-off between greater task parallelism levels (corresponding to finer matrix subdivisions) and intra-task threading, for a fixed problem size, is out of the scope of this study, as it is impractical from the given exact solving approach. Note that the relation between task granularity and intra-task parallelism is explored nevertheless by maintaining the tiling fixed and varying the problem size.

2.3.2.2 Prior measurements and considerations

In [8], authors proposed and validated a highly accurate model for power and energy estimation regarding blocked-dense Cholesky factorization on a Intel Xeon multiprocessor in which tasks are single-threaded. In this line, the same model was adopted and extended in the present study to consider multithreaded-tasks.

All the information needed to set the delay and power matrices for all the optimization cases

Allowed core	$n = 6144$			$n = 12288$			$n = 24576$		
occupations	Unc.	100w	70w	Unc.	100w	70w	Unc.	100w	70w
Homog.									
1	0.21	0.21	0.20	0.23	0.23	0.22	0.24	0.24	0.22
2	0.43	0.38	0.25	0.45	0.40	0.26	0.47	0.41	0.27
4	0.67	**0.48**	**0.30**	0.71	**0.50**	**0.31**	0.77	**0.53**	**0.32**
5	0.68	0.44	0.22	0.75	0.48	0.24	0.79	0.52	0.26
10	**0.73**	0.38	-	**0.75**	0.39	-	**0.82**	0.42	-
20	0.59	-	-	0.64	-	-	0.78	-	-
Heterog.									
4/5/10/20	0.75	**0.55**	0.28	**0.82**	**0.59**	0.29	**0.88**	**0.62**	0.31
2/5/10/20	0.75	0.54	0.30	0.81	0.56	0.32	0.88	0.60	0.34
2/4/10/20	**0.75**	0.54	0.31	0.81	0.57	0.32	0.86	0.61	0.34
1/4/10/20	0.75	0.54	**0.31**	0.80	0.57	**0.33**	0.85	0.61	**0.34**
Improv. (%)	2.96	14.49	4.33	8.50	17.88	7.07	6.58	15.64	7.67

Table 2.3: Performances in TeraFLOP/sec for each problem instance $n \times n$ and different power constraints –unconstrained (*unc*) and limited to 100 and 70 watt (w)–, considering homogeneous and heterogeneous threading. Best values per problem size and power constraint are highlighted in bold. Some power-constrained scenarios are not compatible with specific core occupations (-).

was gathered by combining the kernel set, size set, threading set and measure set, namely: `potrf`, `gemm`, `syrk`, `trsm`} $\times$ {24576, 12288, 6144, 4096, 2048, 1024} $\times$ {1, 2, 4, 5, 10, 20} $\times$ {avg. run time, avg. run power}, respectively. Kernel executions were run under Intel®MKL 2018 enabling AVX512 vector instructions for highest performance. PMLib [17] framework together with Intel RAPL were employed to gather power measurements.

As mentioned in section 2.3.1, the core-independence assumption requires that when two tasks are simultaneously running on two separate subsets of cores, their performance or consumed power are considered independent. To satisfy this simplification, automatic Intel *Turbo Boost* mode had to be disabled and specific combinations of some run time variables –number of active cores, core frequencies and usage of vector extensions– had to be avoided, so automatic processor-internal power capping policies are not triggered. During this study, it was experimentally found and verified [75] that for this processor, setting the base frequency to 1.7GHz was sufficient to keep all the active cores running AVX512-enabled MKL kernels while keeping their frequency stable. Upon this, time and power measurements for each kernel were taken and a linear relation between power and the number of the active cores was observed, from which previous power-modeling simplifications can be validated. Specifically, power values were extracted from energy measurements of `PACKAGE_ENERGY_PACKAGE0` counter exposed by Intel RAPL. The value W_{idle} was taken by extrapolating these linear relations to the origin –zero active cores– and averaging them for the four kernels, resulting the value of 29.3 watt.

2.3.2.3 Performance maximization

Table 2.3 summarizes the simulated performances, in terms of TFLOP/sec, of the optimal or near-optimal solutions found for each instance when minimization target (see Equation 2.12) is set. The improvements relate the best found heterogeneous-threading performance with respect to the best found homogeneous-threading performance for each problem size. In general, the heterogeneous schedules consistently correspond to better solutions, yielding improvements between 2.96% and 17.88%.

Interestingly, greatest improvements correspond to the 100W-constrained case, reflecting that homogeneous thread schedules are unable to fit in that particular power budget as efficiently as the

heterogeneous ones do. It is also worth noting that the performances for all of the heterogeneous threading cases are very similar for a given problem size, which suggests that the particular allowed core occupations are not too important, as long as there is enough variety of threads the tasks are allowed to take –four possibilities only in these cases–.

Figure 2.10 shows the resulting execution traces from four of the best found instances, corresponding to homogeneous and heterogeneous schedules on two power budget cases. Note that in the DAG depicted in Figure 2.9, the very beginning and the end of the application correspond to stages in which the amount of task parallelism is reduced. In the homogeneous threading traces, this inevitably causes some cores to be idle. Contrary, heterogeneous threading schedules are able to alleviate processor starvation by increasing intra-task parallelism in those areas, minimizing internal delays of serialized tasks.

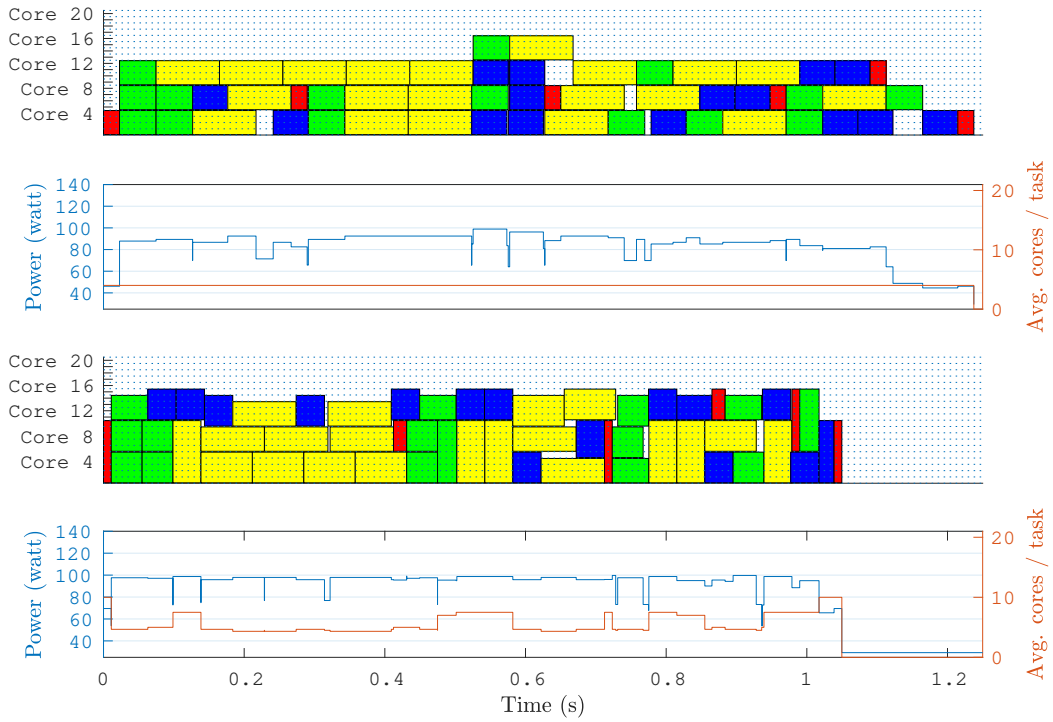
Observe however that deviations from perfect intra-task strong scaling result in a reduction of the performance per core –inversely proportional to the block area, in the shown traces–. For this reason, in an opposite processor-contention scenario –task parallelism greater than available hardware concurrency–, it is wiser to maximize performance per core rather than minimizing internal task delays. Note that these two opposite scenarios –processor starvation caused by task serialization versus processor contention– can occur in the same application, yet can be tackled by the same variable intra-task threading approach, handling opposite policies –increasing and decreasing intra-task threading, respectively–. This idea is reflected in the constrained and unconstrained execution traces of Figure 2.10, in which the rigidity of the homogeneous configurations would not permit an efficient adaptation of the tasks to the evolving characteristics of the application execution.

Focusing on homogeneous threading, it is worth noting that best schedules correspond to a number of threads per task that depends on the underlying constrained scenario –e.g.: homogeneous traces in Figure 2.10 correspond to 4 and 10 threads per task in power-constrained and unconstrained, respectively–. As in current task-parallel runtimes tasks are usually run internally-sequential –i.e., 1 thread executing each task–, this observation makes the idea of *relaxing* the boundary between task- and data-parallelism more appealing, as intermediate degrees of these parallelism paradigms (even fixed across the execution) can be exploited to improve efficiency.

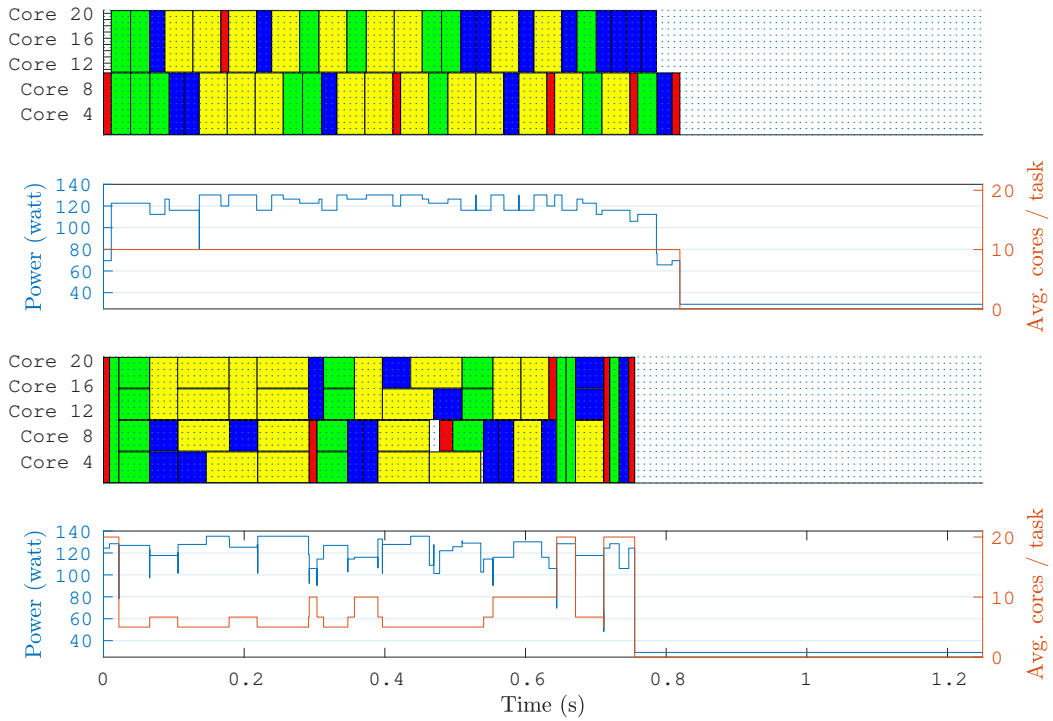
2.3.2.4 Energy minimization

Table 2.4 summarizes the energy consumption of the best-found solutions for each instance considering energy optimization (that is, imposing Expression 2.13). It can be seen that best found heterogeneous threading schedules consistently provide more energy-efficient schedules, with minor deviations among the four different allowed heterogeneous threading configurations, similarly to what was shown in Table 2.3. However, although energy minimization is positively correlated to makespan minimization, the improvements are more modest than the performance case mainly for two reasons.

First, the room for optimization left by processor starvation was tackled by heterogeneous threading by increasing intra-task parallelism, thus reducing internal task delays at the cost of reducing performance per core and per watt –or in other words, increasing the energy cost per computation–. Second, in power-constrained schedules, note that in Figure 2.10 some cores are effectively idle due to power-constrained restriction. However, the static power they consume contributes to the overall energy values exposed in Table 2.4. This affects to the relatively lower improvements seen, as improved heterogeneous threading decisions weight less in terms of energy footprint, compared to the performance-target case. Nevertheless, those idle cores could be effectively disabled in practice, and an extended model exploring threading P/C-states decisions made per core basis, done jointly with variable intra-task threading could provide better energy-efficient schedules.



(a) Homogeneous (top) and heterogeneous (bottom) threading traces (100 watt constraint).



(b) Homogeneous (top) and heterogeneous (bottom) threading traces (unconstrained).

Figure 2.10: Execution traces for best found homogeneous and heterogeneous threading schedules. Tested problem size: 12288×12288.

Allowed core	$n = 6144$, (10^1 j)			$n = 12288$, (10^2 j)			$n = 24576$, (10^3 j)		
occupations	Unc.	100w	70w	Unc.	100w	70w	Unc.	100w	70w
Homog.									
1	2.01	2.01	2.08	1.50	1.50	1.54	1.16	1.16	1.20
2	1.47	1.54	1.83	1.13	1.18	1.41	0.87	0.90	1.09
4	**1.21**	**1.35**	**1.64**	**0.92**	**1.03**	**1.26**	**0.70**	**0.78**	**0.97**
5	1.26	1.44	1.94	0.93	1.06	1.43	0.70	0.80	1.08
10	1.23	1.51	-	0.98	1.20	-	0.72	0.89	-
20	1.62	-	-	1.20	-	-	0.79	-	-
Heterog.									
4/5/10/20	1.22	**1.31**	1.72	**0.91**	**0.99**	1.29	0.69	**0.75**	0.99
2/5/10/20	1.22	1.34	1.70	0.92	1.02	1.26	0.69	0.78	0.96
2/4/10/20	**1.19**	1.32	1.65	0.91	1.00	1.25	0.69	0.76	0.96
1/4/10/20	1.19	1.31	**1.61**	0.91	0.99	**1.24**	**0.68**	0.76	**0.95**
Improv. (%)	1.92	3.00	1.58	1.56	3.89	1.58	2.19	4.14	1.75

Table 2.4: Energy consumption in joule-basis (j) for each problem instance $n \times n$ and different power constraints –unconstrained (*unc*) and limited to 100 and 70 watt (w)–, considering homogeneous and heterogeneous threading. Best values per problem size and power constraint are highlighted in bold. Some power-constrained scenarios are not compatible with specific core occupations (-).

2.4 Summary

2.4.1 Related work

HeSP framework. In relation to the analysis exposed in Section 2.2, focusing on dense linear algebra implementations, [157] propose a hierarchical directed acyclic graph strategy, creating a two-level DAG hierarchy on systems featuring two types of computing platforms (CPU/GPU). Similarly, [64] proposes an offline adaptation of the task grain size to the processor type and to statically assign tasks to distributed compute nodes. On the other hand, [40] proposes an alternative approach in which computing resources are aggregated as needed in order to adapt the computing capabilities to coarse grain kernels. The *Versioning* task scheduler for the OmpSs runtime [122] defines multiple implementations per task, each one targeting a different processor type, and decides at run-time where to map them based on historical runtime information.

HeSP extends the aforementioned efforts by exploring the global impact of *arbitrary* degrees of task granularity on an arbitrary heterogeneous platform, adapting task sizes not only to the individual processor capabilities, but also to the current degree of available parallelism dictated by a specific algorithm.

The HeSP framework and its internal mechanisms toward joint scheduling / partitioning tasks on heterogeneous architectures were presented. Insights reveal that important performance benefits and improved processor loads can be extracted from the framework for a family of scheduling policies. The extracted insights for the Cholesky factorization can be applied to other irregular task-parallel implementations, or to arbitrary heterogeneous architectures.

The *static*, *iterative*, and *global*-scoped characteristics of HeSP algorithms have shown to be useful to explore the practical performance bounds of a scheduling-partitioning problem for specific *application-vs-platform* pairs. The positive findings exposed in this study motivate a *dynamic*, *constructive* and *local*-scoped exploration that would emulate more realistically the effect of partitioning policies done by an actual runtime scheduler, in which local run-time information would be applied on a *per-task basis* to perform efficient task partitioning decisions.

MILP approach. With regard to the analysis done in Section 2.3, the study of techniques to efficiently schedule threads to cores at run-time has been a topic of interest since the rise of multiprocessing architectures, but mainly at the application level. [43], [54], [140] and [100] have developed different thread scheduling approaches considering performance, energy and/or power optimization goals on symmetric multiprocessors, some of them exploring the effect of SMT capabilities for internally-threadable cores. Specifically, [124], [100], [142] and [39] incorporated Dynamic Voltage Frequency Scaling (DVFS) techniques together with threading management decisions, and exposed their joint impact on performance and energy optimization under power constraints. Regarding newer architectures, [101] and [124] explored threading policies in a context of asymmetric / heterogeneous multiprocessors in power-constrained contexts, and [98] studied the effects of different threading configurations on performance and energy for a set of benchmarks running on the Intel MIC architecture. In general, previous works have demonstrated the importance of thread management as a mechanism to assign computations to processors, exposing remarkable results on power-aware performance- and energy-oriented optimizations.

Although similar goals were pursued in the present analysis –performance and energy-efficiency improvement on power-constrained scenarios– it was directly focused on variable thread management at *task* level, representing tasks as inter-dependent blocks of computation that compose the application. In this context, [130] exposed performance and strong scaling measurements of FMM kernels on different multiprocessing architectures, motivating kernel-aware thread management for FMM-based applications. Similarly, in this chapter it was shown by simulation that threading decisions at run time and task level can increase performance and alleviate energy consumption, while eventually satisfying a given power budget.

2.4.2 Limitations of static approaches

All in all, the presented approaches expose some remarkable limitations.

2.4.2.1 Task DAG is static

Previous analysis assumed a fully known task DAG. However, in real scenarios, tasks can be generated on-demand as some tasks are completed and other that were depending on them have their dependencies resolved. Also, regarding dynamic task partitioning, the extra cost of unfolding the computation into a set of inter-dependent tasks was not treated at all in this analysis. This should be considered, specially for finer-grained tasks in which the cost of generation of multiple tasks may affect the practicality of run-time task partitioning policies.

2.4.2.2 Runtime-specific behavior not treated

Mechanisms such as *(i)* task-queue management, that controls the production and consumption rate of tasks, *(ii)* dynamic caching policies, and *(iii)* the effect of parallel synchronization primitives, were not contemplated in previous static models. These mechanisms may greatly influence the execution behavior and even limit the applicability of the proposed approaches.

2.4.2.3 Model dependence

Previous models ultimately require a prior and accurate task and data transfers profiling in order to be able to anticipate the execution and data transfer costs. Even assuming that runtime-specific stochastic effects are sufficiently small (which in most situations is far from true), the cost of *experimenting* is usually higher than in runtime-scenarios: from a *dynamic* standpoint, the effect of a scheduling policy can be straightforwardly evaluated by just executing the application.

2.4.2.4 Model complexity limits the size of problem instances

On top of the NP-hard characteristic of the *classic* task scheduling problem, the presented combinatorial optimization problems expose in practice additional limitations in terms of *how many tasks*, *how many processors*, and *how many possible additional configurations* (number of parametrized partitions for each task and number of *threading configurations*) can be actually treated in these models. It is the non-polynomial growth of the space of solutions what makes these problems not solvable in a *reasonable* time, even for instances with relatively modest sizes.

2.4.3 Application to runtimes

The HeSP approach can be applied directly on actual task schedulers or programming models, in order to implement recursive task partitioning capabilities as an additional degree of freedom. In particular, the concept of *parametrized partitioners* could be implemented on the user interface side, and coupled with a kind of a partitioning engine working in the background.

Concerning the MILP approach, the previous trace analysis provides interesting insights concerning the design and implementation of future task-to-threading heuristics and smart scheduling policies applied from a runtime scope. Note, however, that besides scheduling performance and efficiency considerations, there is a potential benefit of these approaches in terms of programmability: just being able to delegate these *threading decisions* to the runtime system is already a step forward in the path toward more transparent, automated and user-friendly programming models.

In this hypothetical scenario, the programming model can provide a user interface as a mechanism to declaratively expose the available threading configurations that a task running on a specific device is allowed to take at run-time. Whether the underlying runtime scheduler is able to perform this threading decisions in a smart way or not is a problem to be solved afterwards, but the relative simplicity of the previous heterogeneous-threading traces –and their similarity compared to homogeneous cases– suggests that they can be affordable and accessible from runtime-based heuristics. Although not directly applicable from a runtime approach, the exposed MILP model and the availability of powerful exact solvers can provide useful guidance to anticipate potential performance and energy improvement bounds that can be expected from future task-threading schedulers.

The study in this chapter can be applied to other applications and architectures as well. From a general application-based perspective, the allocation efficiency of a set of threadable, heterogeneous and interdependent tasks in a symmetric multiprocessor will be affected –in general– by the task characteristics of the application: globally, the overall task parallelism and the heterogeneity of the tasks; specifically, their individual intra-threading parallelism, granularity and criticality. In these scenarios, task-oriented runtime-aware threading decisions are expected to fill the optimization room left by the complex interplay of these application characteristics.

Diving into these generic characteristics, note that in the tested application, in spite of all the tasks being threadable and interdependent, their heterogeneity in terms of the intrinsic computation is rather lacking –all tasks are $O(n^3)$, compute-bound kernels acting on matrix blocks of the same size–. This fact makes that the heterogeneous threading decisions are mainly based on the existence of available resources rather than on the intrinsic efficiency based on a *computation / granularity-to-threading* relation. From this argumentation, it is expected that the benefits of variable intra-threading approach will be more clear for those applications in which the *task heterogeneity* is higher, so not only load balancing but also *intra-task strong scaling*-aware considerations can be taken for efficient *per-task* threading decisions.

In terms of hardware, architectures that exploit SIMD parallelism managing larger numbers of threads –e.g. Intel Xeon Phi and *SIMT*-based execution model in GPUs– are subject to a similar analysis, as long as they support concurrent task execution and explicit partial processor allocation.

The extracted insights, despite the limitations of the models, motivate the design of a programming model based on these ideas, and a runtime system able to efficiently *navigate* in this complex and enriched decision space.

Don't enjoy the peace
from such abstract heights;
deep below the inverted trees
chaos still thrives.

3

Dynamic relaxed execution. A programming model architecture

In this chapter, the execution relaxations paradigm is developed from a *runtime* or *dynamic* perspective, which permits to address some of the problems related with the static approach exposed in the previous chapter. Those aforementioned ideas revolved around *optimization*-based motivations, which permitted the formulation of a class of combinatorial optimization problems solvable for a set of different optimization targets. On the contrary, issues about how an *expanded-execution*-leveraged runtime can be actually implemented, and how users can benefit from a programming framework in which *execution relaxations* can be expressed, were not addressed at all.

In this sense, the dynamic approach presented in this chapter moves toward a more realistic conception of the expanded execution model, and focuses on *programmability* and *software architecture* issues as central topics, keeping issues about performance optimization as second-tier considerations. Apart from an improved realism, the dynamic model presented in this chapter is equipped with more *execution expansions* than which were treated in the static approach.

This proposal shares the rationale of many of the task-based parallel programming models exposed in Section 1.2.2, with respect to the ultimate goal of delegating issues related with parallel programming from the user to an automatic system. To accomplish this goal, it is proposed a programming framework architecture rooted on declarative-like principles, in which the execution entry point is reduced to a *single-yet-abstract* task (representing the whole application) assigned to an abstract and unique *execution context* or *executor* describing the complete parallel architecture. Apart from this *single-task* assignment, the user is required to expose a set of *execution opportunities* or *paths* to the runtime system, so that those more promising execution paths in terms of performance can be automatically explored and exploited. Moreover, an automatic mechanism to infer the underlying architecture will permit to build an *application-and-system*-tailored *tree*-like structure of executors, and thus complete a model of execution consisting of a set of *tasks* that are automatically distributed at run-time across different execution contexts.

In summary, the goal of this chapter is to present a framework in which user-defined *execution relaxations* yield runtime-driven *execution expansions*, which is built on top of general abstractions that need to be defined in advance (Section 3.1). In particular, a generic Task-Executor taxonomy –which resembles the *instruction / data* Flynn taxonomy [55]– is first presented to provide a reasoning framework about how a generic computation (a complete application or any subset of *tasks*) can be mapped to a generic architecture described as an *executor* or a set of *executors*. This taxonomy will be the base to develop the different types of relationships between *tasks* (abstract units of computation) and *executors* (abstract execution contexts), and will provide a conceptual

basis from which different runtime behaviors and *currently statically-determined* user decisions can be exposed as *potentially automatizable execution expansions* to be resolved at run-time (Section 3.2). These execution expansions are the base to explain how actual execution scheduling is generalized, automatized, and ultimately decoupled from application development.

3.1 Core abstractions

As explained in Section 1.1.1, abstractions are fundamental to encapsulate complexity in different and inter-related layers. The model architecture presented in this chapter is built on top of the same abstraction dualism of the static approach, namely: the *application* (the computational problem to be solved by the user), and the *system* (the machine that will solve it). Similarly, application abstraction is divided into *data* (considered as the collection of all valid input data that can be thought and every possible result to be expected), and *computation* (representing all the possible algorithms and methods that transform the input information into the result). On the other hand, the system abstraction is general enough to represent virtually any processing system, and it is similarly composed by two equally relevant components: the *execution context*, representing any set of processing devices that can cooperate, and the *storage context*, which encompasses any subset of interconnected memory spaces.

Application abstraction. The task-asynchronous and data-flow models are the building blocks for the following developments. As mentioned in Section 1.3, despite these models have succeeded in expressing general computations from simple task dependency definitions, they still have serious limitations. Armed with this motivation, part of the following discussion aims at generalizing tasks by means of *featurizations*, while still maintaining the asynchrony and data-flow characteristics. In the following developments, a *task* refers to a representation of an abstract computation, in the sense that a whole application or a specific subroutine belonging to the application can be considered as tasks. The boundaries of a given task (in terms of its input and output) are assumed to be clearly defined in any case. Regarding data representation, general data structures are considered, being also *decomposable*, in the sense that they can reside in different storage contexts divided into pieces.

System abstraction. Any parallel processing system is represented as a structure of more particular entities: *executors* and *allocators*. Executors are the fundamental recipients for tasks and provide a context in which the actual workload abstracted by tasks can be instantiated in the form of an actual execution. More specifically, an *executor* refers to an abstraction of an execution context, referring to a single or a set of physical processing resources (i.e., an individual core, SMP, GPU, a heterogeneous platform, etc.). Moreover, executors are composable into graph-like structures following specific rules. Allocators are associated to executors and they also follow composition relations directly derived from those of their associated executors. They abstract physical storage contexts (*memory spaces* onwards), providing not only data storage, but also data consistency and coherency guarantees in a transparent way, so that user data can be internally stored, copied and moved consistently across memory spaces as the execution unfolds.

3.1.1 A Task-Executor taxonomy

The *Task-executor taxonomy* shown in Table 3.1 and illustrated in Figure 3.1 provides a framework to reason about how *generic* computations or *workloads* can be mapped to *generic* parallel architectures.

The term *Multiple* assumes a set of heterogeneous and inter-related entities; *multiple tasks* refer to a heterogeneous set of computation units possibly related by data dependencies forming a Directed Acyclic Graph (DAG), while *multiple executors* represent a set of heterogeneous and

	Single Task	*Multiple Task*
Single Executor	STSE	MTSE
Multiple Executor	STME	MTME

Table 3.1: Task-Executor taxonomy.

inter-connected devices that can communicate and synchronize. In addition, multiple tasks (or, similarly, executors) can be contextually collapsed into a single one. The opposite is not true in general, as not all tasks (or executors) can be decomposed into simpler inter-connected parts.

Specifically, a task is a unit of computation that in general reads an input datum, processes it, and writes the result into an output datum. A task can be broadly characterized in terms of properties such as its algorithmic complexity, its data granularity (the amount of data to be processed) or its computational intensity (in terms of the amount of computation required per data unit). Additional considerations such as its required numerical precision or criticality / importance in relation to other tasks could also be considered for characterization.

Similarly, an executor can be characterized by the architectural characteristics of the device(s) it refers to, such as processing power, bandwidth, memory hierarchy, instruction set architecture or its degree of concurrency. Also, a set of executors tied by data interconnects can be represented by a single executor whose parts can communicate and co-operate to resolve a single task. The *caller* will be referred as the actor (either the user or other executor) that demands another executor (the *callee*) to run a task.

In order to improve programmability and portability, the rationale is as follows: the application execution entry point should always be STSE, so that parallel application programming should aim at STSE-like *front-ends* regardless of the underlying system complexity. The remaining combinations are *backend*-like, and they should be ideally hidden from the user and resolved exclusively by an underlying system software or runtime. Specifically, STME, MTSE and MTME classifications involve some form of ambiguity in terms of computation mapping and scheduling that should be delegated to the underlying automatic runtime system. This rationale is not only motivated by programmability and portability reasons, but also based on *practicality* reasons, from the observation that the complexity of the performance-oriented *application-to-system* mapping in HPC is scaling beyond practical human-made tuning.

In the following, some additional details of each class are exposed.

3.1.1.1 Single Task-Single Executor (STSE)

This classification refers to the execution entry point or interface: a single task representing the whole application is mapped onto a single executor representing the whole computing system. How the task is internally distributed into the different subsystems (homogeneous or heterogeneous units) is hidden from the user and potentially resolved automatically at run-time. A STSE context exposes zero ambiguity to the user: at this level, there is not any decision from the user that may impact performance or portability. From a programming perspective, the single executor representing the entry point (or more precisely, the underlying executors encompassed by it) is instantiated in a different way depending on the underlying system for which the source code is compiled.

3.1.1.2 Single Task-Multiple Executor (STME)

This backend scenario arises when a single computation –representing the complete application or a part of it– is free to be mapped to several available execution contexts (e.g., an SMP equipped with multiple GPUs). In general, executors are considered heterogeneous in terms of computational capabilities. Hence, depending on the specific task characteristics (type, granularity, preci-

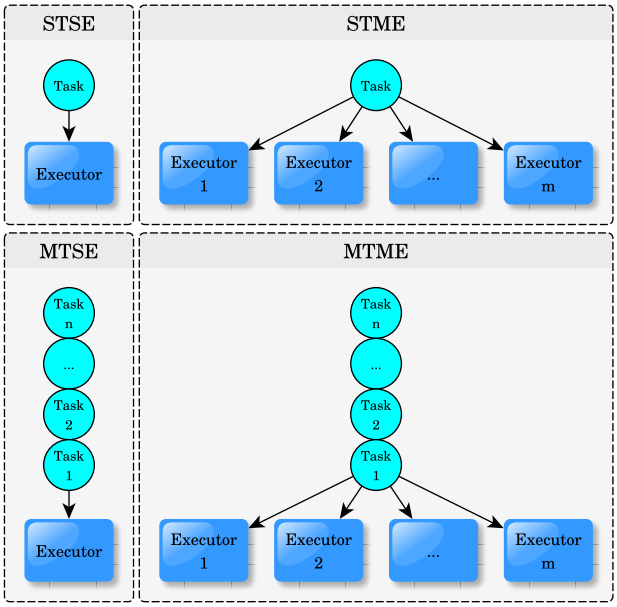


Figure 3.1: Task-executor taxonomy.

sion, arithmetic intensity, etc.), and the run-time *state* of the executors (current load, temperature, clock frequency, etc.), this classification exposes some form of *mapping ambiguity* to be resolved.

3.1.1.3 Multiple Task-Single Executor (MTSE)

In this scenario, several tasks can simultaneously share the resources represented by a single executor, so there is no mapping ambiguity to be resolved by the caller. Nevertheless, this classification exhibits an *ordering ambiguity*, as specific orders in which tasks are received by the executor can have an impact in performance. These situations arise when a set of heterogeneous and inter-dependent tasks motivate considerations about task importance / criticality. Additionally, the presence or absence of data in the memory space(s) local to the executor related to incoming tasks may also motivate a task reordering to minimize idle times or data transfer overheads.

3.1.1.4 Multiple Task-Multiple Executor (MTME)

In this classification, a set of inter-dependent heterogeneous tasks can be mapped to a set of heterogeneous and interconnected executors. Some tasks may be suitable for running in some executors while others may be mapped only to a specific executor. This is a general task-scheduling problem scenario, already explored in the previous chapter, and considered in this case in relation to executors.

3.1.2 Executor typology

The concept of executor refers to a handle to an execution context or a representation of a processing environment, possibly endowed with some scheduling policies and a state, and its purpose is to bridge computation to physical processing resources.

The purpose of executors is to serve as a layer between the requester of computational resources (previously known as the caller) and the physical processing resources. Executors can be composed into hierarchies, and classified into two major classes: *top* and *bottom*. Recalling their role as *recipients* for tasks, executors can be composed or stacked on top of others, so that tasks can flow across several executors. All executors are assumed to behave as *callable asynchronous*

entities (i.e., the caller does not need to wait for the result in order to make progress); this is realized by simply endowing executors with task or *work* queues to which tasks are submitted.

3.1.2.1 Bottom executors

They refer to abstractions of single physical processing devices, for example a single CPU core, a SMP, or an accelerator (GPU, Intel MIC, FPGA, etc.). They are named as *bottom* as they conform the lowest level nodes in an executor tree hierarchy. The way in which sequential processors and multiprocessors are abstracted through bottom executors is explained next.

Abstracting sequential processors. A bottom executor can abstract a sequential CPU core. In this case, several tasks submitted to it will run sequentially respecting a tight submission order. Several core executors representing different CPU cores in a die can be aggregated to form a *pool of uniform resources*. This aggregation can be represented as a higher level executor (not qualifying as a *bottom* kind), which can incorporate policies to manage the lower level executors. In particular, the included bottom executors can share a task queue, own a task queue each, or other combinations (for example, sharing queues and applying work-stealing strategies). The underlying bottom executors run tasks independently, and from the perspective of the caller of the higher-level executor, the submitted tasks are executed out-of-order in general.

Abstracting multiprocessors. A bottom executor can directly abstract a whole SMP processor, so a task submitted to it can be run using several cores in parallel, following a *fork-join* model. In general, distinct subsets of processing units can process different tasks while the processing units within a subset cooperate to execute a task. How the incoming tasks are distributed into its subsets is decided from its internal policies. In essence, these executors run tasks following the *moldable tasking* model considered in the previous chapter. GPU processors can also be managed by a bottom executor and managed under similar considerations as an SMP. In this case, the internal executor policies would forward the task to an underlying runtime system (e.g., CUDA runtime or OpenCL runtime) which would transparently manage the low level components of the GPU.

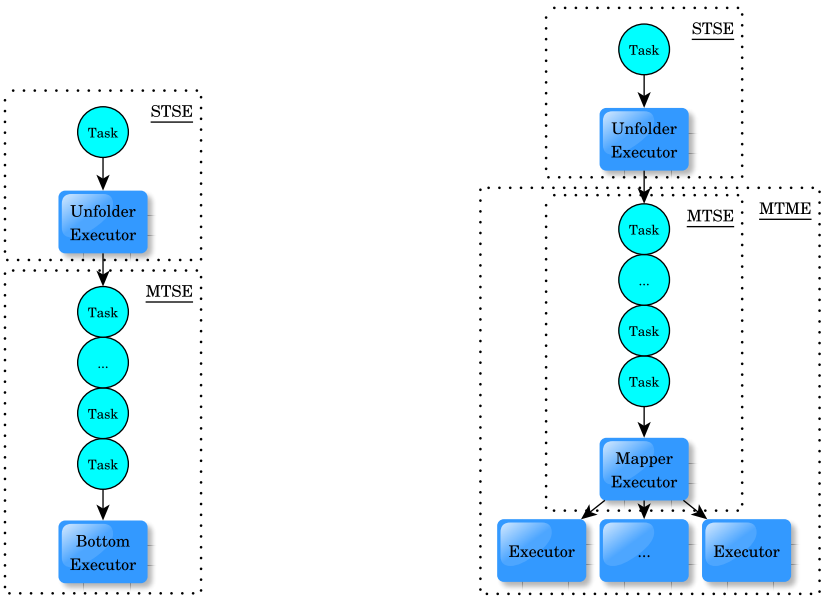
3.1.2.2 Top executors

They refer to executors that work as callers of other executors (either top or bottom). Diagrammatically, they are represented by stacking them on top of other executors. Among top executors, two sub-classes are distinguished depending on the number of callers: *mappers* and *unfolders*.

Mapper executors. This class refers to the executors that are stacked on top of several executors. Mapper executors (or *mappers* for short) offer a *single-executor* layer –without mapping ambiguity– to its caller, so STSE and MTSE scenarios, being the single executor *SE* the mapper, are viewed as STME or MTME from the mapper callees. A mapper executor can incorporate its own mapping policies to forward incoming tasks to the *callees* executors. Specifically, the aforementioned aggregation of bottom executors each managing a CPU core resulted into a mapper executor.

Unfolder executors. This class refers to executors stacked on top of a single callee executor (of any kind) and it provides a context in which tasks can be generated and delegated to that callee executor. In practice, unfolder executors (or *unfolders* for short) provide a context in which tasks can be decomposed into a set, in general heterogeneous, of (sub-)tasks. This set could be a singleton set –i.e., an incoming task is replaced by other task which is then delegated–. A STSE scenario from the unfolder caller perspective is transformed to a MTSE scenario from the unfolder single executor callee, see Figure 3.2. Again, an unfolder executor can incorporate policies, for

example to decide the granularity of the decomposed tasks depending on the runtime state of the callee and the characteristics of the incoming task.



(a) (b)

Figure 3.2: (a) An unfolder executor converts a STSE to a MTSE or (b) to a MTME scenario.

3.1.3 User-defined task featurizations

The *application* encapsulates the computational problem that the user intends to solve. This single problem resolution can be viewed as a single *task*, whose execution processes input data and generates output data. Similarly, a unit of information to be read, processed and / or generated as result of a task, is referred as a *datum*.

As previously said, a task –by itself– cannot be executed, as tasks need execution environments or contexts (represented by executors in this model). In addition, not all applications can be run in any execution context, or equivalently, not all tasks can be associated to any executors. For this to happen, tasks have to be *featurized* or *enriched* in order to become *compatible* with certain execution contexts represented by executors. Specifically, tasks can be featurized by providing *partitions*, *reimplementations* or *kernel instantiations*.

3.1.3.1 Task partition

Task partitioning is a restatement of the developments already introduced in Section 2.2. An application (or task) can be in general *decomposed* or *partitioned* into a set of tasks forming a DAG. As illustrated in Figure 3.3, a task partition could be one possible way to compute a given problem, and a given application can be decomposed in multiple ways according to different levels of data granularity.

3.1.3.2 Task reimplementation

For a given computational problem represented by a task, there can be different algorithmic instantiations to solve it. These instantiations may differentiate in terms of algorithmic complexity, level of concurrency, parallelization possibilities, heap and stack memory footprint, or arithmetic intensity, among other characteristics. These several *ways* to represent a given application or workload

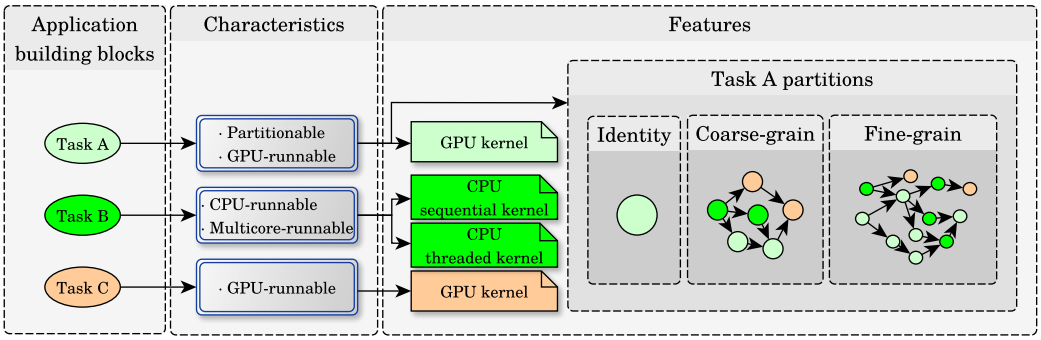


Figure 3.3: Example of featurizations for tasks A, B and C in terms of kernel instantiations and partitions.

are themselves *tasks*, and they are named as possible *task reimplementations* for a given source task that is subject to be *reimplemented* or *replaced*.

3.1.3.3 Kernel instantiation

In general, there are many ways to instantiate the computation (abstracted by a task) in the form of a program or algorithm into source code, and some might be more appropriate for a specific processing architecture than others. These algorithmic instantiations into *computational kernels* (just *kernels* onwards) are usually free functions typically delivered in the form of highly tuned libraries targeting specific processing architectures. Figure 3.3 illustrates different examples in which a task can be featured in terms of kernel instantiations. In general, any children task belonging to a partition or any task representing a reimplementation can be independently featured by its own partitions, reimplementations, and kernel instantiations.

3.1.3.4 Task-executor compatibility

With regard to execution contexts, the STSE execution entry point aims at hiding architectural details to the user by accepting, in general, an abstract definition of a computational problem –i.e., not necessarily a specific kernel instantiation–. Consequently, the question regarding how to formulate a computing problem according to a specific architecture-aware kernel instantiation arises. In this sense, the bottom / top categorization for executors is not only useful to distinguish about proximity of execution contexts to hardware, but it also imposes requirements for *how abstract* the computational units assigned to them can be. Table 3.2 illustrates this statement: bottom executors only allow tasks that expose actual instantiations –in the form of *kernels*– compatible with the physical processing architecture managed by them. For example, a task exposing a GPU-kernel instantiation will be allowed to run on a bottom executor handling a physical GPU. On the contrary, the *logical-and-not-physical* characteristic of top executors only accepts abstract tasks that can either be forwarded to other executors –if it is a *mapper*–, or be transformed in terms of reimplementations or partitions –if it is an *unfolder*–.

Executor type	Compatibility requirements for a task
Bottom	Exposing a kernel instantiation compatible with the target processor
Unfolder	Exposing at least one partition or reimplementation
Mapper	Satisfying at least one requirement of a callee executor

Table 3.2: Task requirements to be satisfied depending on the executor type.

In the following, a number of task-to-executor compatibility restrictions are exposed in detail.

Task *thread forking*. A task that exposes some degree of parallelism can be run by a bottom executor managing a multiprocessor only if the user has declared a compatible kernel instantiation for the multiprocessor. In that case, the bottom executor can decide the amount of resources to devote to that kernel, and it can transparently block those resources in order to prevent simultaneous use of them. Related to parallel architectures, parallelism could be exploited at any level (e.g., whether threads are cooperating in a SIMD, lock-step SIMT, task-parallel, or other fashion is irrelevant in this context).

Architecture-dependent task executions. Similarly, according to section 1.3.3.3, a kernel instantiation of a task could require the use of specific processing units accessible via extensions exposed in the processor ISA (e.g., floating-point units, SIMD vector units, tensor cores, etc.). Hence, a task featured with a kernel instantiation of this kind is only compatible with bottom executors associated to processors that support this kind of characteristic.

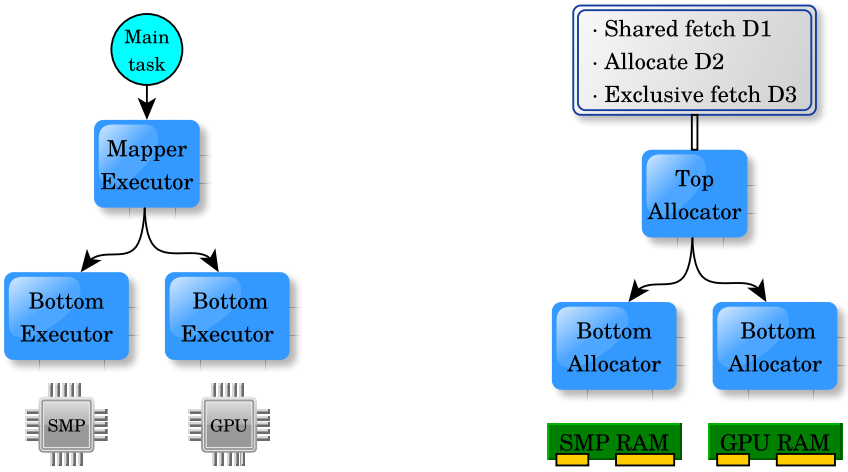
Task unfolding. As mentioned, a task can be assigned to an unfolder executor if it exposes at least one possible way of *unfolding*, either in terms of partitions or reimplementations. To preserve correctness, it is assumed that the processing of the computational DAG after unfolding yields equivalent results –according to some measurement– to the original task. For the sake of generality, that equivalent DAG does not need to be known in advance (i.e., statically), as task dependencies can be resolved as previous tasks are finished and new tasks are generated. Whether the *unfolding* ultimately results in a complex connected DAG, a set of disjoint DAGs, or a single-task DAG –as it is in the case of *reimplementations*–, is irrelevant in this context.

Task mapping. As specified in Table 3.2, a task can be assigned to a mapper executor as long as it satisfies the requirements of any of the executors encompassed by the mapper executor. That is, the compatibility requirement for a task to be run on a *mapper* is essentially *forwarded* or *propagated* to the mapper callees.

3.1.4 Allocator typology

In a similar way in which *executors* and *tasks* reflect abstractions for different *execution-contexts* and *computations*, respectively, this section focuses on how *allocators* associated to executors are designed to abstract a wide variety of *storage contexts*, ranging from single memory spaces to a cache-coherent set of memory spaces. In particular, a *memory space* is any physical device able to store and retrieve data by means of system (OS) or library calls exposed by any third-party program. These memory spaces do not necessarily need to be local to a processing device, but is is assumed to exist an explicit mechanism –again, either by system or library calls– to send and retrieve data from / to them.

The allocator typology is similar *yet* simpler than the exposed executor typology, and it is directly derived from the way in which processing devices are tied to a set of memory spaces in modern architectures. For each executor, a single or a set of memory spaces are allocated, and the data associated to tasks mapped to an executor need to be visible in the memory spaces associated to that executor. In other words, in this model architecture, *task-executor* locality naturally implies *data-to-storage context* locality for the data to be processed by the task. The term *allocator* employed in this model, in general, refers to an active entity able to perform memory management operations such as allocation and deallocation, data caching, eviction and reclamation, and request / satisfy data consistency and coherence operations [135]. Similarly to executors, the typology distinguishes between *bottom* and *top* allocators, that differentiate according to the level of memory management they are designed for, as detailed next.



(a) (b)

Figure 3.4: Executor and allocator hierarchies for a CPU-GPU architecture and associated data management commands triggered if *Main task* reads datum D1, reads and outputs datum D3 and outputs datum D2.

3.1.4.1 Bottom allocators

In particular, a single *bottom allocator* is associated to each *bottom executor*, abstracting away a local memory space referred to the physical device handled by that bottom executor. Bottom allocators are the simplest type of allocators, and they only perform memory allocation and deallocation operations. This *bottom executor / bottom allocator* coupling does not need to be one-to-one, as several bottom executors may refer to a single bottom allocator. The latter situation arises when a set of executors tied to CPU cores are associated to a shared RAM memory space. In theory, local core caches in a multiprocessor could be handled by bottom allocators, but in practice there are no explicit OS calls that permit cache-wise memory allocation, as data coherence in modern multiprocessors is controlled on-chip following a memory model implemented at microarchitectural level [69, 135]. The one-to-one *bottom executor / bottom allocator* coupling becomes otherwise natural in a case in which an accelerator –a GPU or a FPGA, for example– has an associated local memory space directly connected with the physical processor.

3.1.4.2 Top allocators

By definition, top executors *encompass* or *include* lower-level executors. Depending on the characteristics of a top executor, either a *top* or *bottom* allocator will be associated to it. Top allocators abstract several allocators (either top or bottom, or a mixture) and provide a data coherency layer across the managed memory spaces. Figure 3.4 illustrates a simple executor hierarchy, its associated allocator hierarchy, and a set of data management commands to be satisfied by the different allocators, associated to a set of data elements *D1, D2 and D3* of the *Main task*.

Particularizing to *top executors*, an *unfolder executor* stacked on top of other executor will refer to the allocator associated to that executor. Equivalently, a *mapper executor* encompassing a set of executors that ultimately refer to the same memory space will be associated to the *bottom* allocator of that memory space. Contrary, a *mapper executor* encompassing a set of executors referring to more than one memory space will refer to a *top allocator* responsible for data consistency and coherency operations.

Finally, if the system exposes a set of memory spaces untied to any processing device (e.g., permanent storage drives such as hard-drives, flash memories, SSD storage, or non-volatile mem-

ory drives), a bottom allocator is associated to them and any top allocator will implicitly aggregate them to guarantee the visibility of data residing in those memory spaces from any level of the executor tree. Moreover, whether the *Single Executor* at the program entry point belongs to the *top* or *bottom* categories, it will always refer to a top allocator able to resolve coherence requests across all untied allocators together with the allocator(s) referenced by its callee executor(s).

3.1.5 Hierarchical data-dependency resolution

A task is considered to be *ready* when all its *read-only*, *read-write* and *write-only* dependencies are *accessible* within the local memory space(s) associated to the executor to which the task has been assigned. The specific meaning of *accessible* is summarized in Table 3.3 depending on the dependency *kind*. *Data accessibility* requirements are different from *Data locality* requirements in general, in the sense that once accessibility is satisfied, data locality (i.e., data is local to a memory space handled by the current executor) does not necessarily need to occur.

Dependency kind	Conditions for a data object to be accessible
read-only	The last writer has finished
read-write	All readers and the last writer have finished
write-only	Memory allocation is finished

Table 3.3: Necessary events for a task to be ready depending on the dependency kind.

When a task is *ready* in the context of an executor, the actual *execution* can start. Once the resources managed by the executor have completed the task execution, the task is considered *resolved* and other tasks waiting for its completion may become ready.

The existence of unfolder executors and reimplementable / partitionable tasks requires data types featured with partitioning capabilities. A data partitioning is considered as a reversible action (done within the context of an unfolder executor), in terms of the data transformation by itself and the eventual changes in the processing of the partitioned data (again, according to certain measure of equivalence). Partitionable data types must expose *accessing modes* so that data subregions can be accessed by children tasks in which the original parent task is decomposed.

If the task has been assigned to an *unfolder context*, the dependency resolution is performed before the actual partitioning or reimplementation is initiated. Once all tasks generated within an unfolder context have been delegated to the unfolder callee, the partitioned task will not be considered as *finished* unless all generated tasks have finished.

If the task has been assigned to a *mapper*, it is just forwarded to one of the compatible executors handled by the mapper without the need of any dependency resolution. If the mapper encompass a top allocator, then the eventual coherence requests required by dependency resolutions within the contexts of the callee executors are resolved by the same top allocator. Contrary, if the mapper is associated to a bottom allocator, those requests are delegated to the closest top allocator upstream in the executor tree.

3.2 Scheduling execution expansions

This section explains how previous actors establish the scenario in which a runtime-driven *expanded execution model* is defined. Section 3.2.1 introduces the concept of *execution expansions*, that permits (in Section 3.2.2) to generalize the traditional concept of *task scheduling*. Finally, Sections 3.2.3 and 3.2.4 present some considerations regarding the general input and output spaces in which schedulers act, together with diagrams that illustrate how simple hierarchies of executors and task featurizations expand into *execution paths* accessible by the runtime.

3.2.1 Execution expansions

The *execution expansions* are the proposed solution in this thesis to the problem stated in Section 1.3, referring to the lack of automatic capabilities of current runtime systems. As mentioned, the term *scheduling* does not only refer to *task scheduling*, but to a more *general* form in which the abstract *execution* could be orchestrated in richer and multiple ways. In this sense, the term *generalized scheduling* is used in the following to refer to general actions taken at run-time that, in any way, diverts the execution of the program. Hence, the *task scheduling* term or *task-to-processor mapping* is only a *particular* case that considers how tasks are *diverted* to processors by *task-to-processor* actions. In this dissertation, the purpose is to expand current runtime behavior by means of general scheduling decisions that are currently taken by the user guided from his / her intuition and expertise –which comes at the price of higher development costs, ad-hoc and non-portable performance tuning, and possibly suboptimal performance–, that could potentially be done by an automatic agent while attaining new heights in terms of execution performance and efficiency.

As explained, the ability given to the user or programmer to increase the execution possibilities of a parallel program is meant to tackle three problems at once, namely: *(i)* to reduce the *decision cost* during parallel application development by declaring execution opportunities to the runtime system; *(ii)* to let the runtime system access *execution configurations* too complex to be represented in a user-written computer code, yet attaining new levels in terms of performance and efficiency; and *(iii)* to provide a performance-portability development mindset in which an implementation of a parallel application –the implementation in mere computer code– is a higher level abstraction able to be instantiated into many possible binaries depending on the target platform, each of which is able to access many possible opportunities in terms of execution paths.

Execution expansion	*Introduced in Section*	*Done in the context of*
Heterogeneous dispatch	1.3.2.1	mapper executors
Memory management	1.3.2.2	top and bottom allocators
Out-of-order execution	1.3.2.3	any executor
Per-task clocking	1.3.2.4	bottom executors
Moldable task execution	1.3.3.1	bottom executors
Task partitioning	1.3.3.2	unfolder executors
Special vector/matrix processing	1.3.3.3	bottom executors
Mixed precision execution	1.3.3.4	unfolder and bottom executors
Task reimplementation	1.3.3.5	unfolder executors

Table 3.4: A summary of how specific runtime behaviors are expressed in the current model architecture.

Section 1.3.2 exposed how a set of behaviors currently performed without user intervention in a number of runtime systems. Additionally, Section 1.3.3 listed a set of decisions that are usually carried out by users during the application development cycle in HPC. Table 3.4 summarizes these behaviors and user-side static actions –viewed as *execution expansions*–, related with the actors (*executors* and *allocators*) just presented in previous sections. The ability to expose these behaviors and user-handled actions in terms of these presented actors is the first step toward the automatization and full delegation of them to an automatic and generalized scheduler.

3.2.2 Generalized scheduling

The declaration from the user side of the previous execution expansions is meant to let the runtime to fully control the execution. In particular, those execution paths are designed to be promising execution opportunities to be selected by generalized schedulers. For every compatible *task–executor* pair, there is an instance of a generalized scheduler that acts according to a set of available *actions*

or *scheduling decisions*. Specifically, *generalized schedulers* are designed to transparently make decisions that guide the execution to *(i)* optimize some quality measure (e.g., time to solution, energy cost per flop, etc.) and / or *(ii)* to satisfy some constraint (e.g., threshold power).

From the perspective of *ambiguities* at execution time, in the Single Task-Single Executor (STSE) approach, the actor that assigns a task to the top-most executor is the user itself (see Section 3.1.2). In this case, the user is not exposed to any degree of ambiguity, thus there is not any decision to take that could impact neither performance nor portability. That top-most executor could be the tip of a complex and deep executor tree, in which executors are endowed with generalized schedulers that resolve all the ambiguities arisen at run-time, and exposed by the user by means of *featurizations* during the application development process. Armed with this model, and regardless of the underlying complexity, the user role at the development stage is not to *imperatively* drive the execution, but to *declaratively* provide featurizations for the application tasks to enrich the execution, so that the runtime has (and the generalized schedulers in particular have) access to a wide set of execution options.

These *generalized schedulers* are the last necessary entities to complete the proposed model architecture. Summarizing, from the executor hierarchy and the STSE paradigm presented in Section 3.1.1, complex executor tree hierarchies can be built. As mentioned, the frontend context is always STSE and it is connected downstream to a tree executor architecture with virtually unlimited depth. This executor architecture imposes a clear *top-to-bottom* directionality of the computation flow, and this flow in every branch is in essence guided and molded by generalized schedulers in every executor. On the other hand, task dependencies are resolved in general at any level –as every executor but *mappers* do perform dependency resolution–. Lower-level executor layers will typically be associated with finer data granularities –at least able to fit in local memory spaces of bottom executors– while higher-level layers would involve coarse data management referring to several memory spaces, internally managed by *top allocators*. As will be seen in next chapters, specific user declarations may not only influence different execution paths in terms of computation, but certain data operations related to data-management could also be altered from specific user definitions.

3.2.3 Action spaces

In order to reason about how these rich executions can be generated from a STSE context, this section explains how different paths can be branched depending on the typology of executors presented in Section 3.1 (see Figure 3.5). Specifically, in a deep-executor tree context, expanded execution paths are composed at run-time while tasks flow downstream in the executor tree hierarchy, as each executor performs path execution branching according to its typology. Top executors can be identified as those that perform rather *strategic* and *high-level* decisions, while bottom executors will typically perform lower-level decisions in terms of architectural details, possibly based on the precise run-time status of the physical devices. For this reason, top executors (actually, their schedulers) actions constraint the execution paths to be branched by bottom executors, so that the decisions of *top executors* (their schedulers) corresponding to high-level layers in the executor hierarchy are expected to have more impact.

Figures 3.6, 3.7 and 3.8 illustrate the resulting execution paths from specific user-defined execution expansions. The following symbols are used hereafter:

Yellow star: Execution entry point (STSE).

Blue square: Execution branching due to scheduling.

Green circle: Sink of execution path(s), representing the execution on a processing device.

Yellow circle: Fixed decision without any further ambiguity.

Black path: A possible execution path.

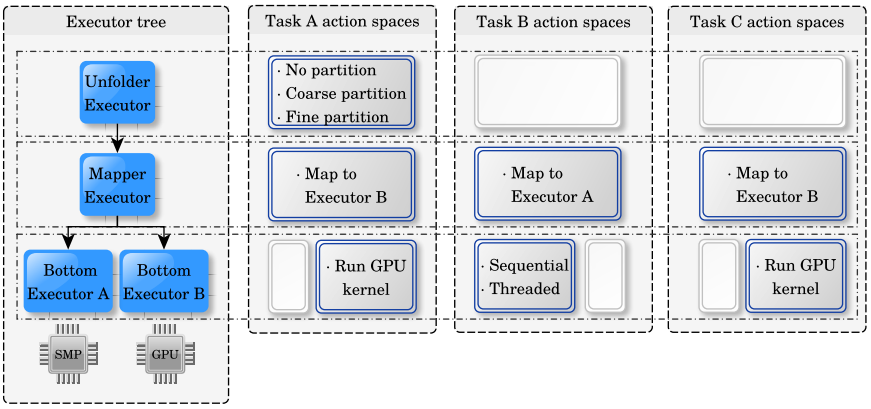


Figure 3.5: Example of an executor tree and action spaces corresponding to featurizations in Figure 3.3. Incompatibility between a task and an executor results into empty action spaces (grey boxes).

Red path: Chosen execution path.

Classical dynamic task scheduling algorithms are just a particular instance of the *generalized scheduling* paradigm presented in this section. Figure 3.6 illustrates the execution paths available for the scheduler to take when a task is able to be mapped to any one of the four underlying devices. Trivially, one execution path will correspond to one task-to-executor assignment. If the task is not able to be mapped to a specific executor –e.g., because it is a bottom executor that manages a processor for which the task does not expose a specific kernel implementation–, the spanned execution paths would be fewer.

Under the context of execution paths and executor architectures, Figures 3.7 and 3.8 illustrate how runtime *generalized scheduling* can be viewed as a generalization of the previous runtime task scheduling problem illustrated in Figure 3.6.

For instance, if a task is mapped to a *bottom* executor managing an SMP, the executor scheduler will choose one *task-to-threading* assignment among the set of the user-declared assignments. Similarly, *unfolder* executors will choose one possible *task unfolding* among the ones declared by the user. Note that *unfolder* executors produce in general MTSE or MTME contexts from the point of view of the executor callee(s). Downstream in the executor hierarchy, the execution of each children task within this multi-task scenario will be subject to subsequent expanded execution depending on the typology of the following executors. In this sense, Figure 3.7 is an example of how higher-level execution paths resulting from reimplementations and partitions branch into lower-level forking paths.

Additionally, the existence of partially ordered (i.e., inter-dependent) parallel tasks sharing resources pose ordering ambiguity that could be subject to a degree of ambiguity equal to the number of possible task execution orders that do not violate task dependencies (i.e., also known as *toposorts*); however this is not illustrated in Figure 3.7 for the sake of clarity.

User-defined execution relaxations can also be composed with *hardware-defined* execution relaxations. Figure 3.8 illustrates this case by composing processor frequency clocking options with user-defined task vectorization possibilities. The same composition can be applied under the scope of a bottom executor managing an SMP.

3.2.4 State spaces

In all practical scenarios, scheduling decisions must be based on some form of input information that is fed into *policies* that define the ultimate behavior of the scheduler. Depending on the in-

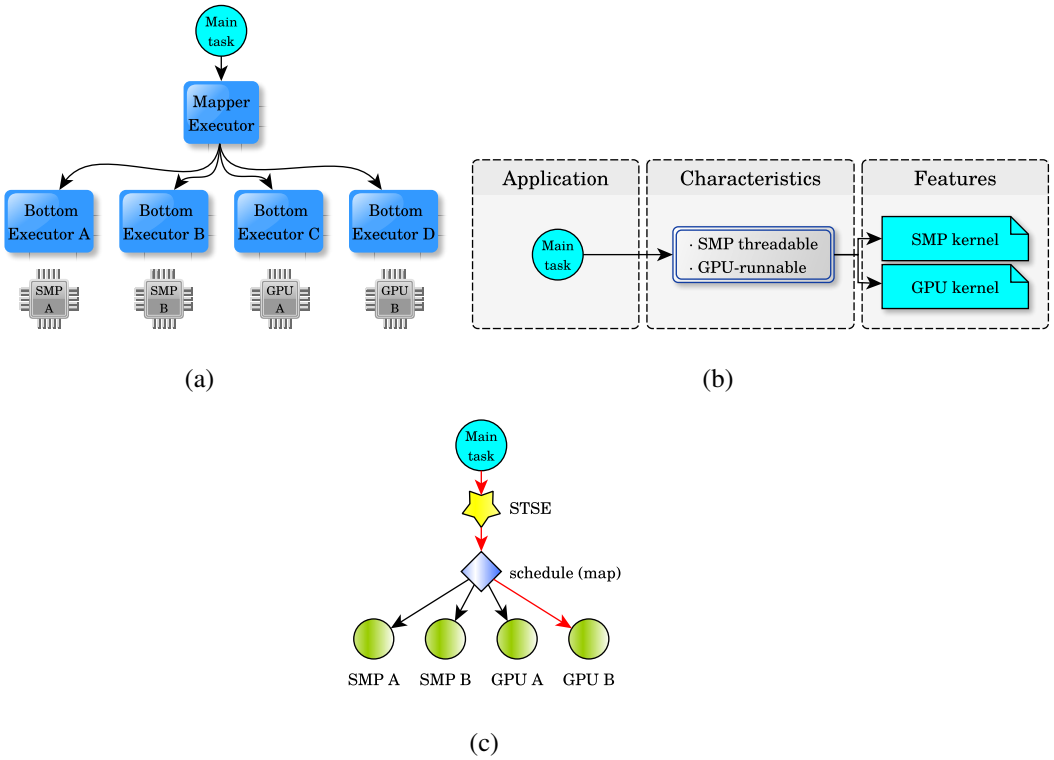


Figure 3.6: Expanded execution paths (c) in a heterogeneous SMP-GPU system (a) resulting when a single task is featured (b) with different kernel implementations for SMP and GPU architectures. The red path illustrates a final decision of the Scheduler that optimizes some metric (e.g., time-to-solution, provided the GPU B is the fastest device for that task).

formation source and policies (and hence, schedulers) can be classified according to the following categories following an increasing level in sophistication:

Blind schedulers. This is the simplest case of scheduling in which the decisions are independent both from the task to be mapped and the properties of the target devices. This could be a fixed or a random decision.

Static schedulers. They are applied based on information not varying at run-time. For example, information regarding the parameters of the task to be scheduled and the type of the target executors can be considered to make the decision. Additionally, performance or efficiency estimations in terms of previous measurements of the task running on the target executors, in the form of a table or in the form of analytic models built from interpolation, would also be encompassed by this category. These policies are usually cheap and greedy: in complex and / or previously unseen scheduling scenarios they are expected to fail, as the static information will be usually too narrow to encompass all possible run-time scenarios. Common scheduling algorithms like HEFT [145] belong to this category.

Dynamic stateless schedulers. They are endowed with policies that are applied based on some form of run-time state of the target devices. They output scheduling decisions based on run-time information of the target devices. They can also be equipped with some form of pre-defined (static) information, but they are also referentially transparent, in the sense that either run-time and possible static information are the only sources considered in the decision making. For example, processor clock scaling decisions implemented in modern processing chips elevate or reduce the processor clock frequency based on the internal temperature state of the chip.

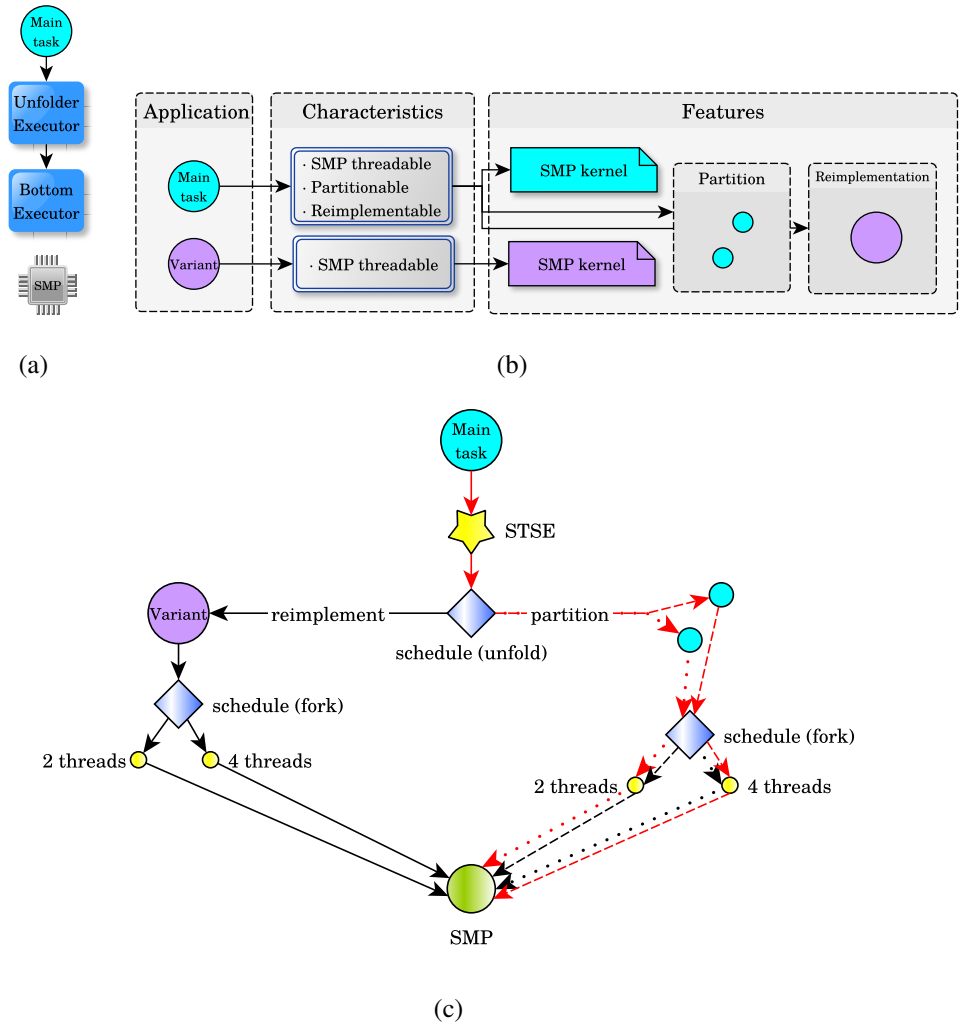
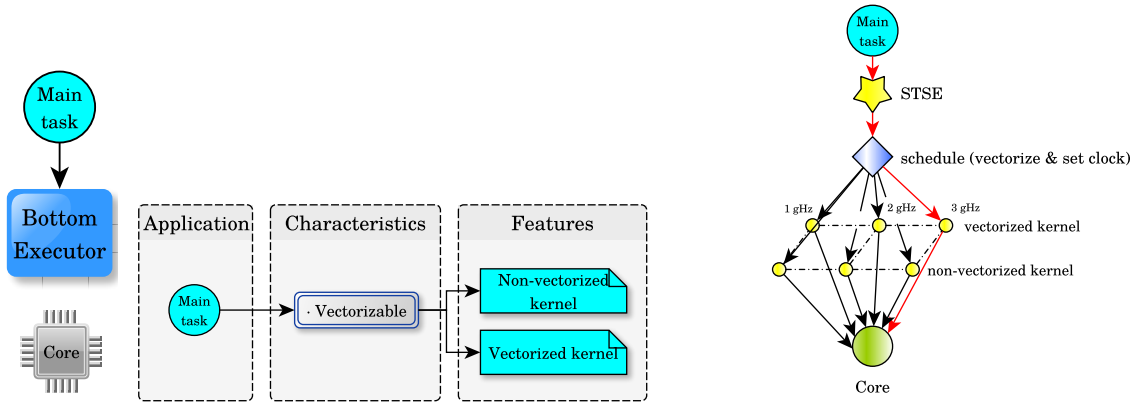


Figure 3.7: Executor architecture resulted from stacking an unfolder executor on top of a bottom executor managing an SMP *(a)*, together with task featurizations *(b)*, result into expanded execution paths *(c)*. It is assumed for simplicity that all *fork* kernel instantiations allow only for two possibilities –either fork using two or four threads–. Paths in red refer to a particular execution in which the main task is partitioned, and each resulting child is executed with two and four threads respectively.

Dynamic stateful schedulers. They are an extension of the previous dynamic schedulers, in which a varying internal state is considered on top of other sources of information –either run-time or static–. This internal state can change and is supposed to enrich the run-time information of the devices with some form of higher-level and abstract representation of the environment, so that this additional insight can yield smarter actions. This internal state is itself supposed to evolve across the execution –this change resembling a training or learning process–, so the application of a dynamic stateful policy may yield different results in different times for the same run-time state and static information. These policies are the most sophisticated ones, and require the design of a complex iterative training process aiming to make the internal state to a scheduler representation able to yield high-quality scheduling actions. For example, machine learning algorithms based on reinforcement learning [83, 154] belong to this category.



(a) (b) (c)

Figure 3.8: Resulting expanded execution paths *(c)* when a bottom executor manages a physical core *(a)* that exposes an ISA with vector extensions and frequency clocking manipulation, and the user provides custom a vectorized kernel instantiation for the task *(b)*. The resulting execution paths are the cross-product of the available user-defined vectorization options and hardware-available frequency clocking options (1GHz, 2GHz, 3GHz). The path in red expresses the joint decision of running the vectorized kernel at 3GHz.

3.2.4.1 Executor run-time states

Noting the importance of dynamic scheduling in modern systems and platforms, in the following it is exposed how the visibility of an executor state is translated to schedulers belonging to any kind of executor, and in general how the exposed executor tree architecture naturally imposes a hierarchical run-time state representation for all levels of any executor tree.

Bottom Executors. As bottom executors manage physical processing devices, their dynamic schedulers must have access to those hardware metrics that can better characterize the state of the processing device at a given time. In particular, metrics such as overall utilization, current clock frequency, cache utilization, or temperature could be considered. It is expected that this information is combined with static information, in terms of characteristics of the task to be scheduled, and dynamic information regarding the executor *task contention* (i.e., the number of tasks assigned to it pending to be executed).

Top Executors. Schedulers of top executors, belonging either to *unfolders* and *mappers*, refer to run-time states that are a mere aggregation of the run time states of the physical devices that ultimately lie at the bottom of the executor tree. Noting this, the run-time state information of executors has a clear *bottom-to-top* directionality, in opposition to the computation flow. In the context of an arbitrary executor tree, and regarding the responsibility of schedulers belonging to high-level executors, this run-time state composition naturally implies a wide view of the state of the system, necessary to perform those high-level decisions with a considerable potential impact.

3.2.4.2 Data-dependent scheduling

In certain applications, the performance of a task may strongly depend on the actual data to be processed. In particular, the motivation of the *reimplementation* execution extension is rooted on this observation. For example, in practical scenarios, linear algebra computations processing matrices that exhibit some degree of sparsity are more efficiently when executed as sparse-based algorithmic variants. Similarly, many computational physics applications based on adaptive decomposition of the simulation domain are another example in which the execution path is strongly

guided by data properties. Hence, based on the importance of this kind of applications in HPC, it seems reasonable to equip the proposed *generalized schedulers* with the capacity of being sensible to data-dependent information. As explained in the following chapters, apart from execution expansion definitions, the user will also be allowed to expose specific function declarations that can be composed with the previously mentioned run-time and static information, so that the generalized scheduling can be additionally guided from data-dependent information.

3.3 Summary

The goals of this section were to establish the basic building blocks from which a programming model can be implemented. The task-executor taxonomy served as a starting point from which rich executor trees abstracting parallel computing architectures can be defined by composition.

Taking into account what this model architecture exposes, the role of the HPC application programmer do not consists of a possibly cumbersome and non-portable parallel programming application development, in which decisions concerning low level parallel programming directives are tightly coupled to considerations of the problem kind / size / heterogeneous system / optimization goal. Contrary, in this model, the expertise of the user translates into decisions about which featurizations are meaningful for some optimization goals, application characteristics, and architecture characteristics of the computing platform.

3.3.1 Related work

3.3.1.1 C++ executors

The concept of *executor* as an abstraction of an execution context is not novel. With regard to C++ language, there are several proposals [73, 95, 94, 72] aiming to incorporate executors to the C++ standard targeting a generalization of the context in which a general computation is mapped. In particular, executors provide a unified layer in which a set of behaviors can be expressed and exploited. For example, executors could provide *(i)* placement semantics to control where and how the execution is performed; *(ii)* control relations between threads –e.g., whether the call to the executor is synchronous, asynchronous or deferred–; and *(iii)* semantics for chaining context-dependent computations. These proposals are meant to be generic, composable with parallel STL algorithms [77] and serve as cornerstones from which more complex and application-dependent computations can be built upon. In the long term, executor proposals are the first step toward enabling heterogeneous computations and networking capabilities in future C++ language standards.

The executors developed in this thesis share the fundamental point of C++ executors proposals, in the sense that all executors are abstractions of execution contexts; however they are fundamentally different for several reasons. Note that C++ executor proposals are meant to enhance the expressivity and capabilities of a general-purpose multi-paradigm language, while the proposed model is an HPC-oriented framework in which user-defined execution relaxations can be expressed to ultimately leverage generalized schedulers to drive fully automatized executions.

Looking at some of the differences, *composability* can be thought as the key characteristic of the presented executors, as they are endowed with tight composition rules that specify how the different executor categories compose with each other. Other key element of the presented executors is that they are strongly parametrized, so that they can be instantiated as *application-and-system*-specialized contexts by means of user-defined execution relaxations and architectural features. Also, these executors are meant to be deployed at the beginning of the program, and compose into an executor tree built from the constraints that the system (ultimately, the hardware) and the user application impose.

3.3.1.2 C++ allocators

The concepts of *executors* and *execution contexts* have a clear correspondence with the concepts of *allocators* and *allocation* or *storage contexts*. Contrary to C++ executors, C++ allocators have been a part of the standard for several decades, and are implemented as a fundamental component of STL containers. C++ allocators have been recently upgraded in the latest C++17 standard to ease their use and extend its capabilities for runtime polymorphism, but their core functionalities –i.e., providing an interface for custom memory allocation– are essentially the same.

On the contrary, although the allocators presented in this chapter also provide contextual memory allocation, their mechanism are fundamentally different. In particular, the presented allocators are always paired with executors –so that memory space locality is derived from executor locality– and always associated to one or several physical memory spaces. Additionally, top allocators are essentially software memory controllers, as they provide context-aware and fully transparent allocation and movement capabilities together with consistency and coherence guarantees. In general, these allocators ensure that data associated to a task running in an executor context can be accessed after task resolution.

3.3.1.3 Model of computation and execution

Among the programming models exposed in Section 1.2.2, many of them [58, 51, 12, 88, 85] provide some level of support for task asynchronous and data-flow computations. This is also the main model of computation and execution in the proposed architecture with an additional subtlety: the task concept that is presented in this chapter refers to a more abstract entity than what is considered a task in current programming models. The presented abstract tasks need to be instantiated *(i)* by user-defined featurizations and *(ii)* by association to an executor context. It is at that point when a task can be actually executed.

Also, as specified in Section 1.2.2, several of the exposed models are designed to support heterogeneous execution –e.g., simultaneous use of a CPU and a GPU(s)–. This is also a possible model of execution in the presented architecture, enabled by the use of a *mapper* executor encompassing a set of *bottom* executors tied to different processing devices.

3.3.1.4 Managing granularity

In this context, the concept of *granularity* is interchangeably used to refer to either *task / data* granularity –i.e., how much of computation and/or data needs to be processed–, and *thread* granularity –i.e. the amount of processing load in terms of number of threads–. Usually, both are strongly related, as coarse-grain data-parallel tasks will perform better when several threads are dedicated to its computation. This problem was illustrated in detail in Chapter 2.

Although several programming models provide explicit mechanisms to choose the granularity associated to a certain computation (e.g., let the user pick (a) the amount of partitions in which a `for` loop is divided for parallelization, or (b) a proper size for data blocks), no model lets the user to relax and to expose *granularity* decisions in order to delegate them to the runtime.

There are however some approaches aiming at collapsing too fine-grained tasks into coarser tasks for efficiency purposes. For example, approaches based on *kernel fusion* [152] or kernel batch execution (employed in some application-specific kernel libraries [113]) have been proposed to decrease the runtime overhead due to excessively fine-grained tasks.

Automatic control of thread granularity in SMPs is also performed in some well-known runtime systems of kernel libraries [76], in which the amount of threads can be decided at run-time based on the kernel parameters. In this case, assumed that this thread management is properly done, the features exposed by the model presented in this chapter in terms of *forking* within a bottom executor, would not be needed.

In summary, despite the performance impact of these decisions is strongly dependent on the granularity of the tasks, there is still no mechamism in current programming models to let the user

relax and expose a set of available threading granularities from which the runtime can pick.

3.3.2 Contributions

The contributions of the proposed model architecture are the result of following the trend of *moving the complexity* (and *responsibility*) from the user to the runtime to simultaneously target problems of programmability, portability and performance. In a more specific terms, several key contributions are exposed in the following.

Systematic complexity parametrization and delegation. Regarding the presented model architecture, task granularity is meant to be runtime-driven by means of generalized schedulers associated to unfolder executors. Similarly, thread granularity is also runtime-driven by generalized schedulers associated to bottom executors, from the set of options exposed by the user. More generally, there is no programming model in which all the exposed *execution expansions* can be declared for runtime delegation. In other words, there are no mechanisms in current parallel programming frameworks to systematically parametrize *and* delegate the application- and system-dependent complexities occurring in typical HPC executions, in which most of the decisions are done statically and incrementally by the user during the application development and optimization processes.

The proposed architecture do not only offers the user to expose and delegate to the runtime any decision that is currently performed statically, but also to *compose* several decision sets.

A computation-execution classification framework. The proposed task-executor taxonomy is meant to provide a framework for composing generic scenarios in terms of computation and execution. From this idea, the motivation for STSE-like frontends was developed, and the executor classes and their composition rules could be proposed.

Encapsulation of complexity. The complexities of the hardware platform are encapsulated by executor abstractions, classified and related by hierarchical structures. Executor composition rules enable the achievement of as many layers of abstraction as demanded, so a reduced number of higher-level execution contexts can be built from a greater number of lower-level executors, until the point in which the whole parallel platform can be reduced to a single abstract execution entity which represents the execution entry point.

Generalized schedulers of executors in the same layer would typically provide similar operations with a similar level of abstraction, while generalized schedulers in different layers will relate to more strategic or more architecture-level decisions. For efficient dynamic generalized scheduling, proper platform and run-time state visibility are naturally propagated *bottom-to-top* during executor composition.

Motivated by the potential benefits of the conceptual framework just presented, the next chapter exposes the design principles and programming interface of a runtime system that implements these ideas.

He caressed a holy crate
but his fingers altered its state.
"You touch, you pay, you take away"
they said.

He offered his soul in exchange.
Of its access, he was not afraid,
as for stateless ghosts in disarray
there is no mutex to take.

4

STEEL. Design principles and implementation

The STSE model and the concept of user-defined execution relaxations have been implemented into the *Single Task / Expanded-Execution-Leveraged Programming Model (STEEL-PM)* and runtime system implementation STEEL-RT, programmable via an Application Programming Interface (API). The goal of this chapter is to present the STEEL-API architecture for application developers and to compare this proposal with other approaches in the literature with similar objectives.

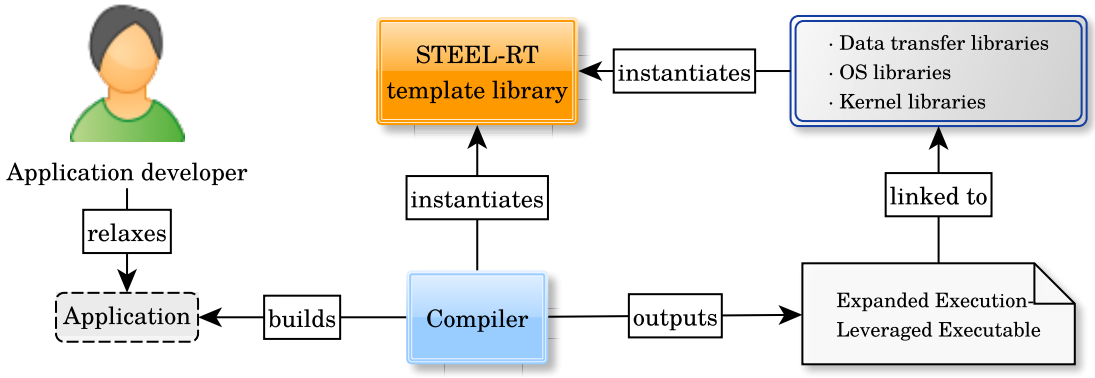


Figure 4.1: Diagram of STEEL programming model implementation.

Figure 4.1 illustrates the basic components of the STEEL programming model implementation. The fundamental principle is that the user (or, similarly, the *application developer*) exposes execution opportunities that might be worth being considered to fully exploit the underlying hardware resources. These opportunities are exposed by means of the aforementioned *relaxations* (or *task featurizations*). Consequently, the compiler outputs an *expanded execution-leveraged* executable, in which all those relaxations are explored and exploited by the underlying runtime system.

As a side etymological note for the sake of disambiguation, the term *expanded execution* is *not* referred here as if the execution is extended *in time*. On the contrary, it is the runtime execution *action space* –or the set of *execution opportunities*– that is *expanded* or *broaden*.

The chapter is structured as follows. Section 4.1 introduces some principles that were followed during the implementation. Some knowledge of basic syntax of C and C++ is welcomed, and the reader is encouraged to review some basic C++ syntax exposed in Appendix A before studying

the contents of Section 4.2, that illustrates with examples the developed API. Finally, Section 4.3 summarizes the fundamental contributions and relates them with state-of-the-art programming models and runtimes, in terms of their expressivity, programmability, and runtime capabilities.

4.1 Introduction

This section illustrates the guiding principles that were taken into account in its design (Section 4.1.1). In addition, the scope in which this programming model is intended to operate is summarized, in terms of both systems –architectures– and the kind of target computations –applications–, also highlighting the scope of the current implementation (Section 4.1.2). Finally, an overview of the implementation architecture is provided in Section 4.1.3, together with a set of basic guidelines related to the implementation language in which STEEL-PM is written (Appendix A), in order to properly understand some syntactic details exposed throughout the chapter.

4.1.1 Implementation requirements

This section is introduced in first place in order to understand the basic architecture and design principles that guided the development of STEEL-API and STEEL-RT. Note that these principles are not specific to the model architecture presented in Chapter 3. On the contrary, they are general principles that, from the opinion of the author, constitute crucial guidelines concerning the development of any programming model implementation designed for HPC application developers.

STEEL-RT can be considered as a *middleware* runtime library that bridges a parallel architecture with the user application. As such, its design principles are rooted on run-time *performance*, *composability* and *orthogonality*, high-level abstractions and hard-to-misuse user interfaces, as exposed next.

Run-time performance

Performance is the most crucial aspect of a runtime library meant to be used for High-Performance Parallel Computing, as the ultimate goal of an application developer targeting parallel application is to extract as much performance as possible from the parallel hardware. In terms of task-based concurrent runtime systems, performance can be reduced by several factors. The typical sources of performance dampening are resource-contention and resource-starvation – i.e., too much or too low utilization of resources–. These problems can be themselves caused by others, like lack of data locality, lack of parallelism, too fine-grained workloads, poor scheduling, or too expensive parallel synchronization operations, among others.

Composability and orthogonality

Composability (in terms of ease of adding new functionalities or *scaling-up* software) and orthogonality (referring to encapsulation and separation of concerns) are crucial aspects for software scalability and robustness. The STEEL-PM implementation is required to be highly scalable in terms of several factors, namely: applications (what the user can express), architectures (from low-level device architectures to heterogeneous and distributed systems), relaxed abstractions (either application-dependent or exposed by hardware), and external kernel libraries (to reuse and exploit highly tuned kernel implementations).

Related to run-time performance, an additional requirement is that new additional functionalities do not introduce unnecessary run-time overhead. This requirement is sometimes paraphrased as *"don't pay for what you don't use"*. With regard of concurrency and its intrinsic programming challenges, it is expected that adding new features do not potentially harm neither program correctness nor parallel performance.

As mentioned in Sections 1.2.1.2 and 1.2.1.4, although object-oriented programming patterns are established as a good solution over previous procedural patterns in terms of encapsulation and separation of concerns, they expose inherent problems in terms of concurrency.

On the contrary, as explained in Section 1.2.1.1, the ever-increasing presence of functional patterns in modern programming languages is partly motivated by the inherent composability and orthogonality properties of pure functions and the lack of global state.

Generic programming patterns have also proven to be an excellent approach when aiming at more composable software designs (see Section 1.2.1.3). Generic algorithms are *feature-parametrized* by design, which usually implies that adding a new feature results to minimal code modifications.

High-level yet expressive abstractions

A software program can be regarded as an abstraction layer that bridges a user intent with some form of computation performed in complex hardware devices; higher-level abstractions naturally imply easier interfaces. However, high-level abstractions should not be *too* abstract so that the programmer's intent is obscured. Hence the goal is to find a proper balance between both, also not sacrificing *control*, thus allowing users to express low-level computations when the situation demands it.

Hard-to-misuse interfaces

Similarly, the main goal of an API is to provide a service while abstracting the programmer from the complexities of the service. The number of ways in which an interface can be misused directly harms programmer productivity and makes the programming learning process unnecessarily harder. Moreover, the importance and complex details of concurrency and parallel computing development in modern software has motivated the rise of programming languages for which programs are not only easy to write, but also more likely to be correct after the first successful compilation. In the context of concurrent and parallel programs this objective acquires extreme importance, as the *usually* non-deterministic nature of the concurrent execution may make the debugging process of run-time errors cumbersome.

In particular, *type-safety* is a built-in feature of *strongly-typed* languages, in which the programming syntax is heavily constrained toward respecting specific relations between data types and discouraging implicit conversions between data of different types. Consequently, strongly-typed languages and APIs drastically restrict their chances of being misused, reducing the possibilities of generating incorrect programs.

In the cases in which run-time errors are expected and / or unavoidable, an efficient mechanism for error handling must also be present. A concurrent or parallel program, to be considered *robust*, must be able to safely retrieve as much information related to the error as possible. However, in a parallel context, this is harder to be satisfied than in a sequential program, as invariants that guarantee concurrent correctness can be easily broken, yielding more errors and potentially causing the program to crash without offering any useful information about the primary error.

Modern C++

Based on the design principles just presented, and the language characteristics exposed in Section 1.2.2.1, STEEL-RT and its API are built on top of ISO C++ 2017 [78]. Specifically, its multiparadigm characteristics in terms of functional and generic programming, together with its low-level concurrency constructs, make it an adequate choice for implementing the STEEL-PM in the form of a high-performing runtime system underneath a high-level abstract interface [138]. Other *not-modern* idiomatic characteristics of C++ such as *RAII* idiom (see Appendix A) will also play an important role for transparent *garbage-collector-free* memory resource management.

4.1.2 Current scope

As exposed in Section 1.1.1, the proposed programming model is designed with the goal of abstracting away certain details of physical processing and storage devices –through *executor* and *allocator* abstractions– as well as computation and data –through abstract tasks and hierarchical / partitionable data structures–.

It is worth differentiating the kind of applications and platforms targeted by STEEL-PM. As the implementation (API and runtime) is fully written in Standard C++, there are no limitations in terms of the programs that can be written in this model. However, the STEEL-PM execution model (Section 3.1.3) consider *tasks* as the main units of computation, so the target applications are those that are decomposable into tasks.

STEEL-PM partially assumes already high-performing computational kernels in the form of specialized libraries or manually highly-tuned implementations. Hence, the target problem is restricted to programming and performance scalability in contexts in which multiple interdependent tasks share parallel computing resources. In this scenario, unless otherwise stated, STEEL-RT is designed to dispatch computational kernels with state-of-the-art performance.

Table 4.1 divides computing architectures into four different levels according to their scale. In relation to the previous chapter, STEEL-PM is intended to provide abstractions for managing the last three levels, while the smallest level is handled within the kernel context. With this regard, the efficient use of the lowest-level processing elements (Arithmetic-Logic units, Control units, Floating-Point units, ...) depends on how the compute kernels are implemented and compiled, so this is not a concern from STEEL-RT. Contrary, the efficient use of the rest of the levels is direct STEEL-RT responsibility. The last level is marked with an asterisk (*) because, although distributed systems are meant to be targeted by top-class executors, the current implementation does not support them yet.

Similarly, Table 4.2 distinguishes four levels of data storage abstractions, with close one-to-one correspondence with those of Table 4.1. In the same line, efficient management of *on-chip* elements of data storage –e.g., registers, caches, memory controller unit– relies on how efficient the memory management is done in the underlying kernel implementations and compilations. Equivalently, as exposed in Section 3.1.4, bottom allocators are designed to provide data management functionalities to local DRAM spaces associated to processing devices (e.g., CPUs, GPUs), while top allocators provide an additional layer for data coherence and consistency across different memory spaces. Identically to Table 4.1, distributed computing architectures are not supported in the current implementation.

Scale	*Processing elements*	*Efficiency managed within*
Smallest	ALUs, CUs, FPUs...	Compiled kernels / external libraries
Small	CPU, GPU, FPGA...	Bottom executors
Middle	Heterogeneous systems	Top executors
Large	Distributed computing clusters	Top executors*

Table 4.1: Four scales regarding processing systems.

Scale	*Storage elements*	*Efficiency managed within*
Smallest	Registers, Caches, ...	Compiled kernels / external libraries
Small	Local RAM	Bottom allocators
Middle	*Intra-node* memory spaces	Top allocators
Large	Distributed memory spaces	Top allocators*

Table 4.2: Four scales regarding data storage systems.

4.1.3 STEEL implementation architecture

Figure 4.2 depicts a global overview of the programming model architecture, together with some of the actors and software building blocks involved. The purpose of this model is to implement the ideas exposed in Chapter 3 under the requirements presented in Section 4.1.1.

The STEEL-RT template library represents the central element of this programming model. First, it consists of a set of generic functionalities parametrized by *support* templates, which have to be instantiated by low-level functionalities in the process of increasing its portability –i.e., augment the number of platforms and backend libraries on which applications can run. Secondly, the core library is also parametrized by application-related functionalities –templates referring to tasks and data structures, which are instantiated by declarative-like user definitions that follow the STEEL-API rules. From the user (application developer) side, three basic events can be distinguished, namely: *installation*, *compilation* and *execution*.

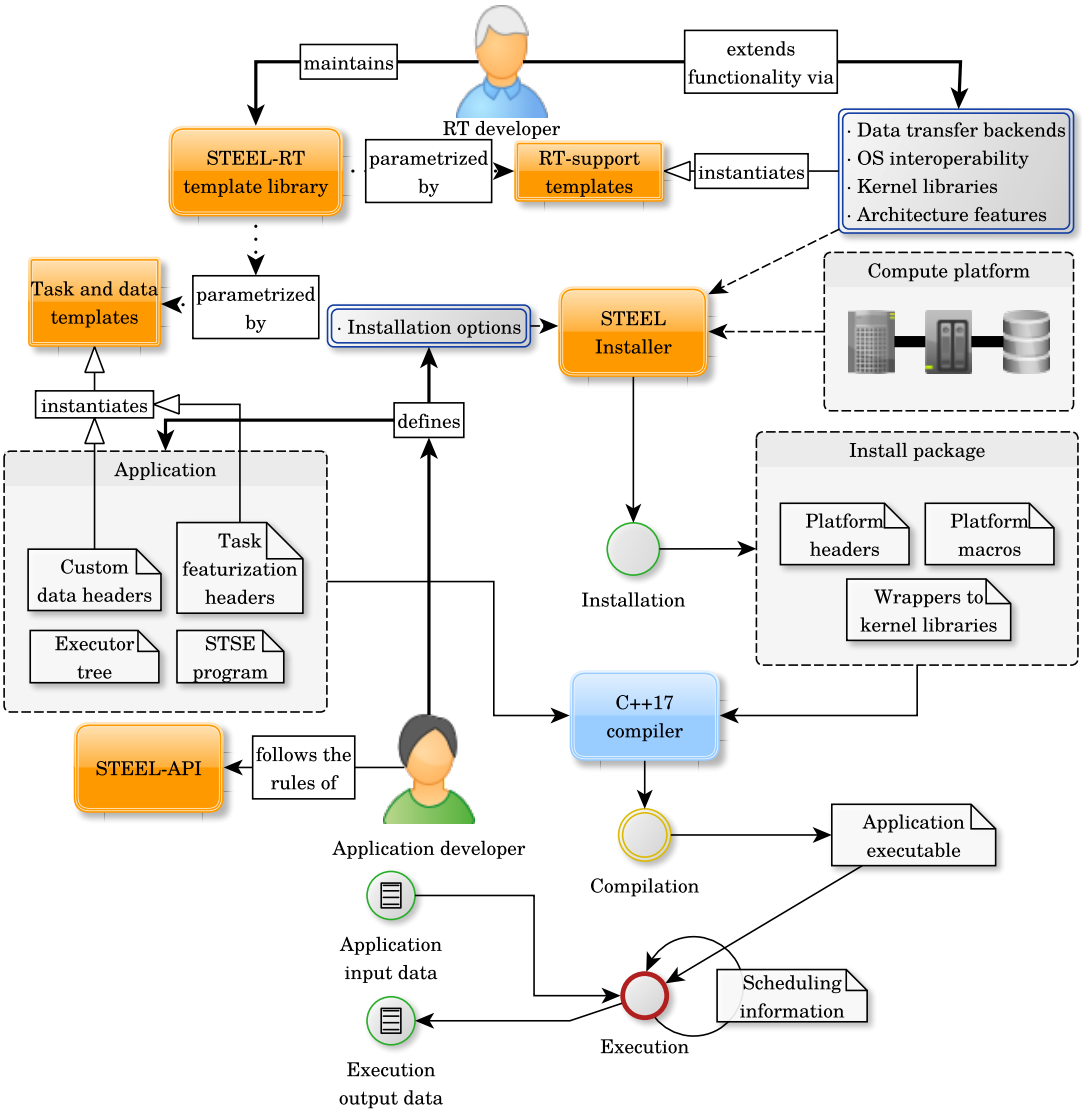


Figure 4.2: STEEL Programming model implementation architecture.

4.1.3.1 Install-time

The installation process is performed by the STEEL installer, which checks that the target compute platform and the installation options defined by the user (application developer) are compatible with the set of supported functionalities. If that is the case, an installation package is created, which encompass platform information (i.e., processing devices, topology, enabled kernel libraries) that fully instantiates the STEEL-RT support templates. Along with platform-dependent headers and macros, library binaries that wrap calls to third party kernel libraries may be built in this step.

4.1.3.2 Compile-time

At this step, user-defined application definitions in terms of *(i)* task featurizations and *(ii)* custom data structures are meant to instantiate STEEL-RT task and data templates. If defined by the user, the execution tree is interpreted and strictly checked for compatibility against application definitions. Otherwise, a default executor tree is inferred from the user-defined task featurizations. In any case, the executor tree will abstract away the target platform expressed in the headers generated after installation.

In summary, all information regarding application and system platform is merged by a C++ compiler supporting the latest standard to date (ISO C++ 17) [78]. At this point, library wrappers built at installation step may be linked. Note that only a small subset of all the code exposed in STEEL-RT template library may be instantiated –i.e., actually converted to executable machine code–.

In particular, the compilation stage does not only pursue the minimization of run-time errors by carrying out strict type-related checks, but also the minimization of run-time overhead is extremely important. For example, at compile-time not only task-to-executor compatibility is checked: a prior compile-time task-to-executor dispatching is implicitly performed, due to the fact that at scheduling time a task will only be able to be mapped to a compatible executor. In particular, a scheduler associated to a mapper executor that encompass a set of executors, when dispatching a task, will in general have a view of a subset of executors compatible with the compile-time featurizations of that task: if the task can only be mapped to a single executor, no ambiguity would have to be resolved and no additional scheduling runtime overhead will be required.

This follows the aforementioned principles of *(1) "don't pay for what you don't use"* and *(2) "move the computation from run-time to compile-time as much as possible"*.

4.1.3.3 Run-time

At this stage, the execution entry point consists of a STSE program. The whole application is simply represented as a single task to be fed by input data and expected to run in a single executor able to exploit all the compatible processing and memory resources in the platform. The executor tree can be considered as a pure function whose return type is just the result of the computation. The executor tree could potentially be composed of many executors, each with its own threads and internal states, but these complexities are inaccessible by the user, who can only interact with it *dispatching* or *assigning* a single task –representing the whole application, and in general decomposable into other tasks–, into it.

As mentioned, for each task arriving at each executor, and from the scheduler perspective, it is fully known at compile-time the dimensionality of either input (state) and output (action) spaces (see Sections 3.2.3 and 3.2.4). An empty action space is equivalent to a task-executor incompatibility, and results into a compile-time error. An action space composed by a single element for a task-executor pair guarantees that no scheduling needs to be done at run-time.

An action space greater than one requires some form of runtime scheduling to resolve the ambiguity. If the schedulers of all executors are endowed with *dynamic stateful scheduling policies* (see Section 3.2.3), their internal states can evolve as the execution proceeds, and they can be saved after each completed execution. With this idea, these internal states could be used to progressively

perform better (with respect to some metric of quality or reward) overall expanded schedules guiding the execution in order to favor those more promising execution paths, thus ultimately pursuing that successive executions of the binary under the same or similar application parameters progressively yield better results.

4.2 STEEL programming interface

The STEEL-API consists of two parts: (1) the *callable* interface, which consists of a set of template *functions* and template *types* to be *called* and *used* by the user, respectively; and (2) the *callee* interface, which comprises a set of rules that the user must follow in order to define template functions and template types that are called and used by STEEL-RT.

Specifically, the deployment and use of *executors* and the *guarding* of data objects via *guards* belong to the first part, while the definition of computational kernels and data structures belong to the second. The following sections expose the interface details starting from prior definitions regarding basic namespaces and enumerations (Section 4.2.1), followed with the rules for basic definition of tasks (Section 4.2.2), executor usage (Section 4.2.3), data structures definition (Section 4.2.4), and a set of auxiliary functions (Section 4.2.5). Finally, Section 4.2.6 exposes some compile- and run-time safety considerations and Section 4.2.7 explains how the execution of a task is *expanded* by defining specific *featurizations*.

4.2.1 Prior definitions

This section introduces some namespaces –in terms of what they encapsulate–, and enumerations –used for parametrization of types, functions and interfaces–, that will be presented along the following sections.

4.2.1.1 Namespaces

The STEEL-API exposes a set of types and functions for the user. All the following namespaces are contained within the main `steel` namespace: `executor`, `dep`, `sched`, `app`, and `detail`. In particular, `executor` encapsulates functions related to executor deployment and execution contexts; `dep` encapsulates both types and functions for dependency handling; `sched` exposes a series of template types to support different features for execution and scheduling relaxation. All user type and function definitions must be contained within the `app` namespace, which in practice encapsulates the *callee* interface (which is *user-defined* and STEEL-RT-*called*). Finally, `detail` encapsulates some STEEL-RT internal template-predicates defined to provide compile-time assertions that ensure compilation correctness of user-defined types and functions.

4.2.1.2 Enumerations

STEEL provides a set of useful enumerations, see Listing 4.1; `dep::kind` provides identifiers for read-only (`in`), read-write (`inout`), and write-only (`out`) data dependencies. `dep::view` identifies two possible data visibility modes for accessing remote or non-local data:
`dep::view::map` refers to *mapped* data resolution via internal (OS or external-library-defined) memory-mapping mechanisms, while `dep::view::copy` refers to *local* data resolution, in which data is copied to a local memory space. `ptype` provides identifiers for run-time values referring to fundamental data types. This enumeration is useful to provide data casting capabilities at run-time for mixed-precision computations. In particular, enumeration values in line 11 refer to `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `half`, `float`, `double`, `void`, and user-defined type (`ud`), respectively.

Listing 4.1: Persistent enumerations.

```
namespace steel {
  namespace dep {
    /* Dependency kind identifiers. */
    enum class kind {in, inout, out};

    /* Dependency visibility identifiers. */
    enum class view {map, copy};
  } /* namespace dep */

  /* Identifiers for fundamental types. */
  enum class ptype {c, uc, s, us, i, ui, l, ul, h, f, d, v, ud};
} /* namespace steel */
```

Listing 4.2: Enumerations for system-supported architectures and backends.

```
namespace steel::support {
  /* Supported executor/processor architectures. */
  enum class arch {...};

  /* Supported devices that parametrize executors. */
  enum class device {...};

  /* Supported backends for memory management. */
  enum class backend {...};
} /* namespace steel::support */
```

Moreover, Listing 4.2 exposes sets of identifiers used to tag currently supported processor architectures and backend libraries for memory management. These identifiers also permit the user to enable his / her own execution relaxations and to endow STEEL-RT for additional memory-management capabilities.

The STEEL post-install system-header will expose several enumerations used to identify certain available system-wide objects, processing devices, memory spaces and secondary backends (see Listing 4.3). Together with these enumerations, a set of *traits* will be defined for each of the values in the enumerations.

A set of user-defined enumerations within `app` are defined after a successful compilation (see Listing 4.4). They refer to identifiers for user-defined tasks (`app::impl`) and data structures (`app::data::impl`).

4.2.2 Task definition

The previously introduced `app::impl` enumeration encompasses user-defined task identifiers that refer to all different kinds of tasks that compose the application. This section explains how a task can be fully defined by means of the following attributes:

Listing 4.3: Post-install system enumerations.

```
namespace steel::sys {
  /* [...] = Comma-separated non-empty set of backend processors. */
  enum class processor {...};
  /* [...] = Comma-separated non-empty set of memory spaces. */
  enum class memspace {...};
} /* namespace steel::sys */
```

Listing 4.4: Post-compile application enumerations.

```
namespace steel::app {
  /* [...] = Comma-separated non-empty set of implementation identifiers. */
  enum class impl {...};
  namespace data {
    /* [...] = Comma-separated set of data structures implementation identifiers. */
    enum class impl {...};
    /* [...] = Comma-separated set of subregion data accessing identifiers. */
    enum class access {...};
  } /* namespace data */
} /* namespace steel::app */
```

Listing 4.5: Required type definitions for task traits.

```
namespace steel::app {
class traits<impl::X> {
public:
  /*  [...] = Comma-separated set of types. */
  using arg_t = std::tuple<[...]>;
  /*  [...] = Comma-separated set of dependency kind enumerations. */
  using arg_kind_t = util::kind_sequence<[...]>;
  [...] /*  Other definitions. */
};
} /* namespace steel::app */
```

- Argument types, dependency kinds in stateless `traits` class and task creation via `make_task`.
- Task featurization via stateless `kernel` class definition.
- Scheduling feed: static and data-dependent information within `traits` class scope.

4.2.2.1 Dependencies and task construction

A task identified as `app::impl::X` (or just `X` for simplicity) must provide type definitions for `arg_t` and `arg_kind_t` within a `traits` class specialized by `X` (see Listing 4.5). There are no requirements for the types in the `arg_t` `std::tuple`. The sequence `arg_kind_t` must have the same number of `dep::kind` enumerations as types defined in `arg_t`, and each dependency kind will correspond *one-to-one* to those types.

The correct definition of types and dependency kinds in `X` traits will permit the creation of a `X` task instance via the `make_task` call in Listing 4.6, whose return type is movable but not copyable. The following sections illustrate how task construction via `make_task` is expected to be carried out in only two contexts, namely: *(i)* in the `main` scope, and *(ii)* within the kernel body of a task restricted to run in an unfolder executor context.

4.2.2.2 Featurization

The previous `traits` class definition (see Listing 4.5) enables the creation of a task instance. In order for a task to be run on an executor, the user must first define a `kernel` stateless class.

Listing 4.6: Creation of task object of type `X`.

```
using namespace steel;
/*  [...] = Comma-separated set of objects of types in
    app::traits<app::impl::X>::arg_t. */
auto taskX = make_task<app::impl::X>([...]);
```

Listing 4.7: Skeleton for user-defined task featurization.

```
1  namespace steel::app {
2  /* Definition of kernel template class (already pre-set in steel library). */
3  template <impl::Impl, sys::processor ProcessorId>
4  class kernel : public std::false_type {};
5
6  /* User-defined restricted partial specialization of kernel class. */
7  template <sys::processor ProcessorId>
8  requires [user-defined compile-time predicate on ProcessorId]
9  class kernel<impl::X, ProcessorId> : public std::true_type {
10 public: /* Public scope (visible from STEEL-RT functions). */
11
12   using expand = /* Definition depending on feature and ProcessorId. */;
13
14   /* PT = a type defined from 'expand' definition.
15       [...] = comma-separated set of objects of types
16       app::traits<app::impl::X>::arg_t. */
17   static void run(PT param, [...]) {
18     /* Function body. */
19   }
20 };
21 } /* namespace steel::app */
```

Specifically, a task implementation `app::impl::X` is featured by partially specializing the `kernel` class with a task identifier (line 9 of Listing 4.7). This definition could be unconstrained or constrained over a *target processor identifier* `ProcessorId` via `requires` keyword (line 8) followed by a compile-time predicate. This way of task featurization through constraints relies on C++ *Concepts*, which is an idiom to constrain template parameters using compile-time predicates [61]. In the next section, it will be shown how specific definitions for `expand` type and `run` static function can be defined to enable the execution of a task instance in different execution contexts. In all cases, these definitions must be in the `public` class scope of a `app::kernel`, so that they are accessible at compile-time by STEEL meta-programs and callable by STEEL-RT at run-time.

Finally, in order to enable compile-time compatibility between user-defined task identifications `app::impl` and different execution contexts, a `public` derivation from a `std::true_type` of all `kernel` definitions is necessary (e.g., see lines 9 and 4 of Listings 4.7 and 4.9, respectively).

4.2.2.3 Featurization for a bottom executor

Skeletons for task featurizations of a task identifier `app::impl::X` are illustrated in Listings 4.8, 4.9, 4.10 and 4.11. Basically, the user must provide a type definition for `expand`, to be referenced by a STEEL-RT scheduler, for which there are several ways to do it. Also, the user must define a static function `void run` in which the computation is expressed, either by means of a call to an external library or by C++ code.

In the simplest case (Listing 4.8), the user just restricts the kernel execution to all processors satisfying the constraint in line 3 (also referred to the set of all *valid* or *compatible* processors). The `expand` definition in terms of `sched::just<ProcessorId>` (line 7) restricts the execution to a valid `ProcessorId` without expanding any execution parameter.

In the next case (Listing 4.9), the user declares its own sequence of options at compile-time by employing the `std::integer_sequence` type and the STEEL template `sched::relax`. From the definition in Listing 4.9, STEEL-RT spans an *expanded scheduling action space* whose size equals the number of elements in the sequence defined in `my_opts`. When the task is ready to be executed, STEEL-RT invokes the `run` function (line 13) passing the scheduling result as `param`, whose type `TI` is derived from `expand` definition.

Listing 4.8: Definition of a bottom executor kernel without expanding any option.

```
1  namespace steel::app {
2  template <sys::processor ProcessorId>
3  requires [user-defined compile-time predicate on ProcessorId]
4  class kernel<impl::X, ProcessorId> : public std::true_type {
5  public:
6    /* Mandatory type definition. */
7    using expand = sched::just<ProcessorId>;
8
9    /* Kernel definition. [...] = Arguments of arg_t tuple. */
10   static void run([...]) {
11     /* Kernel body. */
12   }
13 };
14 } /* namespace steel::app */
```

Listing 4.9: Definition of a bottom executor kernel and expansion of scheduling options.

```
1  namespace steel::app {
2  template <sys::processor ProcessorId>
3  requires [user-defined compile-time predicate on ProcessorId]
4  class kernel<impl::X, ProcessorId> : public std::true_type {
5    /* User declares its own set of options. TI is an integral type. */
6    using my_opts = std::integer_sequence<TI, UserOptions...>;
7
8  public:
9    /* Mandatory type definition. */
10   using expand = sched::relax<my_opts>;
11
12   /* Kernel definition. [...] = Arguments of arg_t tuple. */
13   static void run(TI param, [...]) {
14     /* Kernel body. User diverts the execution depending on [param] value. */
15   }
16 };
17 } /* namespace steel::app */
```

The user is also free to provide multiple sequences of options, as specified in Listing 4.10. In this case, the STEEL-RT spans an *expanded scheduling action space* with as many dimensions as number of sequences, and an action space resulting from the cross-product of all values in the sequences. The number of sequences is in practice unlimited, as it depends on the maximum number of recursive template instantiations in the compiler.

In some cases, a kernel execution may need an *opaque type* to be retrieved by STEEL-RT before execution. They could be architecture-dependent types set at install- or compile-time, or library-dependent handle types defined in an external library header. In any of these previous cases, the `sched::fix` is used to define `expand` (see Listing 4.11) and `sched::fix_relax` (to be exposed in Section 4.2.7), from which a type for `param` kernel argument is also defined.

4.2.2.4 Featurization for an unfolder executor

The skeleton of a kernel able to run in an unfolder executor context is shown in Listing 4.12. In addition to a `param` argument to be passed together with the internal `arg_t` arguments, the `run` function must also accept a reference to a `delegate` executor (see line 16).

Specifically, `auto&` qualifier refers to the type of the executor on which the unfolder executor that will run the task is stacked. In this example, the type of the `param` argument is derived from `expand` and matches `TI`. Equivalently to Listing 4.10, a multidimensional set of options could be used.

Listing 4.10: Skeleton for a bottom executor kernel and a expansion of multidimensional options.

```
template <sys::processor ProcessorId>
requires [user-defined compile-time predicate on ProcessorId]
class kernel<impl::X, ProcessorId> : public std::true_type {
  /* User declares its own sets of options. TA, TB, TC, ... are integral types. */
  using my_opts_a = std::integer_sequence<TA, ValuesA...>;
  using my_opts_b = std::integer_sequence<TB, ValuesB...>;
  using my_opts_c = std::integer_sequence<TC, ValuesC...>;
  /* [...] = Possibly more sequence definitions */
  [...]
public:
  /* Mandatory type definitions. */
  using expand = sched::relax<my_opts_a, my_opts_b, my_opts_c, ...>;

  /* Kernel definition.
    [,,,] = Possibly more parameters. [...] = Arguments of arg_t tuple. */
  static void run(TA paramA, TB paramB, TC paramC, [,,,], [...]) {
    /* Kernel body. User diverts the execution depending
        on [param] value which has the type std::tuple<TA, TB, TC, ...>. */
  }
};
} /* namespace steel::app */
```

Listing 4.11: Definition of a bottom executor kernel using a built-in argument expansion.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires [user-defined compile-time predicate on ProcessorId]
class kernel<impl::X, ProcessorId> : public std::true_type {
public:
  /* Mandatory type definitions. */
  using expand = sched::fix<[Type derived from ProcessorId]>;
  /* PT = type derived from expand definition. [...] = Arguments of arg_t tuple. */
  static void run(PT param, [...]) {
    /* Kernel body. */
  }
};
} /* namespace steel::app */
```

Listing 4.12: Skeleton for a kernel to be run on an unfolder executor.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires constraint<ProcessorId>::support_device == support::device::abstract
class kernel<impl::X, ProcessorId> : public std::true_type {
private:

  /* User declares its own set of options for unfolding. TI is an integral type. */
  using my_opts = std::integer_sequence<TI, UserUnfoldingOptions...>;

public:
  /* Mandatory type definition. */
  using expand = sched::relax<my_opts>;

  /* Kernel definition, to be called by STEEL runtime.
     [...] = Arguments of arg_t tuple. */
  static void run(auto& delegate, TI param, [...]) {
    /* Kernel body.
       User divert the execution depending on [param] value and
       creates and forwards tasks to [delegate] with a call of the form
       delegate( std::make_task<impl::Y>(...) ); */
  }
};
} /* namespace steel::app */
```

Section 5.2 will detail how the format of Listing 4.12 can be used to express data partitions and reimplementations, also possibly with mixed precision computations.

4.2.2.5 Featurization for a mapper executor

As specified in Table 3.2, tasks will be free to be run on several executors encompassed by a mapper executor if the task exposes a `kernel` definition compatible with at least one executor. For this reason, there is no need for a user-defined `kernel` constrained for *mappers*, as previous featurizations in Sections 4.2.2.3 and 4.2.2.4 will implicitly restrict a task instance to be run by a mapper executor.

4.2.2.6 Scheduling feed

As mentioned in Section 3.2.4, expanded schedulers that equip executors perform task scheduling actions based on certain information exposed by the task. In Section 3.2.4.2, it was mentioned why the scheduling could be influenced by data-dependent information. In terms of the STEEL-API interface, the user must define a compile-time value `feedback` for any task `X`, which specifies the kind of feedback that a task instance with `arg_t` arguments is able to provide to the STEEL-RT.

As specified in Listing 4.13, if `feedback` is set to `static` or `static_data`, the corresponding `static_feed_t` type and `get_static_feedback` function definitions must be given. Similarly, if `feedback` is set to `data` or `static_data`, the corresponding `data_feed_t` type and `get_data_dependent_feedback` function definitions must be provided. Finally, if `feedback` is set to `none`, there is no need to provide these types and function definitions. If defined, `static_data` and `data_feed_t` must satisfy a STEEL compile-time predicate `sched::is_castable_to_numeric_v`, that asserts that values of these types can be transformed into an array of numerical values to be interpretable by a expanded scheduler. For example, a tuple of heterogeneous numeric types (e.g., `char`, `int`, `float`, etc.) will qualify that predicate.

Listing 4.13: Type and function definitions for feedback-based scheduling.

```
1  namespace steel::app {
2  struct traits<impl::X> {
3    /* Scheduling feedback definition in the set [static, data, static_data, none]. */
4    static constexpr sched::feed_kind feedback = FEEDBACK_KIND;
5
6    /* Scheduling feedback type for data-dependent feedback.
7          Required if feedback is [data] or [static_data]. */
8    using static_feed_t = SFT;
9
10   /* Scheduling feedback type for data-dependent feedback.
11         Required if feedback is [static] or [static_data] */
12   using data_feed_t = DFT;
13
14   /* Definition of function callable from a STEEL runtime scheduler
15         to get static information. Required if feedback is [data] or [static_data].
16         [...] = Arguments of arg_t tuple. */
17   static static_feed_t get_static_feedback([...]) {
18     /* Function body. A set of parameters is extracted from task arguments
19                   and then returned.*/
20   }
21
22   /* Definition of function callable from a STEEL runtime scheduler to get
23         data-dependent information. Required if feedback is [data] or [static_data].
24         [...] = Arguments of arg_t tuple. */
25   static static_feed_t get_data_dependent_feedback([...]) {
26     /* Function body. A set of parameters is extracted from data in task
27                   arguments and returned.*/
28   }
29 };
30 } /* namespace steel::app */
```

Listing 4.14: Deployment of a bottom executor.

```
1 using namespace steel;
2 /* Deployment of a Bottom executor tied to processor P. */
3 auto& execRefP = executor::deploy<sys::processor::P>();
```

4.2.3 Executor deployment and use

As explained in Section 4.1.3, the information about the available processing hardware and memory spaces is gathered at install time, restricting the kind of bottom executors that can be deployed at run-time. Top executors are always deployed on already deployed top or bottom executors.

4.2.3.1 Deployment of a bottom executor

For each `sys::processor` identifier declared in line 3 of Listing 4.3, only one bottom executor can be deployed. Let `P` be an element in `sys::processor`, then an executor is built according to line 2 of Listing 4.14 and a reference (`&`) to it is returned. If another executor is asked for deployment over the same processor more than once, a run-time exception is thrown. Features of executors depend on the hardware and the installation process, and they are embedded at compile-time in the executor type (hidden in `auto` qualifier in line 2 of Listing 4.14).

4.2.3.2 Deployment of a top executor

Listing 4.15 exposes how an *unfolder executor* is deployed on top of a bottom executor (lines 4 and 7) or on top of another top executor (line 10).

Listing 4.15: Deployment of a Unfold executor.

```
1  using namespace steel;
2
3  /* Deployment of a Bottom executor tied to processor P. */
4  auto& executorBottomA = executor::deploy<sys::processor::A>();
5
6  /* Deployment of a Unfolder executor on top of previous executor. */
7  auto& execRefU = executor::deploy( executorBottomA );
8
9  /* Deployment of another Unfolder executor on top of the previous. */
10 auto& anotherexecRefU = executor::deploy( execRefU );
```

Listing 4.16: Deployment of different Mapper executors.

```
1  using namespace steel;
2
3  /* Deployment of a Bottom executor tied to processor P. */
4  auto& execBottomA = executor::deploy<sys::processor::A>();
5  auto& execBottomB = executor::deploy<sys::processor::B>();
6  auto& execBottomC = executor::deploy<sys::processor::C>();
7  /* [...] = Possibly more bottom executor deployments.  */
8  ...
9
10 /* Deployment of a Mapper encompassing Bottom executors. */
11 auto& execMapper = executor::deploy(execBottomA, execBottomB, execBottomC);
12
13 /* Deployment of a Unfolder on top of a Bottom tied to processing device X. */
14 auto& execBottomX = executor::deploy<sys::processor::X>();
15 auto& execUnfoldX = executor::deploy(execBottomX);
16
17 auto& execBottomY = executor::deploy<sys::processor::Y>();
18
19 /* Deployment of a Mapper encompassing both Top and Bottom executors. */
20 auto& execMapper2 = executor::deploy(execMapper, execUnfoldX, execBottomY);
```

Mapper executors are meant to be used for heterogeneous dispatching, hence they can be deployed on several *Top* or *Bottom* executors in any combination, as specified in lines 11 and 20 of Listing 4.16).

4.2.3.3 Batch and conditional deployment

A set of executors satisfying a condition can be deployed and wrapped into a single mapper executor using the `deploy_only` function. In Listing 4.17, four mappers executors are deployed *(i)* from a set of cores in the form of a thread pool (line 2); *(ii)* from all CUDA GPU devices (line 3); *(iii)* from all SMP processors in a NUMA system (line 4); *(iv)* from all processors able to execute tasks by forking threads (line 5); and *(v)* from all lowest-level processors that can be handled by a bottom executor (e.g., all CPU cores and GPUs) (line 6). A similar template function `deploy_first` is also provided, to which the same template parameters can be passed. Contrary to `deploy_only`, `deploy_first` only deploys the first *non already deployed* bottom executor satisfying the template parameter.

4.2.3.4 Forwarding a task to an executor

Task objects are non-copyable types and they can only be moved. Listing 4.18 shows –in lines 4 and 9, respectively– two ways in which a task can be dispatched to an executor referenced by `execRef`. The second way (line 9) immediately dispatches the task to the executor without the

Listing 4.17: Deploy and group executors based on a condition.

```
1 using namespace steel;
2 auto& mapperOnThreadPool = executor::deploy_only<executor::sequential>();
3 auto& mapperOnCudaGpus = executor::deploy_only<executor::cuda_device>();
4 auto& mapperOnSmpNodes = executor::deploy_only<executor::multicore>();
5 auto& mapperOnForkers = executor::deploy_only<executor::forker>();
6 auto& mapperOnLowest = executor::deploy_only<executor::all_lowest>();
```

Listing 4.18: Two possible ways for task construction and running.

```
1 /* Task creation. [...] = objects from which the task X can be built. */
2 auto taskX = make_task<app::impl::X>([...]);
3 /* Dispatch created task to executor. */
4 execRef( std::move(taskX) );
5 /* Warning. taskX object is invalid in this scope and must not be used. */
6
7 /* Create and dispatch task instance Y.
8 [...] = objects from which the task Y can be built. */
9 execRef( make_task<app::impl::Y>([...]) );
```

need of naming the task object (thus eliminating the possibility of using an invalid object), so this way is encouraged for simplicity and safety reasons.

4.2.4 Definitions for custom data structures

The argument types within `arg_t` type, defined in `app::traits`, can be of any type, but in real applications tasks could refer to data arguments that cannot fit in the program stack –i.e., heap allocation is required–. For these cases, the user can define its own heap-allocatable data structures within the `app` namespace by means of `app::data::traits` template interface, parametrized with a `app::data::impl` enumeration type that identify the user-defined data structure. Moreover, the user can express different data-accessing modes useful to refer to sub-regions of data meant to be used by partitionable kernels, and different *casting* modes useful to represent data in a compatible type representation (e.g., floating point conversions in different binary representation such as 16, 32, and 64-bit, for mixed-precision applications).

The following requirements will need to be satisfied for those user-defined data structures so that STEEL-RT can entirely manage the associated data objects. Specifically, *managed* refers to the ability to manage allocation / deallocation, concurrent access, and movement across memory spaces in a transparently fashion at run-time and without user intervention.

4.2.4.1 Heap-allocatable data

This section exposes the required interface in terms of types and free functions that the user is responsible to define, so that a *data descriptor object* (*data object* or *datum* terms are used in the following) can be entirely managed by the STEEL runtime.

The first type defined in line 5 of Listing 4.19 is `descriptor_t`, which must include an `address_t` type able to store a memory location in the form of a pointer. In practice, this can be a `void*`, a `std::any`, or any other type from which a `void*` can be returned, thus `get_address` can be defined (line 8). After this is considered, the user is free to declare any type that parametrize the data structure (`[...]`), to be used to access the data. Specifically, an object of type `descriptor_t` is a standalone datum, in the sense that all the information encapsulated must be accessible through the `address_t` value and the parameters with types `[...]`.

The remaining functions `get_element_count`, `get_byte_count`, `is_contiguous` and `make_allocatable` are used for allocation and memory copying purposes. If `DS` is completely

Listing 4.19: Basic traits for a user-defined data structure with identifier `DS`.

```
1  namespace steel::app::data {
2  class traits<data::impl::DS> {
3  public:
4    /* Definition of the data descriptor. [...] = Any sequence of types. */
5    using descriptor_t = std::tuple<address_t, [...]>;
6
7    /* Retrieve the void* from the address_t element. */
8    void * get_address(const descriptor_t& datum);
9
10   /* Interface for retrieving the number of elements encompassed in a datum. */
11   static std::size_t get_element_count(const descriptor_t& datum);
12
13   /* Interface for retrieving the size of a datum in bytes. */
14   static std::size_t get_byte_count(const descriptor_t& datum);
15
16   /* Predicate to check whether the information stored in datum
17   is in a compact memory region. */
18   static bool is_contiguous(const descriptor_t& datum);
19
20   /* Return the fundamental type of datum, if exists. */
21   static ptype get_ptype(const descriptor_t& datum);
22
23   /* Make a tabula-rasa version of datum. */
24   static descriptor_t make_allocable(const descriptor_t& datum);
25
26   /* Data copy between descriptors. */
27   static void copy(descriptor_t& destinDatum, const descriptor_t& sourceDatum,
28       copy_function_t copyCallback);
29
30   /* Data casting between descriptors. */
31   static void copy_cast(descriptor_t& destinDatum, const descriptor_t& sourceDatum,
32       cast_function_t castCallback);
33 };
34 } /* namespace steel::app::data */
```

composed by a set of uniform objects, the function `get_element_count` (line 11) retrieves the number of objects and `get_byte_count` (line 14) returns a multiple of the number of objects. Function `is_contiguous` (line 18) returns a `true` value only if all the information encompassed by `datum` is contained within the memory range spanned from its `address_t` value and the bytes returned by `get_byte_count(datum)`.

The function `get_ptype` is used to return a run-time type identifier that refers to a type to which the pointer returned by `get_address` can be casted to. If that type is not a fundamental type, the return value can be `ptype::custom`.

Finally, the function `make_allocatable` (line 24) returns an *allocatable* or *tabula-rasa* version of the input `datum` (with the address set to `nullptr` or `0x0`). In particular, if `datum` is contiguous, the returned type corresponds to another datum with the same parameters but the *zero*-ed address. Otherwise, the parameters must also be modified so that the returned descriptor object is a *compacted* version of the input `datum`. A detailed explanation about this *compacting* process is deferred to Section 5.2.2 as it requires a prior explanation of *data partitions* in Section 4.2.4.3. In particular, data compactions will be required to enable the use of persistent storage spaces, and to improve cache-locality and data movement performance.

A general `datum` is said to be *allocatable* if it satisfies the STEEL-RT internal predicate of Listing 4.20. This predicate is designed to verify valid user-definitions of the `make_allocatable` function.

Functions in lines 27 and 31 are needed for copying and casting data between two objects. These definitions will be needed only if `is_contiguous` returns `false` for one or both objects. If the data structure is one such that it will always return `true` for all objects, these functions

Listing 4.20: Internal runtime predicate to check whether a data object is allocatable.

```
1  namespace steel::detail {
2    template <app::data::impl DS>
3    bool is_allocable(const app::data::traits<DS>::descriptor_t& datum) {
4      return app::data::traits<DS>::is_contiguous(datum) &&
5          (datum == make_allocable(datum));
6    }
7  } /* namespace steel::detail */
```

Listing 4.21: Function types for copying and `copy`, `copy_cast` user definitions.

```
1  namespace steel {
2  using copy_function_t = std::function<void(void*,void*,std::size_t)>;
3  using cast_function_t = std::function<void(void*,void*,ptype,ptype,std::size_t)>;
4  } /* namespace steel */
5
6  namespace steel::app::data {
7  /* Definition for always-contiguous data objects. */
8  static void traits<data::impl::DS>::copy(
9      descriptor_t& dstDatum, const descriptor_t& srcDatum,
10     copy_function_t copyCallback) {
11   copyCallback(get_address(dstDatum), get_address(srcDatum),
12       get_byte_count(dstDatum));
13 }
14
15 /* Definition for always-contiguous data objects. */
16 static void traits<data::impl::DS>::copy_cast(
17     descriptor_t& dstDatum, const descriptor_t& srcDatum,
18     cast_function_t castCallback) {
19   castCallback(get_address(dstDatum), get_address(srcDatum),
20       get_ptype(dstDatum), get_ptype(srcDatum), get_element_count(dstDatum));
21 }
22 } /* namespace steel::app::data */
```

can be reduced to Listing 4.21 definitions in lines 8 and 16. The type definitions of functions `copy_function_t` and `cast_function_t` are defined in lines 2 and 3.

Section 5.2.2 will expose in detail some other examples for simple 2D data buffers, for which non-contiguous objects can exist.

Finally, as descriptors themselves are not meant to be used as valid type-arguments, it is required another type that *lifts* a user-defined type into a `app::data::impl` tagged-type. Specifically, the definition of `descriptor_t` for `DS` enables the definition of a *tag-lifted descriptor template type* called `handle_t<DS>`, which ultimately can be used to define for a task `app::impl::X` a `app::traits<app::impl::X>::arg_t` that refer to *heap-aware* arguments (see Listing 4.22).

4.2.4.2 Data guarding

The process of *data guarding* enable concurrent access of user-defined data objects from different execution contexts provided by executors. Lines 3 to 7 of Listing 4.23 show the template signature of the `handle` pure function, which in practice returns a guard that wraps a data descriptor object with identifier `app::data::impl::DS`. This function requires an allocation context by means of the top allocator referenced by an executor. Identifiers `dep::view::map` and `dep::view::copy` specify the visibility mode when the datum is not local to a processing device. The `dep::view::map` option results into a *mapped* or *on-demand* visibility, by which data regions are implicitly copied to the local memory space as page faults occur. The `dep::view::copy` option specifies a *local* visibility for the data, which implies that a full

Listing 4.22: Basic traits for a user-defined data structure with identifier `DS`.

```
namespace steel::app {
/* Template type handle_t lifts user-defined descriptors into app::data::impl-tagged
types, so that they can be used as task arguments in arg_t definition (line 12). */
template <app::data::impl DS>
using handle_t = util::tag_t<DS, data::traits<DS>::descriptor_t>;

class traits<impl::X> {
public:
  /* Definition task arguments as heap-aware data handles.
  [...] = Any sequence of types. */
  using arg_t = std::tuple<handle_t<DS_A>, handle_t<DS_B>, [...]>

  /* Corresponding dependency kinds of arg_t elements. */
  using arg_kind_t = util::kind_sequence<DepKind_A, DepKind_B, [...]>;
};
```

Listing 4.23: Interface for dependency initialization.

```
namespace steel::dep {
template <
  typename ExecutorT, /* Executor type to provide an allocation context. */
  typename InitT, /* A type from which a data descriptor object can be built. */
  app::data::impl DS, /* Data identifier. */
  dep::kind DepKind, /* Dependency kind of data {in, inout, out} */
  dep::view View /* Placement mode for accessing {map, copy}. */
>
auto handle(ExecutorT& , const InitT);
} /* namespace steel::dep */
```

copy of the data must be present in the local memory space before the execution starts.

In this line, a set of *data acquisition* actions are exposed in Table 4.3, classified according to the dependency kind and the data visibility specified during data guarding. In the runtime side, data guarding with *mapped* visibility require a *binding* between a pair of memory spaces with software or hardware support for data coherence. In addition, as executors are organized hierarchically, the visibility modes in nested executor contexts must also be hierarchically composable –e.g., a dependency guarded as *copy* or *map* in an executor context could be independently guarded in a lower-level executor context (either the whole dependency or subregions of it)–. Hence, as subregions of an already *guarded-as-mapped* data dependency may be also guarded as mapped in a lower-level executor, the corresponding actions would be either inheriting the existing binding or create a new binding (see `dep::view::map` column).

	`dep::view::copy`	`dep::view::map`
`dep::kind::in`	Copy to main memory	Inherit binding or create new
`dep::kind::inout`	Exclusive copy to main memory	Inherit binding or create new
`dep::kind::out`	Allocate on main memory	Allocate on main memory

Table 4.3: Data acquisition actions depending on the dependency kinds (`dep::kind::in`, `dep::kind::inout` and `dep::kind::out` (read-only, read-write and write-only, respectively), and the visibility mode (`dep::view::copy`, `dep::view::map`). The guarding is associated to an executor context, and the *main memory* refers to the main memory space of that executor context.

Similarly, when a task execution has finished, a set of *data publication actions* are summarized in Table 4.4, which are internally done by STEEL-RT to ensure correct data visibility across sev-

Listing 4.24: Dependency initialization from a string.

```
1  using namespace steel;
2  { /* Start of scope. */
3  /* [...] = A context with an executor reference 'execRef'. */
4  [...]
5
6  /* Create a dependency from a string. [execRef] is a deployed
7  executor under which the dependency is expected to be accessed. */
8  auto dataGuard1 = dep::handle<DS_A, DepKind_A, Place>(execRef, "stringDesc");
9
10 /*** Create a dependency from another descriptor. ***/
11 /* User-initialized descriptor. */
12 app::data::traits<DS>::descriptor_t userDatum = /* Custom initialization. */;
13
14 /* Build a data guard from the descriptor and an executor. */
15 auto dataGuard2 = dep::handle<DS_B, DepKind_B, Place>(execRef, userDatum);
16
17 execRef( make_task<app::impl::X>(dataGuard1, dataGuard2) );
18 } /* End of scope. */
```

eral memory spaces and to manage the use of the system memory resources. In case of *map* visibility, a synchronization before unbinding is needed to ensure the visibility of read-write and write-only data. In case of *copy* visibility, the data is registered as a candidate datum to be *evictable*, so it can be saved to other memory space before eviction according to some cache-eviction policy.

	`dep::view::copy`	`dep::view::map`
`dep::kind::in`	Register as evictable	Unbind if not inherited
`dep::kind::inout`	Register as evictable	Synchronize and unbind if not inherited
`dep::kind::out`	Register as evictable	Synchronize and unbind

Table 4.4: Data publication actions depending on the dependency kinds (`dep::kind::in`, `dep::kind::inout` and `dep::kind::out` (read-only, read-write and write-only, respectively), and the visibility mode (`dep::view::copy`, `dep::view::map`).

One possible use of `handle` template function is exposed in Listing 4.24, in which a data descriptor is built from a string (line 8). This instruction is considered legal only if the user has provided a function definition to build a data descriptor object from a string (see Listing 4.25). The returned `datum` object must be *allocatable*. Secondly, a guard can also be built from a plain descriptor created by the user (see lines 12 and 15 of Listing 4.24).

Assuming that `app::impl::X` exposes a `DS_A` and `DS_B` as `arg_t = std::tuple <handle_t<DS_A>, handle_t<DS_B>>` in its traits, previous guard creations (lines 8 and 15 of Listing 4.24) permit a task `app::impl::X` construction and executor dispatching as expressed in line 17. Finally, at the end of the scope (line 18) the destruction of `dataGuard1` and `dataGuard2` are triggered, which will automatically block until the task `app::impl::X` instance is finished.

Note that functions in lines 5 and 8 of Listing 4.25 are opposite operations. The function `get_string_descriptor` is not used for guard initialization but will be required in the context of file operations (see Section 4.2.5.2).

4.2.4.3 Data accessing

This section exposes the additional interface to be defined by the user so that *subregions* or *pieces*, combined with possible *representations*, can be accessed individually and concurrently.

Listing 4.25: String conversions for `DS` data identifier.

```
1  namespace steel::app::data {
2  class traits<data::impl::DS> {
3  public:
4    /* Pure function to transform a string into a descriptor. */
5    static descriptor_t get_descriptor(const std::string filename);
6
7    /* Pure function to transform a string into a descriptor. */
8    static std::string get_string_descriptor(const descriptor_t& datum);
9  };
10 } /* namespace steel::app::data */
```

Listing 4.26: Subregion access.

```
1  using namespace steel;
2  { /* Start of scope. */
3
4  /* [...] = A context with an executor reference 'execRef'. */
5  [...]
6
7  /* Creation of a parent guard. */
8  auto parentGuard = /* Initialize via dep::handle, execRef and a data descriptor. */;
9
10 /* User initialization of sub-data descriptors. */
11 using subdata_desc_t = app::data::traits<app::data::impl::DS>::subdata_descriptor_t;
12 subdata_desc_t childDescriptorA = /* Initialize */;
13 subdata_desc_t childDescriptorB = /* Initialize */;
14
15 /* Get references of children guards. */
16 auto& childGuardA = parentGuard.get(childDescriptorA);
17 auto& childGuardB = parentGuard.get(childDescriptorB);
18
19 /* Creation and dispatch of a task Y with child dependencies. */
20 execRef( make_task<app::impl::Y>(childGuardA, childGuardB) );
21
22 } /* End of scope. */
```

Section 4.2.4.4 introduces the STEEL interface exposed to let the user perform custom memory allocation inside a kernel context.

Function `handle` in line 2-9 of Listing 4.23 is not only used to enable concurrent access and full STEEL-RT management to heap-allocatable task arguments. With additional function definitions, the user can access to subregions of a data guard by means of the `get` method, and those subregions will acquire their own protections for finer-granularity concurrent access.

Listing 4.26 illustrates how to accomplish this goal. Assuming a `parentGuard` (line 8) already constructed (by any of the methods exposed in Listing 4.24), and descriptors that fully characterize subregions of data (lines 12-13), references to children guards can be created (lines 16-17). From them, a task `app::impl::Y` that take these subregions as dependencies can be built and dispatched (line 20). Additionally, at the end of the scope (line 22), the deletion of `parentGuard` object will ensure that all children dependencies have been resolved –i.e., task instance `app::impl::Y` has finished–.

In order for the code in Listing 4.26 to work, and assuming `app::data::impl::DS` to be the type of the parent data descriptor, the user must provide the `subdata_descriptor_t` type and a function definition `key_to_descriptor`, as exposed in lines 5 and 8 of Listing 4.27, respectively.

Section 5.2 will expose how this functionality can be applied to specific data structures such as bi-dimensional arrays, in which bi-dimensional subregions of data can be expressed for data-

Listing 4.27: User definitions for subregion access.

```
1  namespace steel::app::data {
2  class traits<data::impl::DS> {
3  public:
4    /* Type definition for sub-data characterization. */
5    using subdata_descriptor_t = /* Any type. */;
6
7    /* Function to build data descriptors of sub regions of a parent descriptor. */
8    static descriptor_t key_to_descriptor(
9              const descriptor_t& parentDesc, const subdata_descriptor_t& childDesc);
10 };
11 } /* namespace steel::app::data */
```

Listing 4.28: Interface for context-aware RAII-based memory allocation.

```
1  using namespace steel::executor;
2  template <typename ExecutorT> /* Executor type to provide an allocation context. */
3  auto raw_allocate(ExecutorT& execRef, std::size_t bytes);
```

partitioning and mixed-precision purposes.

4.2.4.4 Custom memory allocation

Users can leverage the STEEL-internal RAII-based allocation mechanisms to request the creation of raw data buffers –i.e. not identified by a `app::impl`– inside the scope of an unfolder executor. Listing 4.28 exposes the function signature `raw_allocate` which performs a data allocation in the storage context abstracted by the *top allocator* internally referenced by an `ExecutorT` object.

Line 7 of Listing 4.29 invokes this function and returns a temporary buffer object whose data can then be used. At the end of the scope (line 11), RAII-idiom guarantees a proper release of memory resources in a transparent fashion.

4.2.5 Auxiliary interface

In order to complete the full STEEL-RT API function definitions, auxiliary functions related to input / output and to the execution entry point are exposed next.

Listing 4.29: Context-aware RAII-based memory allocation from user side.

```
1  using namespace steel;
2  { /* Start of scope. */
3    /* [...] = A context with an executor reference 'execRef'. */
4    [...]
5
6    /* Memory allocation request within the storage context referenced by execRef. */
7    auto temporaryBuffer = executor::raw_allocate( execRef, nBytes );
8
9    /* Access to the actual data. */
10   void * dataPtr = temporaryBuffer.get();
11
12   /* [...] = Use allocated memory in dataPtr. */
13   [...]
14 } /* End of scope. */
```

Listing 4.30: Function for parsing program arguments.

```
1 using parse_map_t = std::map<std::string, std::string>;
2 parse_map_t steel::build_parse_map(int argc, char** argv);
```

Listing 4.31: Identification by mount-point of a persistent-storage memory space identified as `MS`.

```
1 namespace steel::sys {
2 struct traits<memspace::MS> {
3   static constexpr bool is_persistent = true;
4   static constexpr const char* source_path = "[PATH_MOUNT_POINT]";
5   /* Other definitions ... */
6 };
7 } /* namespace steel::sys */
```

4.2.5.1 Parsing main program arguments

The arguments passed to a C/C++ program by command line can be parsed by `build_parse_map` function, as specified in line 2 of Listing 4.30. This function returns a C++-standard associative container `std::map` that interprets command line arguments in the form `--parameter_name = parameter_value` as *key-to-value* pairs.

4.2.5.2 Data dependencies in persistent memory spaces

System-dependent memory spaces expose a set of traits defined at install-time. Memory spaces associated to persistent storage devices, either in the form of hard-disk, solid-state, or non-volatile memory, are qualified as *persistent* and expose a compile-time path identification in its traits (see lines 3-4 of Listing 4.31), which is used to uniquely associate file names to them.

Previous source-path association to a memory space and the use of `handle_dependency` function exposed in Listings 4.23 and 4.24, enable the treatment of binary files as data dependencies to be safely accessed by tasks.

Specifically, line 6 of Listing 4.32 creates a data guard, so it can be directly read (`dep::kind::in`), read-written (`dep::kind::inout`) or written (`dep::kind::out`) by tasks concurrently. Depending on the `dep::view` identifier, data access is performed either as *mapped* (`dep::view::map`) –using internal OS mechanisms of file mapping–, or as *copy* (`dep::view::copy`) letting STEEL-RT to make a full copy when data are requested in a different memory space. Persistent memory space in which the file is stored is identified by matching `FULL_PATH_FILENAME` against source_path definition of all persistent memory spaces identified at install-time.

Recalling the RAII-nature of a data guard exposed in Section 4.2.4.2, at the end of the scope

Listing 4.32: Guarding a binary file as a data dependency.

```
1 using namespace steel;
2 /* [...] = A context with an executor reference 'execRef'. */
3 [...]
4
5 { /* Start of scope. */
6 /* Treat a file as a task dependency. */
7 auto fileGuard = dep::handle<app::data::impl::DS, DepKind, Place>(
8     execExec, "[FULL_PATH_FILENAME]");
9 } /* End of scope. */
```

Listing 4.33: Run of a STEEL program from command line.

```
$ ./steel_binary \
  --input=/media/spaceA/dataX.bin \
  --inoutput=/media/spaceB/dataY.bin \
  --output=/media/spaceC/dataZ.bin
```

(line 9), a proper release of resource is transparently performed, first blocking until all tasks no longer need to access the guarded data. Second, if `DepKind` is `dep::kind::inout` or `dep::kind::out`, data is synchronized back to ensure visibility from the persistent storage.

Finally, memory spaces qualified as *persistent* in its traits can be accessed from any allocation context of an executor scoping *node-* or *lower-level* execution contexts.

4.2.5.3 Example of a STSE program

An example of a main program that illustrates the STSE paradigm is exposed in Listing 4.34, which is compiled into a `steel_binary` executable. This program could correspond to an application requiring three arguments with *read-only*, *read-write* and *write-only* access modes. These arguments are meant to be task dependencies of the whole application viewed as a *Single Task - ST*, and correspond to different user-defined data structures identified as `app::data::impl::DSX`, `app::data::impl::DSY` and `app::data::impl::DSZ`, respectively.

In Listing 4.33, three arguments corresponding data dependencies are passed to the executable. Each of the files are located in three different persistent memory spaces with different mount points (`/media/spaceA`, `/media/spaceB` and `/media/spaceC`) which must be referenced in the corresponding memory space traits defined at install-time.

Arguments in command line are passed as `argc` and `argv` parameters at program entry point (line 5 of Listing 4.34) and parsed (line 9). Secondly, in line 11 a system-wide executor is deployed automatically from the system information defined at install-time and the application information defined at compile-time.

Data guards are built (lines 14-26) from the files specified in the arguments. Specifically, data in `dataX.bin` file is treated as a *read-only* datum located in memory space A (line 14), and from `dep::view::copy` a copy to local memory space is needed in order for the task to be ready to run. Contrary, data in `dataY.bin`, as required by `dep::view::map`, is mapped to the local memory space and transferred on page-fault basis when needed (line 19). Finally, `dataZ.bin` is considered as a *w*rite-only dependency which will require an allocation in local memory space (as required by `dep::view::copy`), without requiring any fetch from space C.

Lines 29-30 of Listing 4.34 leverage the STSE pattern: an application-wide *Single Task - (ST)* is created and asynchronously dispatched to the system-wide *Single Executor - (SE)*. Under the hood, multiple tasks could be created and transparently dispatched across several executor contexts, depending on the user-defined relaxations by which the given application has been featured. Similarly, data is distributed possibly across a deep memory hierarchy in a coherent and transparent way. Finally, the caller thread blocks at the end of the scope (line 31) until data guarded by previous guards are no longer accessible –or equivalently, until the computation finalizes–.

In lines 7 and 35 optional (although recommended) calls wrapped in initialization and finalization macros are performed to handle eventual exceptions triggered from any thread.

Listing 4.34: Example of the `main` entry point of a STSE program.

```
#include <steel.hpp>
using namespace steel;

/* Program entry point. */
int main(int argc, char** argv) {
  /* Optional initialization of exception-aware scope. */
  STEEL_TRY();
  { /* Start of scope. */
    const auto pMap = build_parse_map(argc, argv);
     /* Deploy system-wide single executor. */
    auto& mainExec = executor::deploy_main();

    /* Treat binary file referenced in --input as a read-only dependency. */
    auto readOnlyGuard = dep::handle<
        app::data::impl::DSX, dep::kind::in, dep::view::copy>(
            mainExec, pMap["--input"]);

    /* Treat binary file referenced in --inoutput as a read-write dependency. */
    auto readWriteGuard = dep::handle<
        app::data::impl::DSY, dep::kind::inout, dep::view::map>(
            mainExec, pMap["--inoutput"]);

    /* Treat binary file referenced in --output as a write-only dependency. */
    auto writeOnlyGuard = dep::handle<
        app::data::impl::DSZ, dep::kind::out, dep::view::copy>(
            mainExec, pMap["--output"]);

    /* Create task and dispatch it to the executor. */
    mainExec(
        make_task<app::impl::T>(readOnlyGuard, readWriteGuard, writeOnlyGuard));
  }  /* End of scope. Blocking until computation is finished. */

  /* Finalization of exception-aware scope. Eventual exceptions
    from any thread are handled. */
  STEEL_CATCH();

/* End of main scope. Executors are destroyed and all resources released. */
  return 0;
}
```

Listing 4.35: Compile-time predicate to check application-processor compatibility.

```
1 namespace steel::detail {
2 template <app::impl X, sys::processor ProcessorId>
3 constexpr bool are_compatible_v = app::kernel<X, ProcessorId>::value
4 } /* namespace steel::detail */
```

4.2.6 Internal safety patterns

Robustness is a crucial characteristic in any runtime system. In this section, different patterns that endow STEEL-RT with compile- and run-time safety mechanisms are exposed.

4.2.6.1 Compile-time task-executor compatibility

Every template function exposed to the user by STEEL-API is equipped with C++ *Concepts*-based (see Section A.2) predicates that impose restrictions for the *instantiable* template parameters.

In particular, previous commands for executor deployment return references (`&`) to executor objects internally managed by the STEEL runtime. A task can only be mapped to an executor if that executor type satisfies the compile-time predicate specified in the `requires` that precedes a task `kernel` definition. This predicate can be based on any characteristic related to the processor architecture encompassed by the executor or on any information referring to specific third-party kernel libraries. Specific examples for this cases will be exposed in Section 4.2.7.

Specifically, when the compiler parses an instruction corresponding to a task `app::impl::X` being dispatched to an executor object associated to a processor identifier `sys::processor ::ProcessorId`, the predicate defined in Listing 4.35 is evaluated to check the task-executor compatibility. Note that `::value` is a `constexpr` boolean value inherited either from `std::true_type` or `std::false_type`, which correspond to base classes of existing or non-existing `app::kernel` definitions, respectively (See Listing 4.7).

In the context of an executor tree, the mentioned application-to-processor compatibility is transferred to the associated bottom executor, and transitively toward higher-level executors upstream in the executor tree. Ultimately, as all the user-defined and system identifiers for processors are compile-time variables, the previous constraints are translated into a final constraint that verifies whether the main implementation that represents the application or *single task* (ST) is compatible with the entry-point executor (SE) and the full executor tree.

4.2.6.2 Compile-time legal task construction

The `make_task` function is a free function endowed with C++ concept-based semantics to ensure, at compile-time, that any call to it is legal according to the user definitions in the application traits. Lines 9 to 17 of Listing 4.36 define a predicate `equal_impl_args_v`, built on top of other three clauses (lines 11, 13-14 and 17) which ensures that a task construction is legal with respect to the number of arguments passed in the call and the number of elements defined in `arg_t` and `arg_kind_t` types belonging to the traits of task `X` (See Listing 4.5).

4.2.6.3 Exception-safety at run-time

STEEL runtime uses standard C++ exceptions [138] to capture situations in which the execution cannot continue due to an error happening at run-time. Moreover, any exception thrown from the user code –either from kernel definitions or copy / cast functions of custom data structures–, is captured by the STEEL runtime and then retrieved to the user before shutting down safely.

Listing 4.36: Compile-time verification for legal use of `make_task`.

```
namespace steel {
/* Declaration of make_task function. */
template <app::impl Impl, typename... Args>
auto make_task(Args& ...);

namespace detail {
/* Internal compile-time predicate to assert legal make_task calls. */
template <app::impl Impl, typename... Args>
constexpr bool equal_impl_args_v =
  /* 1st condition: arg_t must be a tuple. */
  util::is_instantiation_of_v<std::tuple, app::traits<Impl>::arg_t> &&
  /* 2nd condition: # of elements of arg_kind_t and arg_t must match. */
  app::traits<Impl>::arg_kind_t::size() ==
    std::tuple_size_v<app::traits<Impl>::arg_t> &&
  /* 3rd condition: # of arguments passed and # of elements of arg_kind_t
  must match. */
  app::traits<Impl>::arg_kind_t::size() == sizeof...(Args);
} /* namespace detail */
} /* namespace steel */
```

4.2.6.4 Executor lifetime

Once the user requests an executor deployment via `executor::deploy` calls, the STEEL runtime internally builds the executor object and returns a reference (`&`) to it. With this design, the lifetime of executor objects is neither restricted to a scope {...} nor to deployment ordering, but managed internally to relieve the user from an unnecessary complexity. Note that complex hierarchies of executors can be built, and each executor can manage its own set of threads that may be actively running a task or not. Also, the fact that all executors are asynchronous makes the executor callers –in general– unaware of whether a task is running, has finished, or has thrown an exception. Designing the STEEL runtime to manage executor objects guarantees a proper and transparent release of resources, thread joining, data cleanup, executor destruction, and exception handling.

4.2.6.5 Future contract-based support

C++23 ISO specification will probably enable the use of *Contracts* [57, 125], which are an idiomatic feature that ensures run-time correctness and facilitates static verification of programs. STEEL-RT will integrate this feature to strengthen the user-side API exposed to STEEL-RT. For example, the STEEL-RT internal `is_allocatable` predicate (see Listing 4.20), among others, can be used to define the STEEL-RT internal contracts to ensure that the user has provided correct function definitions in `data_traits` (Listing 4.19).

4.2.7 Cases for user-defined relaxations

This section exposes specific instances of the general definitions presented in Listings 4.9, 4.10, 4.11 and 4.12. The following examples correspond to specific user-defined instantiations of the template class `kernel`, which are themselves constrained by C++ *Concepts* [61] and STEEL-API-internal `constraint` template (which serves as a proxy template-type to evaluate the existence of platform-dependent features defined at install-time). The user must provide a type named `expand` within the scope of every definition of a `kernel` class, as a specialization of any of the following template types `just`, `fix`, `relax` and `fix_relax`. Thus, the definition of `expand` type is interpreted in the compilation process to deduce the size of the action space required for scheduling the kernel execution. Specifically, these auxiliary template types are parametrized according Listing 4.37: `sched::just` (which restricts the execution to `ProcessorId` results into a singleton action space) `sched::fix` (for a library- or architecture-dependent parame-

Listing 4.37: Auxiliary template types for user-defined `expand` type.

```
1  namespace steel::sched {
2
3  /* No options expanded, just restrict execution to a processor. */
4  template <sys::processor ProcessorId>
5  struct just;
6
7  /* 'expand' option is a library-dependent value. */
8  template <support::ext_library ExtLib>
9  struct fix;
10
11 /* 'expand' options are the cross product of values in user-defined sequences. */
12 template <typename... SequenceT>
13 struct relax;
14
15 /* 'expand' options are library-dependent values combined with
16   user-defined sequence values. */
17 template <support::ext_library ExtLib, sys::processor ProcessorId, typename... SequenceT>
18 struct fix_relax;
19
20 } /* namespace steel::sched */
```

ter to be required, `sched::relax` (to expand the action space from a user-defined sequence) and `sched::fix_relax` (to expand the action space from either a library- or architecture-dependent and a user-defined sequence.

4.2.7.1 Threading relaxation and external libraries

This section exposes how to express variable-thread scheduling capabilities in terms of two backends: OpenMP and CUDA. The existence of an SMP and NVIDIA GPUs target platform and the availability of the corresponding OpenMP and CUDA libraries in the system enable the definition of internal enumeration identifiers `support::ext_library::cpu_threads` and `support::ext_library::cuda_threads` at install-time.

Listing 4.38 shows the required definitions to enable variable-thread scheduling for a task with identifier `app::impl::X` running with OpenMP. The template type `sched::relax` is specialized with the user-defined SMP partitions (`core_partitions` type) that the task is allowed to occupy (see line 9 of Listing 4.38). In this case, when ready to be run, the kernel is allowed to run on all the SMP (1/1), on just half of it (1/2), or on a quarter (1/4) of the total number of cores. As other tasks could be run simultaneously in the same SMP, STEEL-RT internally manages the allocation of tasks to avoid core over-subscription via waiting for resource acquisition and thread binding only to idle cores.

Similarly, Listing 4.39 shows the definitions to relax the CUDA-specific *number of thread blocks* and *number of threads per block* that parametrize the execution of a CUDA kernel (lines 9 and 10) [115]. Internally, STEEL-RT runs this function from a thread handled within a bottom executor assigned to a NVIDIA GPU, being this thread previously attached to a CUDA Stream at program initialization.

Several threads (and streams) will be in general associated to the same GPU, so multiple different CUDA kernels could be running concurrently at a given time on the same device. STEEL-RT tracks which and how many kernels are running at a given time in the GPU and, equivalently to the OpenMP case, a thread assignment decision will be based on the kernel characteristics –e.g. granularity–, and the current state of the GPU. Contrary to the previous OpenMP case, how the physical GPU resources are assigned to kernels is entirely resolved by the CUDA runtime.

Listing 4.38: Relaxing number of threads for a OpenMP kernel.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires /* Kernel definition if ProcessorId is a multicore cpu. */
    constraint<ProcessorId>::support_device == support::device::cpu &&
    constraint<ProcessorId>::parallelism > 1
class kernel<impl::X, ProcessorId> : public std::true_type {
private:
  /* Define set of allowed core partitions: 1/1, 1/2 and 1/4. */
  using core_partitions = std::index_sequence<1, 2, 4>;

public:
  /* Mandatory type definition. */
  using expand =
      sched::fix_relax<support::ext_library::cpu_threads, ProcessorId, core_partitions>;

  /* Kernel definition.
    Allowed 'nThreads' values deduced from core_partitions and ProcessorId parallelism.
    [...] = Arguments of arg_t tuple. */
  static void run(unsigned nThreads, [...]) {
    omp_set_num_threads(nThreads);
    /* OpenMP kernel body. */
  }
};
} /* namespace steel::app */
```

Listing 4.39: Relaxing number of threads for a CUDA kernel.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires
  constraint<ProcessorId>::support_device == support::device::gpu &&
  constraint<ProcessorId>::has_ext_library<support::ext_library::cuda_threads>
class kernel<impl::X, ProcessorId> : public std::true_type {
private:
  /* Define set of allowed number of blocks */
  using n_blocks = std::index_sequence<1, 16, 64>;
  using threads_per_block = std::index_sequence<64, 256, 1024>;

public:
  /* Mandatory type definition. */
  using expand = sched::fix_relax
      <support::ext_library::cuda_threads, ProcessorId, n_blocks, threads_per_block>;

  /* Kernel definition. 'param' type is deduced from 'expanded' definition.
  [...] = Arguments of arg_t tuple expanded. */
  static void run(unsigned nBlocks, unsigned nThreadsPerBlock, [...]) {

    /* Set [kernelArgs] from [...]. */

    /* CUDA kernel call. */
    my_kernel<<<nBlocks, nThreadsPerBlock>>>([kernelArgs]);

    /* Calling thread must block until my_kernel is finished. */
    cudaStreamSynchronize(cudaStreamPerThread);
  }
};
} /* namespace steel::app */
```

Listing 4.40: Example calling a kernel library requiring a built-in type.

```
1  namespace steel::app {
2  template <sys::processor ProcessorId>
3  requires
4    constraint<ProcessorId>::has_ext_library<support::ext_library::cusolver>
5  class kernel<impl::X, ProcessorId> : public std::true_type {
6  public:
7    /* Mandatory type definition from built-in. */
8    using expand = sched::fix<support::ext_library::cusolver>;
9
10   /* Kernel definition. 'param' type is deduced from 'expand' definition.
11   [...] = Arguments of arg_t tuple. */
12   static void run(auto param, [...]) {
13     /* Get custom handle set by STEEL scheduler.  */
14     auto cusolverHandle = std::get<cusolverDnHandle_t>(param);
15
16     /* Get handle for local memory allocation. */
17     auto localAllocCallback = std::get<local_allocator_callback_t>(param);
18
19     /* Allocate temporary buffer. Set [requiredMemory] from [...]. */
20     auto tempBuffer = localAllocCallback([requiredMemory]);
21
22     /* Set [kernelArgs] from [...] and tempBuffer. */
23
24     /* A generic call [cusolverCall].  */
25     cusolverCall(cusolverHandle, [kernelArgs]);
26
27     /* Calling thread must block until my_kernel is finished. */
28     cudaStreamSynchronize(cudaStreamPerThread);
29
30   } /* End of scope. Memory in tempBuffer is released. */
31 };
32 } /* namespace steel::app */
```

Regarding the use of some third-party CUDA-based kernel libraries, kernel calls may not require explicit threading decisions, so there is no relaxation that the user can expose to the STEEL-RT. However, these kernel calls might require opaque types to be passed to kernel calls and / or require temporary memory allocation. This may be the case for some calls to popular cuBLAS [113] or cuSOLVER [114] libraries. Listing 4.40 shows an example for a cuSOLVER kernel call that requires a specific opaque handle type `cusolverDnHandle_t` (see [114]) and a prior memory allocation (line 19). The mandatory type `expand` is defined from a `sched::fix` template type specialized for `support::ext_library::cusolver` identifier, set at STEEL install-time if the cuSOLVER library is detected in the system.

4.2.7.2 Vectorization relaxation

Listing 4.41 exposes a case in which the user defines the available options in terms of possible vector extensions featured by the architecture. For example, the definition of the `expand` type in line 11 could be internally set to a sequence of options such as {`sse, avx2, avx512`} (in the case of a Intel Skylake-X architecture [75]), associated with different extensions for instruction vectorization.

Similarly, in Listing 4.42 the user exposes a kernel that could potentially be running in two different modes with respect to the use of Tensor cores in modern NVIDIA Volta GPUs.

Listing 4.41: Relaxing kernel execution via vector extension.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires
  constraint<ProcessorId>::support_device == support::device::cpu &&
  constraint<ProcessorId>::vector_ext::size() > 0
class kernel<impl::X, ProcessorId> : public std::true_type {
public:
  /* User defines vectorization options from supported extensions
  provided in the processor encompassed by the executor. */
  using expand = sched::relax<constraint<ProcessorId>::vector_ext>;

  /* Kernel definition. 'vExt' refers to a particular vector extension.
  [...] = Arguments of arg_t tuple. */
  static void run(auto vExt, [...]) {
    /* Kernel body. Divert execution depending on vExt allowed values. */
  }
};
} /* namespace steel::app */
```

Listing 4.42: Relaxing a cuBLAS kernel execution via CUDA tensor cores.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires
  constraint<ProcessorId>::tensor_ext::size() > 0 &&
  constraint<ProcessorId>::has_ext_library<support::ext_library::cublas>
class kernel<impl::X, ProcessorId> : public std::true_type {
private:
  /* User enables options from supported extensions
  provided in the processor encompassed by the executor. */
public:
  /* Mandatory type definition. */
  using expand = sched::fix_relax
      <support::ext_lib::cublas, ProcessorId, constraint<ProcessorId>::tensor_ext>;

  /* Kernel definition. 'param' value refers to whether use or not tensor cores.
  [...] = Arguments of arg_t tuple. */
  static void run(auto param, [...]) {
    cublasHandle_t cbHandle = std::get<cublasHandle_t>(param);
    using tensor_opt_t = typename constraint<ProcessorId>::tensor_ext::value_type;
    auto useTensorCores = std::get<tensor_opt_t>(param);
    /* Kernel body. Divert execution depending on useTensorCores. */
  }
};
} /* namespace steel::app */
```

Listing 4.43: Kernel for task partitioning.

```
1  namespace steel::app {
2  /* Partial specialization */
3  template <sys::processor ProcessorId>
4  requires ProcessorId == sys::processor::unfolder
5  class kernel<impl::X, ProcessorId> : public std::true_type {
6  private:
7    /* User enables allowed partition granularities. */
8    using my_opts = std::index_sequence<2, 4>;
9
10   /* Data argument type. */
11   using data_t = util::tag_t<data::impl::DS, traits<data::impl::DS>::descriptor_t>;
12
13   /* Return size parameters of 2d dimensional data. */
14   int get_size_x(const data_t& dh) {
15     return /* Size parameter x in dh descriptor. */; }
16   int get_size_y(const data_t& dh) {
17     return /* Size parameter y in dh descriptor. */; }
18 public:
19   /* Mandatory type definition. */
20   using expand = sched::relax<my_opts>;
21
22   /* Kernel definition.
23   [...] = Arguments of arg_t tuple expanded. */
24   static void run(auto& delegate, unsigned nPartitions, data_t& dataArg) {
25     /* Create parent data guard from which children data is with 'copy' visibility.*/
26     auto parentGuard = dep::handle<DS, dep::kind::inout, dep::view::copy>();
27
28     /* Number and size of partitions is set from 'nPartitions' (which is either 2 or 4)*/
29     auto grainSizeDimX = get_size_x(dataArgs) / nPartitions;
30     auto grainSizeDimY = get_size_y(dataArgs) / nPartitions;
31     for(auto j = 0u; j < nPartitions; ++j) {
32       for(auto i = 0u; i < nPartitions; ++i) {
33         /* Create children tasks on partitioned data
34         and forward them to 'delegate' executor. */
35         delegate( make_task<impl::X>( parentGuard.get(
36             {i*grainSizeDimX, j*grainSizeDimY, grainSizeDimX, grainSizeDimY}) ) );
37       }
38     }
39   } /* End of scope. Thread blocks until all children tasks have finished. */
40 };
41 } /* namespace steel::app */
```

4.2.7.3 Task partitioning

Listing 4.43 exposes a kernel instance of a task `app::impl::X` for data partitioning, that once running in the context of an unfolder executor (see constraint in line 4) generates and delegates children tasks to a lower-level executor (referenced by `delegate` in lines 24 and 35). It is assumed, for the sake of simplicity, that the kernel receives one argument (e.g. with a data-identifier `app::data::impl::DS` in *read-write* mode.)

In order for the `parentGuard.get()` instruction to be legal for the compiler, the user must define a `key_to_descriptor` function in the traits associated to `app::data::impl::DS`, as exposed in Section 4.2.4.3 and Listing 4.27. In particular, regarding lines 35-36 of Listing 4.43 in which two integers are passed –representing the shift from the parent origin address and the grain size of the data partitions–, an array of integers could be used as the *key* (see Listing 4.44).

Chapter 6 will provide specific examples exposing how other access patterns can be built.

4.2.7.4 Task reimplementation

Similarly, Listing 4.45 explains how the user can express possible reimplementations –`app::impl::Y` and `app::impl::Z`– of a parent task `app::impl::X`. Line 9 defines the

Listing 4.44: User definitions for subregion access.

```
1  namespace steel::app::data {
2  class traits<impl::DS> {
3  public:
4    /* Type definition to characterize a 2D data tile. */
5    using subdata_descriptor_t = std::array<int, 4>;
6
7    /* Function to build data descriptors of sub regions of a parent descriptor. */
8    static descriptor_t key_to_descriptor(const descriptor_t& parentDesc,
9        const subdata_descriptor_t& child2DTileDescriptor) {
10     /* Return a descriptor referring to the 2d tile. */
11   }
12 };
13 } /* namespace steel::app::data */
```

sequence of options, representing the first option to *not-reimplement*. New data guards need to be created in any case, and in the particular cases in which STEEL-RT decides to perform a reimplementation, some reimplementation-specific data transformations might also be needed (lines 22 and 26).

4.2.7.5 Precision casting

Listing 4.46 assumes, for the sake of simplicity, that the task `app::impl::X` kernel receives one argument (e.g. with data-identifier `app::data::impl::DS` and *read-write* mode) in double precision. The options defined in line 9 refer to not performing the cast (`app::cast::identity`) or cast data to single precision (`app::cast::to_singlef`). In any case, a new data guard needs to be created (lines 18 and 24) to be forwarded to the new task instance. In the case of casting, the guard needs to be created with `dep::viewcopy` visibility, and a *data subguard* that will represent the casted-data to single precision is returned by `get(ptype::f)` (line 26). In Section 5.2.5 will expose how data partitions, together with precision casting can be expressed within the same kernel.

4.2.7.6 Other feasible combinations

At this point, it is clear that the execution possibilities can be either expressed by the user, defined by the architecture details of the encompassed devices, or defined by internal built-in types provided from third party libraries. Also, it became clear that some task featurizations will lie under the scope of *bottom executors* (e.g., threading, vectorization...) while others lie into higher-level unfolders (e.g., partitioning, reimplementation, precision casting...). In particular, thanks to sequence composition exposed in Listing 4.10, the user is also free to compose different features associated with bottom-execution contexts and to unfolder-executor contexts. With this regard, Chapter 6 will expose some particular examples illustrating how this featurization composition can be done in both contexts.

4.3 Summary

To put some of the concepts exposed in previous sections into a wider and a contextual scope, Section 4.3.1 briefly exposes some of the main characteristics of STEEL-PM, put into context in Section 4.3.2 with regard to some of programming model approaches presented in Chapter 1.

Listing 4.45: Allow different reimplementations for a task.

```
namespace steel::app {

/* Partial specialization */
template <sys::processor ProcessorId>
requires Processor == sys::processor::unfolder
class kernel<impl::X, ProcessorId> : public std::true_type {
public:
  /* User-defined reimplementations. */
  using allowed_reimplements = util::impl_sequence<impl::X, impl::Y, impl::Z>;

  /* Mandatory type definition. */
  using expand = sched::relax<allowed_reimplements>;

  /* Kernel definition.
  [...] = Arguments of arg_t tuple expanded. */
  static void run(auto& delegate, app::impl chosenImpl, ...) {
    if (chosenImpl == impl::X) {
      /* Reimplementation not requested. Relaunch same task. */
      delegate( make_task<impl::X>([...]) );
    } else if (chosenImpl == impl::Y) {
      /* Reimplementation to Y requested. Relaunch same task. */
      /* Create new guards [newGuardsY...] from [...] */
      delegate( make_task<impl::Y>([newGuardsY...]) );
    } else if (chosenImpl == impl::Z) {
      /* Reimplementation to Z requested. Relaunch same task. */
      /* Create new guards [newGuardsZ...] from [...] */
      delegate( make_task<impl::Z>([newGuardsZ...]) );
    }
  }
} /* namespace steel::app */
```

Listing 4.46: Example for floating point down-casting.

```
namespace steel::app {
/* Partial specialization */
template <sys::processor ProcessorId>
requires ProcessorId == sys::processor::abstract
class kernel<impl::X, ProcessorId> : public std::true_type {
private:
  /* Assuming there are two options :
  no casting or cast to single precision floating point. */
  using my_opts = util::cast_sequence<cast::identity, cast::to_singlef>;
public:
  /* Mandatory type definition. */
  using expanded = sched::relax<my_opts>;

  /* Kernel definition. */
  static void run(auto& delegate, cast castKind, data_t& inoutArg) {
    if (castKind == cast::identity) {
      /* Casting requested. Guard again and relaunch the same task. */
      auto newGuard = dep::handle
          <DS, dep::kind::inout, dep::view::map>(delegate, inoutArg);
      delegate( make_task<impl::X>(newGuard);
    } else if (castKind == cast::to_singlef) {
      /* Casting requested. Create new guard, represent it
      as [newGuards...] from [...] */
      auto newGuard = dep::handle<DS, dep::kind::inout, dep::view::copy>
          (delegate, inoutArg);
      delegate( make_task<impl::X>(newGuard.get(ptype::f)) );
    }
  }
};
} /* namespace steel::app */
```

4.3.1 STEEL-PM main characteristics

The ultimate aim of STEEL-PM is to lower the programmability barrier of HPC application development without sacrificing performance. With this goal in mind, user-defined relaxations are essential. However, there are some advantages with regard to the programming styles characteristic of STEEL-PM that are worth mentioning.

4.3.1.1 Declarative style

Sections 4.2.2 and 4.2.4 exposed specific rules that the user must follow for task and data definitions. With regard to *task featurizations*, the corresponding definitions follow a *declarative style*, in the sense that, to some extent, the user no longer commands *what* to compute, but exposes different options that define *how* a task can be computed. Consequently, the burden of non-portable application- and system-dependent performance tuning is delegated from the user to the runtime. Imperative style is however applied at a lower level in kernel definitions, in an unfolder executor, when tasks are created and delegated, and in kernels within bottom executor.

4.3.1.2 Inversion of Control (IoC)

One of the characteristics of STEEL-API is that it is partially based on the IoC principle, by which the framework calls the application code instead of the more traditional opposite approach. This is a clear pattern in the interface explained in Sections 4.2.2 and 4.2.4: the user must define types and functions –related to kernel computations, task partitions and data structures–, with specific naming so that they are visible and called from compilation and runtime side.

The IoC principle is also known as the *Hollywood Principle: "Don't call us, we'll call you"*, and although it was first introduced in the context of Object-Oriented programming [107, 79] as a mechanism to enhance modularity and extensibility of classes, in this context it is not applied to user-defined classes (hence not from an Object-Oriented approach), but rather to pure functions belonging to template interfaces `app::kernel` and `app::data::traits`. Specifically, user-defined `static void run` functions (callable within an executor context), are preprocessed at compile-time, so that they can internally treated as *continuations*, following a *Continuation-Passing-Style* [141] by STEEL-RT.

4.3.1.3 Minimal callable interface

In relation to the set *callable* STEEL-API functions –i.e., functions meant to be called from user code at run-time–, unlike other large interfaces of some programming models [12, 84, 90], it is worth noting that it is designed to be very small.

At a glance, the basic functionalities consist on task creation (`make_task`), executor deployment (`executor::deploy`) and run (*call* operator `()` of deployed executor references), task unfolding and kernel executions (`static void run` within `app::impl::kernel` definitions), data wrapping for partitioning (`executor::handle`, and `get`), and custom allocation (`executor::raw_allocate`).

4.3.1.4 Full sequential style

Moreover, note that there is no need for using any concurrency-related special syntax at any point. Deployed executors are essentially thread-safe interfaces that only accept objects returned by *proper* tasks –i.e., objects returned from a `make_task` function–. Otherwise, the compilation will fail. Also, the compilation will only succeed if for all task-to-executor dispatching happening in a program –i.e., in `main` context and any delegation within an unfolder executor scope–, there ultimately exist a compatible executor anywhere in the executor tree headed by the executor to which the task is delegated.

Executor deployment and task creation are thread-safe although in principle there is no reason to call them concurrently. Also, `executor::handle` is not thread-safe (neither data-related functions `executor::handle` and `get`). However this is not a problem since `static void run` functions are called within a single thread managed by STEEL-RT.

4.3.1.5 Compilation and portability

As specified in Figure 4.2, STEEL-PM applications strictly adhere to standard C++, so only a C++17 compliant compiler is needed (e.g., GNU GCC [56] or LLVM Clang [97]). In order to exploit specific platform characteristics (e.g., GPUs), external dependencies such as libraries and additional compilers may also be needed. In addition, specific *support templates* may need to be instantiated by a STEEL-RT maintainer so that at install-time the platform characteristics and external software components can be properly detected.

As mentioned in [160], this *compiler-free*[1] approach offers some benefits over the *library-* or *compiler-dependent* approaches. Generic programming mechanisms based on C++ template metaprogramming and template specializations provide sufficient expressive power to construct high-level compositions exposing greater compilation optimization possibilities such as inlining or evaluation of *constant expressions* which can result into high-performance binaries. Additionally, better interoperability with third-party programming models and lower development effort are other benefits derived from simply relying on already well-established, portable and robust compilers.

4.3.1.6 Purity of user-defined functions

One of the intents of STEEL-PM is to expose a parallel programming framework in which the user does not have to deal with any issue related to concurrency, for example dealing with race-conditions, deadlocks, etc. As exposed in Section 1.2.1.1, function purity is a simple and thread-safe *per se* characteristic that is recommended in all user-defined functions, as a way to provide safety –they are callable from STEEL-RT at any time and possibly concurrently–.

Since there is no mechanism in C++ to assert no-side-effect computation of these user-defined functions, their purity is encouraged but not forced. For example, although user-defined kernels running bottom executors are meant to represent computational kernels, in practice there is no constraint to express any kind of computation –effectful or not– within them.

Consequently, STEEL-PM leaves room for *controlability*, so that developers can for example spawn a set of threads within an unfolder executor kernel to speed-up the deployment of tasks to the lower level executor. In this case, as guard objects returned by `executor::handle` are not thread-safe, the user would need to provide protection mechanisms to ensure that the corresponding `.get` calls are safe. Note however that there is no requirement whatsoever to perform any concurrency-related operation within unfolder executor contexts, and internal STEEL-RT implementation is featured with mechanisms to maintain a proper balance between production rate and consumption rate of tasks.

As a final note regarding Section 4.2.6, the only situation in which the user is encouraged to sacrifice purity is in the case of runtime exceptions happening in kernel functions.

4.3.2 Related work

As exposed in Section 1.2.2, many frameworks and runtimes have been implemented aiming at increasing programmability and portability without sacrificing performance, and in the majority of these cases, modern C++ plays a central role. In the following, it is first exposed a set of task-

[1] *Compiler-free* characteristic refers to the fact that unlike other programming models, STEEL-PM does not expose any intrinsic programming language that would demand a custom compiler.

oriented, general purpose and pattern-oriented programming frameworks, based on modern C++ constructs, that exhibit some similarities with the STEEL-PM presented in this chapter.

Interestingly, task-based and pattern-based frameworks have been greatly influenced from functional and declarative programming paradigms; for example, the functional composability between either patterns and tasks is key to chain computations in a correct way. Also, it is commonly assumed that either tasks and patterns are essentialy *pure* or *side effect-free* functions, so that parallelism can be expressed more naturally.

Pattern-based models are however more constrained than task-based ones, in the sense that not all applications can be expressed in the form of patterns. Some algorithms that compose the target application may not expose a representation transformable to a simple pattern. On the contrary, task-based models can represent virtually any application in the form of DAG- or stream-like computations. Also, in task-based models it is common that dependencies between tasks cannot be computed at compile-time, as the flow of task generation may depend on the output data. However, the generality of computation models based on tasks comes with the price of a higher programmability barrier in the corresponding programming models. As in pattern-based models a minimal or no knowledge of concurrency / parallelism is required, in task-based programming models it is common to require some form of concurrency-dependent language constructs to explicitly do coarse-grain synchronization or to manually compose computations for fine-grain synchronization –e.g., employing some form of CPS-style and *futurization* based techniques–.

4.3.2.1 Task-oriented and general purpose models

Several interesting projects related to system software focusing on HPC runtimes and frameworks have been conducted since the first standarization of modern C++ in 2011. In addition, the apeal of task-parallelism has guided the philosophy of several of them.

HPX [84, 85] is an open-source multi-platform implementation of the *ParalleX* execution model. It is written using modern C++ and Boost [133] libraries and aims at targeting modern heterogeneous and distributed systems on which applications of any scale can run. Also, HPX exposes a rather big API that follows latest C++ standard to date and it already implements several features proposed for future standarization. Internally, HPX manages its own set of internal lightweight threads, and implements mechanisms for automatic load balancing. From the programming perspective, HPX encourages a of programming style oriented to task-based models and asynchronous data-flow chaining of computations employing advanced generic- and functional-based patterns based on coroutines and asynchronous composition through *futures* and *continuations*. HPX has also demonstrated that task-asynchronous executions are able to attain excellent scalability properties in large peta-scale simulations [68].

SYCL programming model aims at simplifying HPC application development by providing a modern C++ API from which the capabilities of OpenCL-programmable processing devices can be exploited. In addition, SYCL device compiler follows a *single-source* paradigm to enhance compilation portability. ComputeCpp [102] and trySYCL [50] are respectively proprietary and open source implementations of SYCL. A computational kernel implemented in a SYCL program can be instantiated as machine code for different target architectures by means of LLVM *Intermediate Representation* [97] mechanisms and SPIR / SPIR-V cross APIs [90]. This approach shares some similarities with the STSE model exposed in this thesis: in the STEEL-PM implementation, the full application and the *application-* and *system-dependent* runtime is instantiated for multiple target platforms by means of template specializations. However, recall that computational kernels in STEEL-PM are in general assumed to be external components, that can also be orthogonally appended (and also subject to *relaxation*) to STEEL-RT, as it was exposed in Section 4.1.2.

In a similar line of SYCL, PACXX [65] *Programming Accelerators with C++* is a programming model that uses C++14 language features together with a Clang-based / LLVM-IR [97] compiler and a runtime system for automatic memory management. PACXX aims at providing a unified programming environment that eases development of kernels meant to be run in CUDA

or OpenCL backends targeting accelerators programmable with intermediate representation languages NVVM-IR [116] or SPIR [89], also following the *single-source* paradigm.

Kokkos [52] is a C++ library focusing on performance portability of computational kernels on a wide set of manycore architectures. To achieve this, Kokkos looks at performance problems in kernel execution operating on multidimensional arrays, and proposes an interface to abstract fine-grain data parallelism and memory access patterns, so the data layout can be modified depending on the optimal pattern associated to a specific architecture. Similar to STEEL, Kokkos uses *memory spaces* and *execution spaces* concepts to abstract *where* data is residing and *how* a kernel is parallelized, respectively. Also, similar to STEEL, execution and memory spaces are paired at compile-time, so threads associated to an execution space are prevented from accessing outer memory spaces without the need of runtime checks. Kokkos has shown excellent performance results without sacrificing portability for a set of computational kernels running on CPU, GPU and Xeon Phi architectures, and general guidelines based on Kokkos were exposed in order to migrate legacy codes to more performance-portable ones.

Raja [74, 91] proposes a performance-portability model motivated by performance-portability and programmability issues appearing in a set of multiphysics applications. Its goal is to provide a way to perform high-level performance optimization and –in a very similar line of STEEL-PM– encapsulating architecture-specific and other programming model-specific constructs. One of the main concerns of Raja is the code disruption caused by architecture-specific optimizations and the main starting point of the proposed model is to abstract the *loop*. In the target applications, the computation is characterized by a set of nested loops, and they propose a way to make code adaptations to newer architectures and models less disruptive and with minimal impact. They propose to abstract the inner loop, where most of the computational workload is performed, and propose a set of language constructs on top of modern C++ in which loop computations, execution policies, and data attributes can be generalized and specified.

UPC++ [160] is a Partitioned Global Address Space (PGAS) extension for C++. Specifically, it is an object-oriented PGAS model meant to extend functionalities of UPC [26] by providing remote asynchronous function execution, multidimensional arrays and easier interoperability with other programming models such as MPI, OpenMP and CUDA. UPC++ has demonstrated that the level of performance and scalability of UPC can be achieved in large distributed systems, thus opening the door of efficient PGAS programming for C++ applications. Equivalently to STEEL-PM, UPC++ uses a *compiler-free* approach, by which rich and generic programming patterns are used to express PGAS idioms at compile-time.

4.3.2.2 Pattern-oriented approaches

Programming frameworks based on patterns / skeletons rely on the importance of composability, readability and genericity oriented to architecture-independent code, toward achieving high-performance and scalable parallel programs [123, 108]. In addition, C++ template metaprogramming techniques provide language constructs aimed at genericity and composability rooted on a strongly-typed functional language, able to express generic parallel algorithms [22].

In particular, parallelized versions of C++ Standard Template Library algorithms have been proposed [77] and are already implemented as part of some programming models (HPX, SYCL) and by C++ compiler vendors (Intel and Microsoft). STL comprises a large set of algorithms and patterns of general use applicable to STL containers.

The Standard Template Adaptive Parallel Library (STAPL) [24] is a general purpose library in C++ to enable SPMD and nested parallelism in shared memory and distributed architectures, also oriented toward genericity and productivity / programmability. STAPL provides execution and communication abstractions and has reported good scalability and performance portability of a set of parallel algorithms applied to distributed data structures.

RaftLib [18] is another template library that targets parallel stream-like computations under high productivity programming patterns. RaftLib is equipped by internal mechanisms for au-

tomatic parallelization to extract pipeline and task parallelism at run-time. Also, it provides a dynamic monitoring system in order to improve execution performance at run-time.

Similarly, Generic Reusable Parallel Pattern Interface (GRPPI) [45] is a compile-time library targeting stream- and data-like patterns that employs advanced C++ metaprogramming techniques to chain computations with minimal runtime overhead. GRPPI is also composable with several backends (native C++ threads, OpenMP and TBB) and has demonstrated clear benefits in terms usability / programmability (lines of code) with respect to language constructs of the backends, and without incurring into additional performance overhead.

With regard to proprietary pattern-based models, CUDA Thrust [19] and Intel Threading Building Blocks [150] target pattern-based applications optimized for their architectures and relying on modern C++ programs. Intel TBB also supports (and encourages) task parallelism.

Other approaches like Kanga [92], SkePU [53], SkeTo [143], Muesli [37], FastFlow [7], Delite [139] and MaLLBa [6], share the same aims targeting performance-portable pattern-based high-level abstractions via skeletons, but differ in the target applications, target data structures, compute backends or internal runtime characteristics (queuing management, lock-based / lock-free synchronization, etc.).

In [44], PARSEC benchmarks were programmed in the form of patterns to build the P3ARSEC benchmark suite. In addition, a comprehensive study is done regarding programmability and performance measurements for these benchmarks on three shared memory parallel architectures of a set of *(1)* pattern-based models (SkePU, FastFlow and TBB), *(2)* task-based (OmpSs) and *(3)* three backends (sequential execution, using pthreads, and OpenMP). In summary, for the tested cases it has shown that pattern-based frameworks need in general less lines of code with excellent performance results.

STEEL-PM exposes several similarities with pattern-based programming frameworks while still maintaining the generality of the task-based computational models. The first (and most evident) one is the problem statement –the need for generic, composable and high-level parallel programming interfaces– and the solution approach –the use of advanced programming mechanisms consisting on C++ templates and metaprogramming–. The second common feature is that STEEL executors can be regarded as a kind of skeletons: in essence, they are generic, computation-parametrizable, and composable containers that channel computations with a clear *top-bottom* directionality; being the top layer the actual user-level frontend, and the bottom layer referring to the compute backends and parallel hardware.

Love deep,
work hard,
sail the sea.

Save heap,
burn the card,
be lock-free.

5 Use cases and experimental results

This chapter instantiates a subset of the interface exposed in Chapter 4, particularizing the use of STEEL-PM by means of well-known *data*- and *compute*-related use cases. A set of real codes following STEEL-API are provided together with real execution traces on a high-end heterogeneous platform with 40-core CPU parallelism, two GPUs (MACHINE2, see B.2) and four different memory spaces (one non-volatile –NVME– in which the original data is residing and three local RAM spaces, one for CPU-bound RAM –CPU-RAM– and one for each GPU –GPU-RAM–).

The goal of this chapter is not to expose detailed performance analysis, but to demonstrate a set of STEEL-RT features from specific examples in the simplest possible way. As the tested cases were chosen from simplicity and clarity considerations, eventual performance improvements derived from user-defined relaxations are not treated in this chapter. Indeed, the potential benefits of user-defined relaxations would demand a more detailed and deeper study targeting more complex and challenging applications.

5.1 Driving examples

2D data buffers and two linear algebra operations are used as driving examples to expose some STEEL functionalities. As mentioned in Sections 4.2.2 and 4.2.4, user-defined tasks and data structures must be defined in `app` and `app::data` namespaces, respectively, being `app` within the `steel` namespace.

5.1.1 2D buffers

User-defined data structures are specified in terms of `traits` in `app::data`. Current STEEL-RT implementation supports multi-dimensional arrays, implemented as specific template instantiations of `data::traits` for identifiers `app::data::impl::buffer_1d`, `::buffer_2d`, `::buffer_3d`, and so on, for which corresponding *handle_t* template type instantiated by them (recall Listing 4.22) `dim_1d_t`, `dim_2d_t` and `dim_3d_t` exist. The following examples illustrate different features for the 2D particular case (corresponding to `data::impl::buffer_2d` identifier), exposing different *accessibility* modes (to access data subregions), and different *casting* possibilities that can be considered orthogonally with several API- and runtime-specific features exposed in the previous chapter, such as data visibility modes –`dep::view::copy` and `dep::view::map`–.

Listing 5.1: C implementation of the tiled Matrix-multiplication.

```
1  void tiled_matrix_multiply (double *A[s][s], double *B[s][s], double *C[s][s],
2      int b, int s) {
3    for (int k = 0; k < s; k++) {
4      for (int i = 0; i < s; i++) {
5        for (int j = 0; j < s; j++) {
6          matrix_multiply (A[i][k], B[k][j], C[i][j], b, b);
7        }
8      }
9    }
10 }
```

5.1.2 Linear algebra kernels

5.1.2.1 Matrix multiplication

Matrix multiplication is a well-known computational kernel fundamental in a broad set of applications. It is a common yet simple benchmark in many architectures or programming models, and it features some basic characteristics that will guide the description of STEEL features from the perspective of data and task management.

General matrix multiplication operates the matrices $A \times B + C \rightarrow C$ with respective dimensions $m \times k$, $k \times n$, $m \times n$ (Figure 5.1). It can be decomposed into smaller matrix multiplications with a *tiled* matrix multiplication algorithm (Listing 5.1). These possible decompositions are illustrated in Figures 5.2 and 5.3.

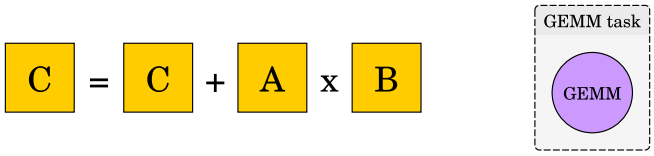


Figure 5.1: Representation of a matrix multiplication operation.

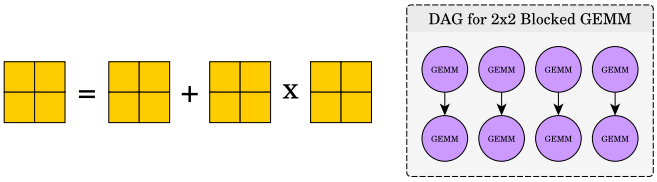


Figure 5.2: Representation of a 2×2-tiled matrix multiplication operation and associated task DAG.

Listing 5.2 implements the type characteristics of the matrix multiplication task identified with `app::impl::gemm`. Line 6 declares that the task features three arguments typed as 2-dimensional buffers, while line 8 specifies a sequence of dependency annotations in correspondence with the previous tuple.

5.1.2.2 Cholesky factorization

In addition, Cholesky factorization, which served as a driving use case in Chapter 2, will serve again as an example from which executor deployment management in Section 5.4 will be presented.

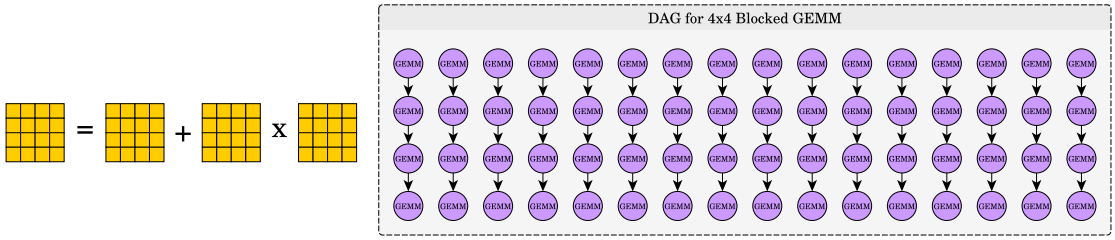


Figure 5.3: Representation of a 4×4-tiled matrix multiplication operation and associated task DAG.

Listing 5.2: Definition of type characteristics (*traits*) for the `gemm` task.

```
1  namespace steel::app {
2  template <>
3  class traits<impl::gemm> {
4  public:
5    /* Tuple of arguments. */
6    using arg_t = std::tuple<dim_2d_t, dim_2d_t, dim_2d_t>;
7    /* Associated dependency kind. */
8    using arg_kind_t = util::kind_sequence<dep::kind::in, dep::kind::in, dep::kind::inout>;
9  };
10 } /* namespace steel::app */
```

Listing 5.3 implements the type characteristics of the `app::impl::potrf` task, also in terms of type *traits*. Line 6 declares that the task features one argument: a data handle representing a 2-dimensional buffer. Line 8 specifies a (singleton) sequence of dependency annotations for the arguments in `arg_t` tuple –in this case a single read-write argument. Lines 11 and 12 present two function signatures whose bodies are omitted for the sake of brevity. These auxiliary functions are intended to be callable from user-defined kernel instantiations.

5.2 Data management. Visibility, access modes and OOC computation

Using `impl::gemm` as a driving example, this section exposes a set of use cases differentiating between how data- and compute-related operations are expressed.

Considerations about latency and bandwidth in parallel computing systems with deep and wide memory hierarchies open considerations about the *placement* of computation that impacts

Listing 5.3: Definition of type characteristics (*traits*) for the `potrf` task.

```
1  namespace steel::app {
2  template <>
3  class traits<impl::potrf>{
4  public:
5    /* Tuple of argument. */
6    using arg_t = std::tuple<dim_2d_t>;
7    /* Associated dependency kind. */
8    using arg_kind_t = util::kind_sequence<dep::kind::inout>;
9
10   /* Auxiliary functions callable from user code. */
11   static std::tuple<int, int> get_parameters(const dim_2d_t& matrix);
12   static void * get_pointer(const dim_2d_t& matrix);
13 };
14 } /* namespace steel::app */
```

Listing 5.4: *Main* entry point of a `impl::gemm` program employing *copy visibility*.

```
1  /* Run example: ./binary_name \
2                      --input-a=/mnt/nvme/ina/1024-1024-d.bin \
3                      --input-b=/mnt/nvme/inb/1024-1024-d.bin \
4                      --inoutput=/mnt/nvme/inout/1024-1024-d.bin */
5
6  #include <steel.hpp>
7  using namespace steel;
8
9  void main(int argc, char** argv) {
10
11   const auto pMap = build_parse_map(argc, argv);
12
13   auto& mainExec = executor::deploy_main();
14
15   auto inMatAGuard = dep::handle
16       <app::data::impl::buffer_2d, dep::kind::in, dep::view::copy>
17       (mainExec, pMap["--input-a"]);
18
19   auto inMatBGuard = dep::handle
20       <app::data::impl::buffer_2d, dep::kind::in, dep::view::copy>
21       (mainExec, pMap["--input-b"]);
22
23   auto inOutGuard = dep::handle
24       <app::data::impl::buffer_2d, dep::kind::inout, dep::view::copy>
25       (mainExec, pMap["--inoutput"]);
26
27   mainExec( make_task<app::impl::gemm>(inMatAGuard, inMatBGuard, inOutGuard) );
28 }
```

execution performance. Specifically, given a pair of physically separated memory spaces, one remote and the other local to a processing device, it is an open question whether the data must be *(1)* fully *copied* to the local memory space before the computation starts –following a *copy visibility* model– or *(2) mapped* to the local memory space, so the data is remotely streamed (or *pulled* from the remote to the local memory) as the execution proceeds –following a *map visibility* model–.

5.2.1 Dependency visibility

Listing 5.4 shows how to wrap a data dependency, setting the visibility as a *copy visibility* by means of the variable `dep::view::copy`. The program can be executed according to the command in lines 1-4, in which three files representing read-only and read-write data present in the NVME memory space are guarded. Traces in Figure 5.4 represent the data copy and the single kernel execution operations corresponding to Listing 5.4. The first three blocks in the top trace correspond to data copies from the the NVME memory space to local CPU-RAM, performed by the main thread. The final block in the trace corresponds to the copy of the result of the computation (illustrated as the `kernel_run` region in bottom trace) from local CPU-RAM to the original NVME memory space.

Assuming a case in which matrices are guarded via `dep::view::map`, the corresponding trace would lack the blocks in the top trace, and data would be internally transferred to local CPU-RAM on a page-fault basis during kernel execution (that is, the kernel start would not be blocked until all data is present in local CPU-RAM).

5.2.2 2D access modes

Accessing modes are ways to access to subregions of a given multidimensional data object. Listing 5.5 illustrates a *block* access mode identified with `data::access::block`, which in general is used to refer to contiguous multidimensional subregions of a given region of the same

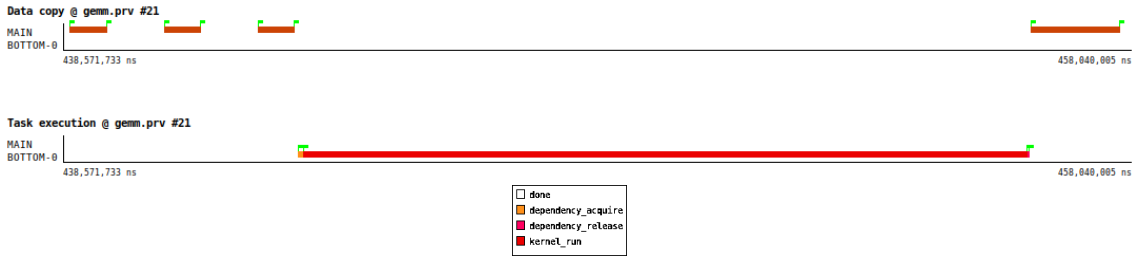


Figure 5.4: Basic execution traces illustrating data copies (top) and kernel execution (bottom). Legend only corresponds to bottom trace colors.

Listing 5.5: Example of `data::access::block` mode to partition a matrix into quadrants.

```
/* Function callable by STEEL-RT from an unfolder context. Defined inside a kernel class*/
static void run(auto& dlg, dim_2d_t& matrix) {

  /* Create acc_block_t type alias. */
  using acc_block_t =
      data::traits<data::impl::buffer_2d>::access_type<data::access::block>;

  /* Wrap matrix into a data guard. */
  auto matGuard =
      dep::handle<data::impl::buffer_2d, AnyDepKind, AnyDepView>(dlg, matrix);

  auto [nRows, nColumns] = get_parameters(matrix);
  int blockRows = nRows / 2;
  int blockColumns = nColumns / 2;

  /* Create references to guarded quadrants. */
  auto& upLeftQuadrant =
      matGuard.get(acc_block_t({0, 0, blockRows, blockColumns}));
  auto& upRightQuadrant =
      matGuard.get(acc_block_t({0, blockColumns, blockRows, blockColumns}));
  auto& downLeftQuadrant =
      matGuard.get(acc_block_t({blockRows, 0, blockRows, blockColumns}));
  auto& downRightQuadrant =
      matGuard.get(acc_block_t({blockRows, blockColumns, blockRows, blockColumns}));

  /* [...] Operate on guarded quadrants. */
  [...]
}
```

dimension. In 2D, variables of type `acc_block_t` define the 2D origin and row / column count (`acc_block_t` = *[origin row, origin column, row count, column count]*).

Interleave-like access (identified as `data::access::interleave`) refers to non-contiguous subregions with a specific separation in each dimension. This access mode is also known in literature to refer to a *strided* or *non-unit strided* data layout. In 2D, variables of type `acc_interleave_t` define the 2D origin and a row / column separation (`acc_interleave_t` = *[origin row, origin column, row separation, column separation]*). Listing 5.6 exposes an *interleave* access to refer to the real and imaginary parts of a matrix of complex numbers.

Any subregion of a `dep::kind::in`, `::inout` or `::out` argument can be resolved as either `dep::view::copy` and `dep::view::map` (noted as `AnyDepKind` and `AnyDepView`). In case of `dep::view::copy`, a new copy or allocation (depending on `dep::kind`) to a contiguous block / interleaved regions in local memory is performed. Figure 5.5 illustrates these two accessibility modes used in previous listings.

When a task has finished manipulating *read-write* and / or *write-only* data objects, data is guaranteed to be visible again from the original data scope (i.e., if the original data object was

Listing 5.6: Example of `data::access::interleave` mode to interleave matrix elements.

```
/* Function callable by STEEL-RT from an unfolder context. Defined inside a kernel class*/
static void run(auto& dlg, dim_2d_t& matrix) {

  /* Create acc_interleave_t type alias. */
  using acc_interleave_t =
      data::traits<data::impl::buffer_2d>::access_type<data::access::interleave>;

  /* Wrap matrix into a data guard. */
  auto complexMatrixGuard =
      dep::handle<data::impl::buffer_2d, AnyDepKind, AnyDepView>(dlg, matrix);

  /* {0,0,1,0} and {1,0,1,0} arrays specify real and
      imaginary interleaved regions, respectively. */
  auto& realMatrix = complexMatrixGuard.get(acc_interleave_t({0,0,0,1}));
  auto& imagMatrix = complexMatrixGuard.get(acc_interleave_t({0,1,0,1}));

  /* [...] Operate on guarded realMatrix and imagMatrix. */
  [...]
}
```

guarded with `dep::view::copy`, then the updated subregions are transparently copied back to the original data object).

In addition, the *copy visibility* can be used to perform an *out-of-place subregion casting*, so that a casting from the underlying data type to another compatible data type is performed during data resolution. In Listing 5.7, assuming that the original matrix stores floating point double precision data, the previous block accessing mode of Listing 5.5 could be composed with a *down-casting* to generate 2D data subregions in floating point single precision.

Note that, although this subregion casting is only compatible with *copy visibility*, the user is free to perform an *in-place casting* in a bottom executor context (e.g., for *precision down-casting* the user can receive a data object referring to a double precision buffer, but casting the double precision data to single precision during computation).

5.2.3 2D data partitions

Listing 5.8 shows a kernel instantiation executable on an unfolder executor. Departing from Listing 5.4, with `mainExec` representing a particular unfolder executor on top of a bottom executor representing a CPU core, traces in Figure 5.6 reflect the associated data copies (in *main* and *unfolder* thread) and kernel executions, distinguished by green flags, in executor contexts (bottom and unfolder). First three copies in main thread reflect data copies from NVME to local CPU-RAM while the last one illustrate the update of the result back to the NVME space. Copies in unfolder thread reflect the *compaction* of subregions in original data into contiguous blocks in memory,

With regard to the bottom trace, the kernel running on the unfolder executor partitions, wraps data subregions into blocks, and dispatches 8 `impl::gemm` tasks (marked with green flags) running on finer-grained matrices (corresponding to `divm = divn = divk = 2`), which are run sequentially on a bottom executor handling a CPU core[1].

5.2.4 Out-of-core Computation (OOCC)

Out-of-core computation refers to the execution model in which the data to be processed is too big to fit in the local memory of a processing device. Essentially, in the OOCC model, *non-volatile* or *disk-like* memory spaces are treated as main memory spaces, and closer-to-processor memory

[1]The first kernel running on the bottom executor takes longer due to the initial Intel MKL execution initialization

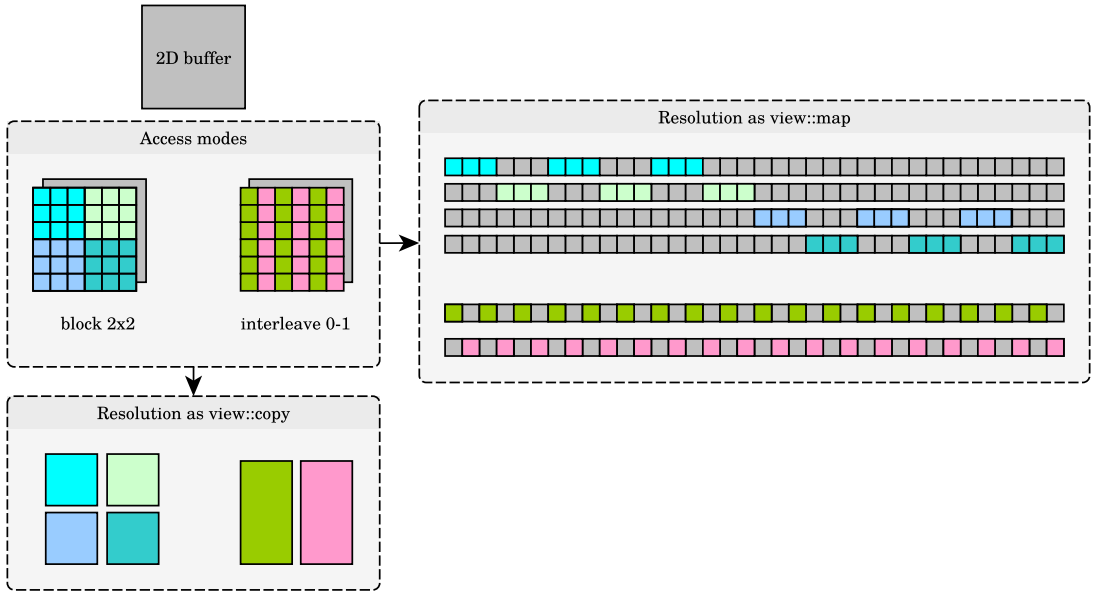


Figure 5.5: An example of accessibility modes for a 2D data buffer, corresponding to Listings 5.5 and 5.6. Guarding the origin data object as *copy* (`dep::view::copy`) implies a resolution of the subregions into newly-allocated smaller and contiguous 2D data objects in memory. Guarding the origin data object as *map* (`dep::view::map`) implies a resolution of the subregions into smaller and non-contiguous 2D data objects in memory, without any allocation.

space levels are considered as *caches*. Since STEEL-RT implements a data cache coherence protocol which treats disk-like memory spaces indistinguishably from other processor-local and volatile memory spaces, out-of-core functionality is intrinsically implemented.

An example to illustrate OOCC functionality in STEEL-RT is presented next. This use case consists of a 2048×2048 `impl::gemm` execution in double precision, which requires 100MB (33MB per matrix), so the original problem is decomposed into 8 sub-`impl::gemm` operating on 1024×1024 submatrices (resulted from `divm = divn = divk = 2` values in Listing 5.8), each submatrix requiring 8MB, thus each sub-`impl::gemm` requiring 24MB when the submatrices are resolved with `dep::view::copy`. In order for the initial 2048×2048 matrices to *not* be loaded into local CPU-RAM (thus avoiding a runtime failure due to lack of memory) the data guarding in Listing 5.4 must be done with *map visibility* (i.e., `dep::view::map`).

In relation to main code in Listing 5.4 is also assumed a executor configuration in which the main executor is an *unfolder* on top of a *mapper* encompassing a uniform set of bottom executors, each handling a sequential CPU core belonging to a SMP with a local CPU-RAM. Thus the main executor is an abstraction of *thread pool* to which partitioned tasks (in this case, 8 `impl::gemm` tasks operating on 1024×1024 matrices) are delegated.

If the SMP local CPU-RAM is restricted to 32MB, not all sub-`impl::gemm` *not* tied by dependencies would be able to actually run in parallel due to RAM contention. This is precisely illustrated in Figure 5.7, in which the execution of 8 `impl::gemm` kernels is done by different threads (i.e., there is *processing availability*) but it is effectively serialized due to the lack of *memory availability*. In particular, the *dependency_reacquire_taskwise* regions in the bottom trace reflect the situation in which the dependencies of each kernel (which are at first resolved one after another) have to be reacquired altogether (i.e., task-wise), after failure to reserve memory in a full or nearly full local CPU-RAM.

Copies (top trace) triggered by the unfolder thread are done as unfolder kernel proceeds, while copies in bottom threads illustrate both data evictions to a *fallback* memory space (in this case, the NVME space) and data copies of the kernel data.

Listing 5.7: Example of `data::access::block` mode to partition and cast quadrants.

```
1  /* Function callable by STEEL-RT from an unfolder context. Defined inside a kernel class*/
2  static void run(auto& dlg, dim_2d_t& matrix) {
3
4    /* Create acc_block_t type alias. */
5    using acc_block_t =
6        data::traits<data::impl::buffer_2d>::access_type<data::access::block>;
7
8    /* Wrap matrix into a data guard. */
9    auto matGuard =
10       dep::handle<data::impl::buffer_2d, AnyDepKind, AnyDepView>(dlg, matrix);
11
12   auto [nRows, nColumns] = get_parameters(matrix);
13   int blockRows = nRows / 2;
14   int blockColumns = nColumns / 2;
15
16   /* Create references to guarded quadrants. */
17   auto& upLeftQuadrant = matGuard.get
18       (acc_block_t({0, 0, blockRows, blockColumns}), ptype::f);
19   auto& upRightQuadrant = matGuard.get
20       (acc_block_t({0, blockColumns, blockRows, blockColumns}), ptype::f);
21   auto& downLeftQuadrant = matGuard.get
22       (acc_block_t({blockRows, 0, blockRows, blockColumns}), ptype::f);
23   auto& downRightQuadrant = matGuard.get
24       (acc_block_t({blockRows, blockColumns, blockRows, blockColumns}), ptype::f);
25
26   /* [...] Operate on guarded quadrants. */
27   [...]
28 }
```

5.2.5 Joint relaxation of visibility, granularity and precision

Given a general application or kernel, to be run on a device in a heterogeneous parallel platform with a deep and wide memory hierarchy, for a given optimization objective (e.g., execution time) it is in general uncertain whether its data dependencies (or some subset) must be *copied* or *mapped*. In particular, this optimal decision would in general depend on the original data placement prior to execution, the granularity of the data, the current occupation of the memory spaces and the contention of the interconnect buses. In summary, a *static* decision taken by the user with this regard would not be flexible enough to adapt to all possible runtime situations. Following the *user-defined execution relaxations* paradigm, the user should be able to *relax* and delegate these decisions so that the runtime can take them automatically depending on the particular runtime states and application characteristics.

Listing 5.9 illustrates a *synthetic* example (i.e., not necessarily reflecting a real-world case) example in which the data visibility, granularity and precision of block subregions of a 2D buffer are relaxed in a composed way, in the context of a synthetic kernel instance with identifier `app::impl::X`, run in an unfolder executor. In this case, STEEL-RT runtime decides the granularity (whether to divide the matrix into 2×2, 4×4 or 8×8) 2D blocks, according to sequence in line 8), passing it as `chosenDivision` variable in line 38. Similarly, STEEL-RT decides the *visibility* (whether it is `dep::view::copy` or `dep::view::map`, according to sequence in line 10), passing it as `chosenView` variable; and the floating point precision in which blocks are casted (according to the option sequence in line 12, whether it is casted to half-, simple or double precision).

Since `dep::view` are compile-time parameters in `dep::handle` functions, and in this case it is decided at run-time, a function dispatch in lines 27-31, to a function defined in lines 36-57, is needed.

Listing 5.8: Unfolder kernel for `impl::gemm`.

```
namespace steel::app {

template <sys::processor ProcessorId>
requires ProcessorId == sys::processor::abstract
class kernel<impl::gemm, ProcessorId> : public std::true_type {
private:
  /* Define the allowed granularities. */
  using divs_m = std::index_sequence<2, 4>;
  using divs_n = std::index_sequence<2, 4>;
  using divs_k = std::index_sequence<2, 4>;

  /* Compile definitions for access mode */
  using acc_block_t =
      data::traits<data::impl::buffer_2d>::access_type<data::access::block>;

public:
  /* Relax partition granularity according to user-defined sequences. */
  using expand = sched::relax<divs_m, divs_n, divs_k>;

  /* Function callable by STEEL-RT from an unfolder context. */
  static void run(auto& dlg, int divm, int divn, int divk,
      const dim_2d_t& inMatA, const dim_2d_t& inMatB, dim_2d_t& inoutMat) {

    /* Guard matrix arguments. */
    auto aGuard = dep::handle
        <data::impl::buffer_2d, dep::kind::in, dep::view::copy>(dlg, inMatA);
    auto bGuard = dep::handle
        <data::impl::buffer_2d, dep::kind::in, dep::view::copy>(dlg, inMatB);
    auto cGuard = dep::handle
        <data::impl::buffer_2d, dep::kind::inout, dep::view::copy>(dlg, inoutMat);

    auto [m, n, k] = traits<impl::gemm>::get_parameters(inMatA, inMatB, inoutMatC);

    auto mSize = m / divm;
    auto nSize = n / divn;
    auto kSize = k / divk;

    /* Decompose GEMM computation in blocks. */
    for (auto kk = 0u; kk < divk; ++kk) {
      for (auto mm = 0u; mm < divm; ++mm) {
        for (auto nn = 0u; nn < divn; ++nn) {
          auto& aBlock = aGuard.get
              (acc_block_t({mm * mSize, kk * kSize, mSize, kSize}));
          auto& bBlock = bGuard.get
              (acc_block_t({kk * kSize, nn * nSize, kSize, nSize}));
          auto& cBlock = cGuard.get
              (acc_block_t({mm*mSize, nn*nSize, mSize, nSize}));
          dlg( make_task<app::impl::gemm>(aBlock, bBlock, cBlock) );
        }
      }
    }
  }
};
} /* namespace steel::app */
```

Listing 5.9: A synthetic example for composing data-dependent orthogonal relaxations.

```
namespace steel::app {

template <sys::processor ProcessorId>
requires ProcessorId == sys::processor::abstract
class kernel<impl::X, ProcessorId> : public std::true_type {
private:
  /* Define the allowed granularities. */
  using div_opt = std::index_sequence<2, 4, 8>;
  /* Define the allowed visibility modes. */
  using view_opt = util::view_sequence<dep::view::copy, dep::view::map>;
  /* Define the allowed precision casting modes. */
  using prec_opt = util::ptype_sequence<ptype::h, ptype::f, ptype::d>;

  /* Compile definitions for access mode */
  using acc_block_t =
      data::traits<data::impl::buffer_2d>::access_type<data::access::block>;

public:
  /* Relax partition granularity according to user-defined sequences. */
  using expand = sched::relax<div_opt, view_opt, prec_opt>;

  /* Function callable by STEEL-RT from an unfolder context. */
  static void run(auto& dlg,
      int chosenDivision, dep::view chosenView, ptype chosenPrec,
      dim_2d_t& matrix) {

    if (chosenView == dep::view::copy) {
      run_detail<dep::view::copy>(dlg, chosenDivision, chosenPrec, matrix);
    } else if (chosenView == dep::view::map) {
      run_detail<dep::view::map>(dlg, chosenDivision, chosenPrec, matrix);
    }
  }

private:
  /* Function called from previous public run method, parametrized with dep::view. */
  template <dep::view View>
  static void run_detail(auto& dlg,
      int chosenDivision, ptype chosenPrec,
      dim_2d_t& matrix) {

    /* Guard matrix arguments. */
    auto matGuard = dep::handle
        <data::impl::buffer_2d, ArbitraryDepKind, View>(dlg, matrix);

    /* Calculate block rows and block columns. */
    int bRows = rows / chosenDivision;
    int bCols = columns / chosenDivision;

    for (auto mm = 0u; mm < chosenDivision; ++mm) {
      for (auto nn = 0u; nn < chosenDivision; ++nn) {
        auto& dataBlock = matGuard.get
            (acc_block_t({bRows * mm , nn * bCols, bRows, bCols}, chosenPrec));
        /* [...] = Delegate tasks to dlg executor operating on dataBlock */
        [...]
      }
    }
  }
};
} /* namespace steel::app */
```

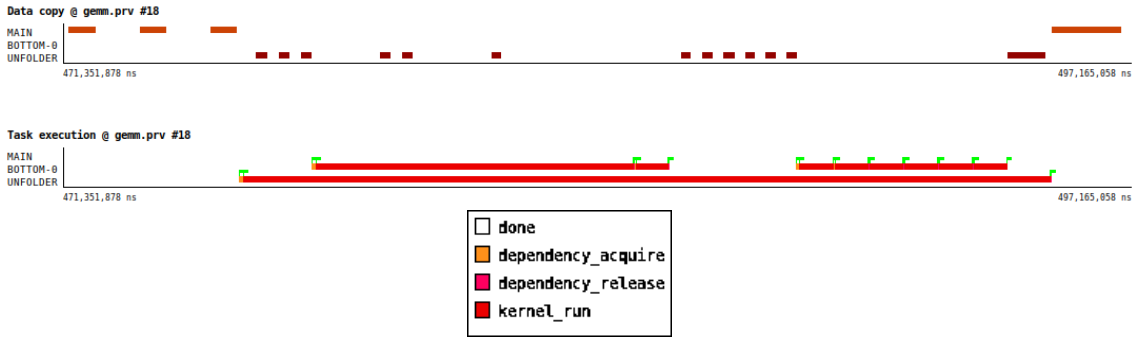


Figure 5.6: Execution traces for task and data partition. Basic execution traces illustrating data copies (top) and kernel execution (bottom) corresponding to Listing 5.8. Legend only corresponds to bottom trace colors.

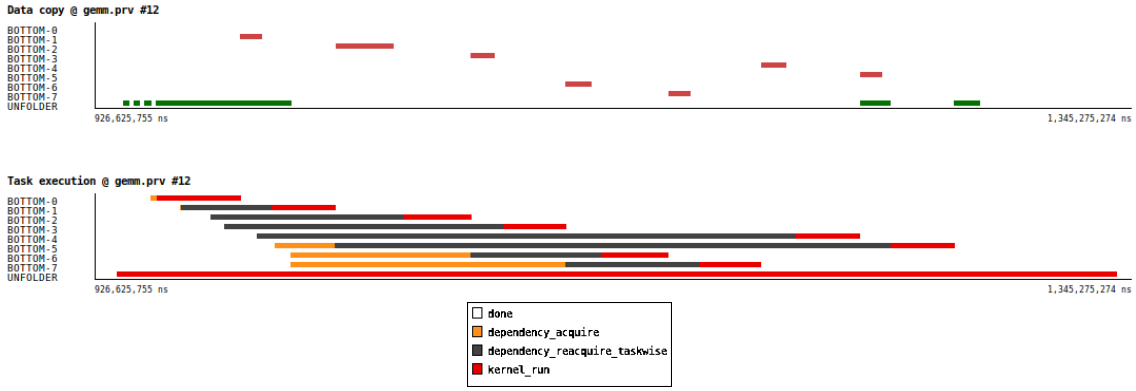


Figure 5.7: Data copy (top) and kernel execution (bottom) traces, in which the kernel executions are serialized due to contention in the memory space local to the SMP. Legend only corresponds to bottom trace colors.

5.3 Task management: parallelism and heterogeneity

5.3.1 Task- versus thread-parallelism

This section illustrates a set of execution traces for a matrix multiplication of size 16384×16384 and block size 1024×1024 running on MACHINE2 (see B.2). Three cases are exposed next in order to *(i)* expose basic tracing examples to illustrate the parallel execution of DAGs illustrated in Figures 5.1, 5.2, 5.3, and *(ii)* to introduce some performance issues and subtleties regarding two levels of parallelism (i.e., *task* and *thread*-level parallelism), repeatedly mentioned in this thesis. The corresponding executor hierarchy of the first case is an unfolder executor on top of a mapper, which itself handles a set of bottom executors tied to CPU cores. The second and third cases are extracted from an executor hierarchy consisted of an unfolder executor on top of a bottom executor handling the full multicore processor. Following the STSE model, the main `impl::gemm` is run in the context of the main executor (i.e., the *unfolder*), following the kernel defined in Listing 5.8

Figure 5.8 illustrates an execution trace in which each task is run with a single thread on each core. The first stage in which the parallelism is reduced is not due to the application – blocked `impl::gemm` DAG exposes uniform parallelism– but caused by the compaction of the initially non-compacted 2D sub-regions (note that the input matrices are linear in memory, and the subsequent 2D blocks are compacted according to a `dep::view::copy` mode, see lines 24-30 of Listing 5.8). Similarly, these compactions are performed as copies from non-compacted 2D subregions previously moved to the same local CPU-RAM according to the `dep::view::copy` specified in lines 15-25 of Listing 5.4. Note that in order to perform these 2D compactions from

non-volatile NVME memory space, it would be enough to just set the visibility mode in the same lines (15-25) as `dep::view::map`.

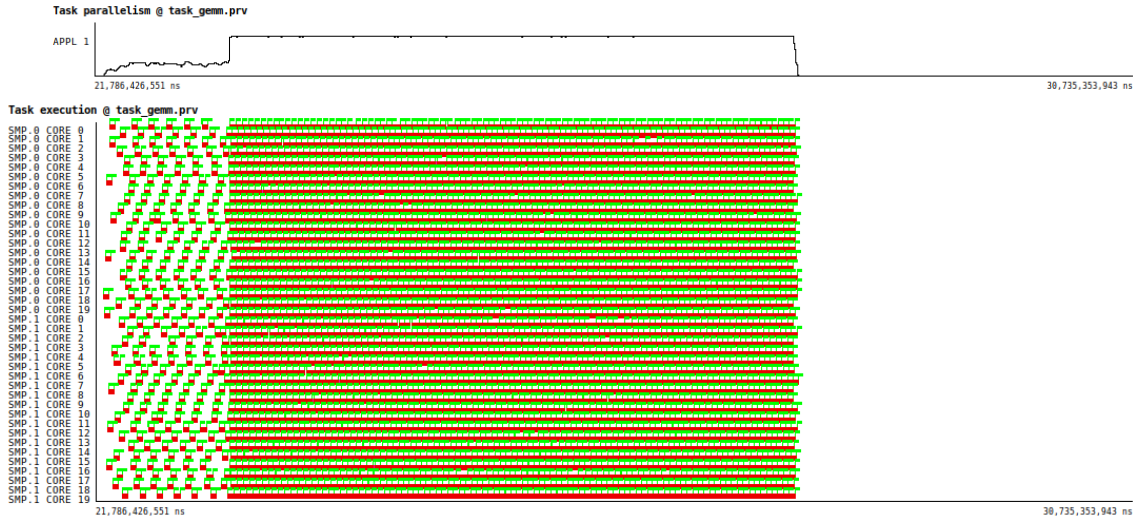


Figure 5.8: Execution traces with 1 thread per task. Top trace corresponds to instantaneous multiprocessor occupation (in [0-40] range). Bottom trace corresponds to task execution stages (`kernel_run` events) in red. First stage with lack of parallelism corresponds to 2D-subregion compaction.

Figure 5.9: Execution traces with 10 threads per task. Up to 4 (= 40 / 10) tasks can run concurrently. Top trace corresponds to instantaneous multiprocessor occupation (in [0-40] range). Bottom trace corresponds to task execution stages (`kernel_run` events) in red. First stage with lack of parallelism corresponds to 2D-subregion compaction.

As traces of Figures 5.8, 5.9 and 5.10 were normalized to the same duration, differences in performance (or equivalently, overall duration of red `kernel_run` events) demonstrate that, with regard to parallelism, both its quantity *and* its quality matter. In other words, best performing execution (Figure 5.8) corresponds to a purely task-parallel execution (zero *intra*-task parallelism, recalling the vocabulary of Section 2.3), in spite of lacking task parallelism in the first 2D-data compaction stage. On the contrary, *fork-join*-based executions (Figures 5.9 and 5.10), demonstrate that not an increased level of processor occupation necessarily involves an increase in performance.

5.3.2 Support for task moldability

Task-wise variable threading or *task moldability* was motivated in Section 1.3.3.1, explored statically in Section 2.3, and treated in the context of bottom executors on top of multicores and GPUs in Section 3.1.2. In the following, the simplicity of `impl::gemm` application permits to demonstrate the support for task moldability in STEEL-RT in the simplest way.

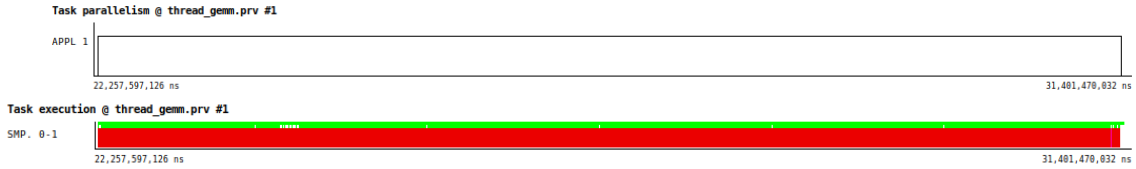


Figure 5.10: Execution traces with 40 threads per task. Tasks are run sequentially. Top trace corresponds to instantaneous multiprocessor occupation (in [0-40] range). Bottom trace corresponds to task execution stages (`kernel_run` events) in red.

Specifically, sections 1.3.3.1 and 2.4, illustrated the motivation behind task-wise thread scheduling decisions, potentially desirable in a context in which a set of heterogeneous and inter-dependent tasks share the resources (memory and processing) of a (possibly heterogeneous) parallel architecture. On the contrary, `impl::gemm` application is very simple (in terms of parallelism) and homogeneous (it decomposes into other `impl::gemm`) and in this example related to moldability is only tested on an homogeneous (SMP) architecture. As the goal of this chapter is not expose any detailed performance analysis, but demonstrate a set of STEEL-RT features from specific examples in the simplest way possible, considerations about eventual performance improvements derived from run-time task moldability scheduling decisions are left for future work.

The following exposition illustrates the support for task moldability in STEEL-RT, and it is based on the hierarchy of Figure 5.11.

Figure 5.11: Executor hierarchy for the moldability case.

When this executor hierarchy is instantiated to the multicore architecture in Section B.2, and the moldable kernel defined in Listing 5.10, then the bottom executor handles a SMP with 40 cores with two *worker* threads. These threads pull tasks from the bottom executor task queue and executes them in a fork-join fashion (hence the worker threads are also referred as *forker* threads), with a number of threads decided by STEEL-RT, and relaxed by the user according to the occupations defined in line 11 of Listing 5.10.

The *expanded-execution* performed in STEEL-RT in terms of task parallelism, thread parallelism per task, and task execution events is illustrated in Figure 5.12. These traces correspond to an execution of a blocked 16384×16384 `impl::gemm` in double precision with block size 4096×4096. During `kernel_schedule` events the number of threads is automatically set by STEEL-RT, and a strict thread-to-core binding is performed to avoid core oversubscription (which is desirable in compute-bound kernels).

In this case, since only two *threading* possibilities (either occupy half –20 cores– or full –40 cores– SMP) are considered, the two worker threads are able to run tasks in parallel only if their tasks were scheduled with 20 cores. Otherwise, if at a given moment 20 cores (i.e., threads) are assigned to a given forker thread and the other forker thread is running a task with 40 cores (i.e., threads), then the former will block until the latter is finished. This blocking is reflected in `resource_acquire` events in the bottom trace of Figure 5.12 (*resource* meaning the *processing* resource), also directly correlated with task and thread parallelism traces (top and middle traces,

Listing 5.10: Definition for moldable `impl::gemm` kernel targeting BLAS library.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires /* Kernel definition if ProcessorId is a multicore cpu. */
    constraint<ProcessorId>::support_device == support::device::cpu &&
    constraint<ProcessorId>::parallelism > 1
class kernel<impl::gemm, ProcessorId> : public std::true_type {
private:
  /* Define set of allowed core partitions as
      1/1 and 1/2. 100% and 50% multicore occupation. */
  using core_occupations = std::index_sequence<1, 2>;

public:
  /* Mandatory type definition. */
  using expand =
      sched::fix_relax<support::ext_library::cpu_threads, ProcessorId, core_occupations>;

  /* Moldable kernel definition, callable by STEEL-RT. */
  static void run(unsigned nThreads,
      const dim_2d_t& matA, const dim_2d_t& matB, dim_2d_t& matC) {

    const auto [mm, nn, kk, xLead, yLead, zLead, pt] =
        traits<impl::gemm>::get_parameters(matA, matB, matC);
    auto [aPtr, bPtr, cPtr] = traits<impl::gemm>::get_pointers(matA, matB, matC);

    /* Execute GEMM. */
    wext::blas::call_gemm
        (nThreads, mm, nn, kk, xPtr, xLead, yPtr, yLead, zPtr, zLead, pt);
  }
};
} /* namespace steel::app */
```

respectively). Finally, once the task running on 40 cores finishes, a `resource_release` event marks the release of the multicore. Note that `resource_release` and `kernel_schedule` events are present in the bottom trace but they are barely visible because their duration is negligible in an execution dominated by kernel execution (`kernel_run`) and resource acquisition (`resource_acquire`) stages.

5.3.3 Support for heterogeneous computation

As explained in Section 4.2.2.5, heterogeneous computation is automatically enabled when the platform exposes at least two heterogeneous processors (e.g., a SMP and a GPU) and the user provides compatible kernel implementations for them –i.e., a `class` kernel definition for which the predicate after `requires`, evaluated on the corresponding processor identifiers `sys::processor`, is satisfied. With this regard, Listing 5.11 exposes the kernel definition to enable a `impl::gemm` execution on GPU using cuBLAS library.

A 16384×16384 problem size and 1024×1024 blocks was performed on the heterogeneous platform specified in Appendix B.2, abstracted as a two-level executor hierarchy composed by an unfolder executor on top of a mapper handling all CPU cores and all GPUs. The corresponding execution traces are shown in Figure 5.13. The top-most thread in top trace (representing data copies) marks NVME-to-local CPU-RAM copies, while bottom-most thread in top trace marks 2D data compactions in local CPU-RAM. In addition, GPU threads (second and third) also trigger data copies from- and to local CPU-RAM, to respectively acquire and publish data in their GPU-RAM memory spaces. All data copies are managed in all memory spaces implicitly handled by STEEL-RT (NVME, local CPU-RAM in respective SMPs, and local GPU-RAM in respective GPUs), following a directory-based cache protocol.

Similarly to traces in Figure 5.8, in the task execution trace of Figure 5.13 (bottom), a first stage with lack of parallelism is due to compaction, followed by a stage where all the parallelism

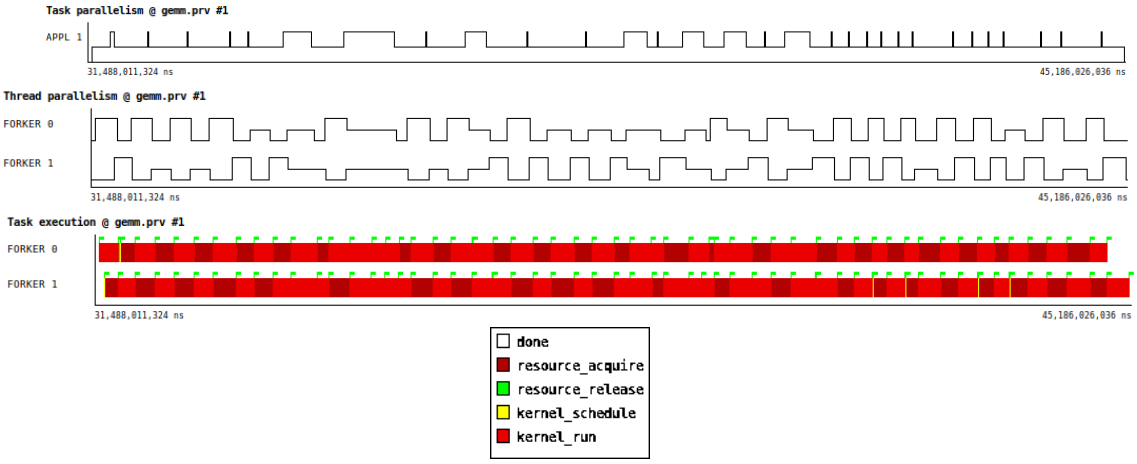


Figure 5.12: Execution traces with moldable tasks. Runtime decides the number of threads for each task, occupying the full or half of the multicore. Traces correspond to *task parallelism* (top trace, values in [0, 1, 2]), *thread parallelism per task* (middle trace, values = [0, 20, 40]), and execution events (bottom trace) with associated legend.

of the execution is exploited, and a third stage in which all the threads of the bottom executors (handling CPU cores and GPUs) have finished, but the unfolder thread is updating the results of all tasks into its single read-write data object. After the unfolder thread has finished, the final result is copied back to the original NVME memory space (shown as the last block of the main thread in top trace). Note also that, as `impl::gemm` kernels are extremely data-parallel, they are well-fitted to GPUs, which is reflected in the higher performance in corresponding GPU0 and GPU1 threads of bottom trace.

5.4 Hierarchical executor deployment

Previous examples relied on tasks being managed by the *default executor*, deployable by means of the `executor::deploy_main` function, which abstracts a basic executor hierarchy derived from user-defined `kernel` class specializations.

In this section, after introducing application-related code for the Cholesky factorization, the executor deployment interface is exposed to show how a more *explicit* and controlled executor deployment can be used to target some platform architectural features, also illustrating how minimal differences in code result into substantial qualitative differences in the execution traces obtained.

5.4.1 STEEL implementation for the Cholesky factorization

Listing 5.12 shows a sample code for the Cholesky factorization of a symmetric positive definite matrix leveraging the STEEL API. In line 8, the executor representing the complete system is deployed and a reference to it taken. The actual computation following the STSE model is depicted in line 14, in which a single Cholesky factorization task –`potrf` following LAPACK naming [9]– is created and asynchronously delegated to the system executor previously created.

Listings 5.13 and 5.14 implement wrapper code to forward (OPENMP or CUDA) kernel calls to final external libraries or implementations. Internally, the `wext` namespace exposes useful types and wrapper functions of calls to external libraries –in this case LAPACK/Intel MKL and NVIDIA cuSOLVER libraries–. In line 13 of Listing 5.14, the variable `handle` has a type that encapsulates the cuSOLVER opaque type `cusolverHandle_t`, which at run-time stores a cuSOLVER handle associated to a particular GPU device together with an allocator associated to the local RAM of that device, needed to perform a memory allocation prior to the Cholesky computation on the

Listing 5.11: Definition for a GPU `impl::gemm` kernel targeting cuBLAS library.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires /* Kernel definition if ProcessorId is GPU and cuBLAS is installed. */
    constraint<ProcessorId>::support_device == support::device::gpu &&
    constraint<ProcessorId>::has_ext_library_v<support::ext_library::cublas>
class kernel<impl::gemm, ProcessorId> : public std::true_type {
public:
  /* Mandatory type definition. */
  using expand = sched::fix<support::ext_library::cublas>;

  /* Moldable kernel body definition, callable by STEEL-RT. */
  static void run(cublasHandle_t ch,
      const dim_2d_t& matA, const dim_2d_t& matB, dim_2d_t& matC) {
    /* Get kernel parameters and pointers. */
    const auto [mm, nn, kk, xLead, yLead, zLead, pt] =
        traits<impl::gemm>::get_parameters(matA, matB, matC);
    auto [aPtr, bPtr, cPtr] = traits<impl::gemm>::get_pointers(matA, matB, matC);

    /* Execute GEMM. */
    wext::cublas::call_gemm
        (ch, mm, nn, kk, xPtr, xLead, yPtr, yLead, zPtr, zLead, pt);
  }
};
} /* namespace steel::app */
```

Listing 5.12: Main program running Cholesky factorization with STSE paradigm.

```
#include "steel.hpp"
using namespace steel;

void main(int argc, char** argv) {

  const auto pMap = build_parse_map(arg, argv);

  auto& mainExec = executor::deploy_main();

  auto matrixGuard = dep::handle
      <app::data::impl::buffer_2d, dep::kind::inout, dep::view::copy>
      (mainExec, pMap["--inoutput"]);

  executorRef( dep::make_task<user::impl::potrf>(matrixGuard) );
}
```

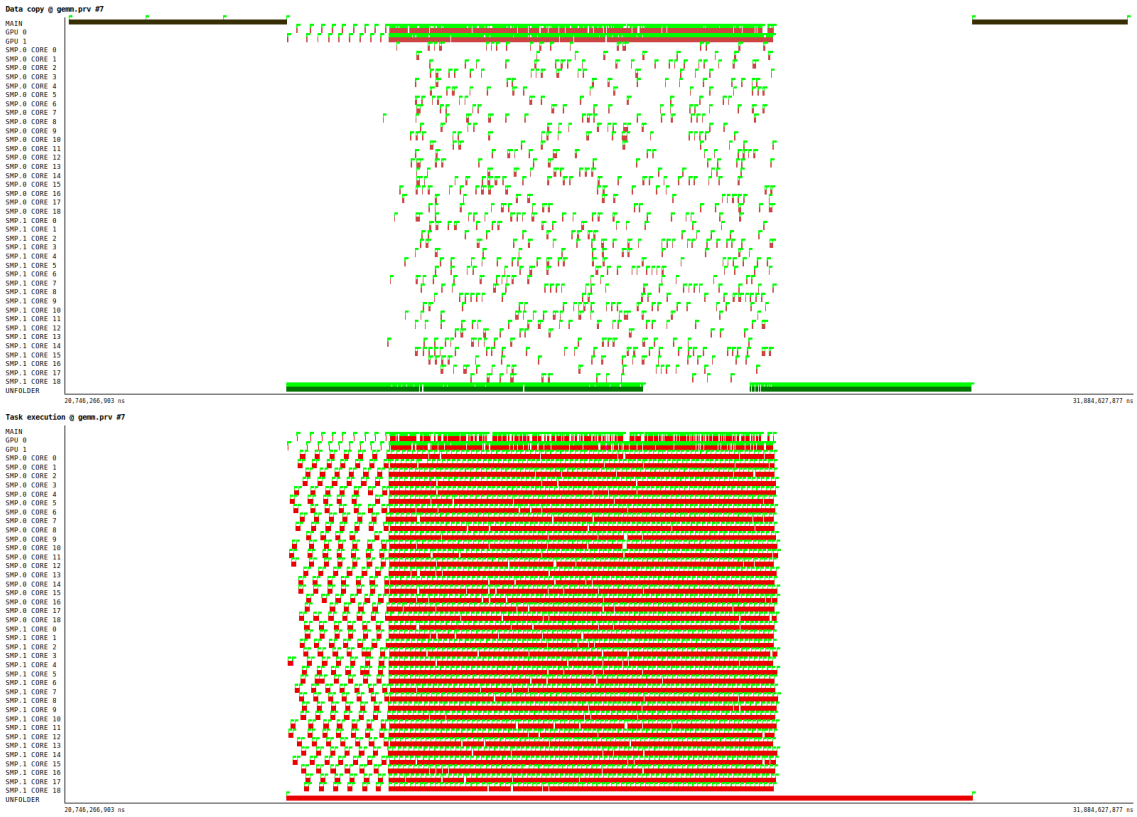


Figure 5.13: CPU-GPU heterogeneous execution traces of data copies (top trace) and kernel executions (bottom trace).

GPU (see [114]). This *in-kernel* allocation is done within `wext::cusolver::call_potrf` function.

Listing 5.15 exposes the tiled Cholesky implementation, that allows `app::impl::potrf` tasks to run on unfolder executors. Lines 22 to 44 express the actual function body of the Cholesky blocked algorithm, (already presented as a C implementation in Listing 2.1) run by the partitioner executor thread, which forwards the computation through asynchronous calls via `dlg`.

5.4.2 Leveraging STEEL to exploit heterogeneity. A step-by-step use case

The composability property of executors enable the construction of complex executor hierarchies. In particular, this section exposes how four incrementally complex executor hierarchies are created. The target application is a Cholesky factorization identified as `impl::potrf`, and the target heterogeneous architecture is MACHINE2, replacing one of its Tesla V100 by a Nvidia GeForce GTX 1080 to gain heterogeneity. Note that the blocked Cholesky factorization decomposes the computation in terms of BLAS-3 and LAPACK tasks, namely `impl::potrf` (factorization of diagonal blocks), `impl::gemm` (general matrix-matrix multiplication), `impl::syrk` (symmetric rank-k update) and `impl::trsm` (triangular system solve with multiple right-hand sides). In the following Listings that expose executor deployments, the use of `using namespace steel::executor` is assumed.

Listing 5.13: Definition for a moldable `impl::potrf` kernel targeting LAPACK library.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires /* Kernel definition if ProcessorId is a multicore cpu. */
    constraint<ProcessorId>::support_device == support::device::cpu &&
    constraint<ProcessorId>::parallelism > 1
class kernel<impl::potrf, ProcessorId> {
private:
  /* Define set of allowed core partitions as
      1/1 and 1/2. 100% and 50% multicore occupation. */
  using core_occupations = std::index_sequence<1, 2>;

public:
  /* Kernel body definition, callable by STEEL-RT. */
  static void run(unsigned nThreads, dim_2d_t& matrix) {
    auto [n, nLead, pt] = traits<impl::potrf>::get_parameters(matrix);
    auto ptr = traits<impl::potrf>::get_pointer(matrix);
    wext::lapack::call_potrf(nThreads, n, ptr, nLead, pt);
  }
};
} /* namespace steel::app */
```

Listing 5.14: Definition for a GPU `impl::potrf` kernel targeting cuSOLVER library.

```
namespace steel::app {

template <sys::processor ProcessorId>
requires /* Kernel definition if ProcessorId is GPU and cuSOLVER is installed. */
    constraint<ProcessorId>::support_device == support::device::gpu &&
    constraint<ProcessorId>::has_ext_library_v<support::ext_library::cusolver>
class kernel<impl::potrf, ProcessorId> : public std::true_type {
public:
  /* Mandatory type definition. */
  using expand = sched::fix<support::ext_library::cusolver>;

  /* Kernel body definition, callable by STEEL-RT. */
  static void run(auto handle, dim_2d_t& matrix) {
    auto [n, nLead, pt] = traits<impl::potrf>::get_parameters(matrix);
    auto ptr = traits<impl::potrf>::get_pointer(matrix);
    wext::cusolver::call_potrf(handle, n, ptr, nLead, pt);
  }
};
} /* namespace steel::user */
```

Listing 5.15: Implementation of the unfolder kernel for `impl::potrf`.

```
namespace steel::app {
template <sys::processor ProcessorId>
requires ProcessorId == sys::processor::abstract
class kernel<impl::potrf, ProcessorId> : public std::true_type {
private:
  using div_opt = std::index_sequence<4, 8, 16, 32>;

  /* Compile definitions for access mode. */
  using acc_block_t =
      data::traits<data::impl::buffer_2d>::access_type<data::access::block>;

public:
  /* Relax partition granularity according to user-defined sequences. */
  using expand = sched::relax<div_opt>;

  static void run(auto& dlg, unsigned div, dim_2d_t& matrix) {
    auto mGuard = dep::handle
        <data::impl::buffer_2d, dep::kind::inout, dep::view::copy>(dlg, matrix);
    const auto n = traits<impl::potrf>::get_parameters(matrix);
    const auto bSize = n / div;

    /* Cholesky blocked algorithm */
    for (auto k = 0u; k < div; ++k) {
      auto& kkGuard = mGuard.get(acc_block_t({k*bSize, k*bSize, bSize, bSize}));
      /* POTRF */
      dlg( make_task<impl::potrf>(kkGuard) );
      for (auto i = k + 1; i < div; ++i) {
        auto& ikGuard = mGuard.get(acc_block_t({i*bSize, k*bSize, bSize, bSize}));
        /* TRSM */
        dlg( make_task<impl::trsm>(kkGuard, ikGuard) );
      }
      for (auto i = k + 1; i < div; ++i) {
        auto& ikGuard = matGuards[{i*bSize, k*bSize, bSize, bSize}];
        for (auto j = k + 1; j < i; ++j) {
          auto& jkGuard = mGuard.get(acc_block_t({j*bSize, k*bSize, bSize, bSize}));
          auto& ijGuard = mGuard.get(acc_block_t({i*bSize, j*bSize, bSize, bSize}));
          /* GEMM */
          dlg( make_task<impl::gemm>(ikGuard, jkGuard, ijGuard) );
        }
        auto& iiGuard = mGuard.get(acc_block_t({i*bSize, i*bSize, bSize, bSize}));
        /* SYRK */
        dlg( make_task<impl::syrk>(ikGuard, iiGuard) );
      }
    }
  }
};
} /* namespace steel::app */
```

Listing 5.16: Deployment of an unfolder on top of all available sequential processors (Fig. 5.14).

```
1 auto& mainExecutor = deploy( deploy_only<sequential>() );
```

Step 1: Exploiting task-parallelism with a Partitioner-Pool hierarchy

The first hierarchy enables homogeneous task parallelism on individual cores of one of the SMP sockets in the target architecture by mapping individual tasks to them. The unfolder executor delegating tasks to a set of bottom executors serves as the *single-executor* entry point (See Figure 5.14 and Listing 5.16).

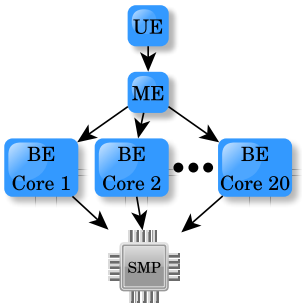


Figure 5.14: An executor hierarchy that unfolds and maps tasks to a core pool. *UE:* unfolder executor. *ME:* mapper executor. *BE:* bottom executor.

The trace in Figure 5.15 exposes a task-parallel execution in 20-core SMP for $n = 4096$ and $nb = 512$. Note how the runtime transparently deploys 20 execution threads (one per line in the trace) and maps (sub-)tasks to them managing data dependencies.

Step 2: Adding CPU-GPU heterogeneity with a Partitioner-heterogeneous execution

To enable heterogeneous computation by means of a single GPU exploitation, another bottom executor tied to a GPU device could be appended to the previous hierarchy. In this case, only tasks generated by the partitioner that provide a GPU-compatible *kernel* class specialization can be mapped to that bottom executor (see Figure 5.16 and Listing 5.17).

The corresponding execution trace in Figure 5.17 exposes reports a real heterogeneous execution of 16 threads running on an SMP (single socket, bottom rows) and 4 CUDA streams running in the Geforce GTX1080 GPU (top rows). Observe how, in this case, a simple change in user code yields substantial changes in terms of thread deployment and, even though not visible in the

Figure 5.15: Step 1. Execution trace for a single-socket / multi-core execution. Task colors: POTRF: magenta. GEMM: blue. SYRK: orange. TRSM: green.

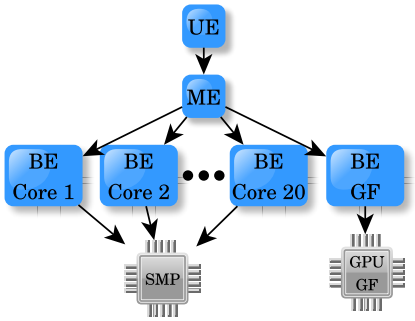


Figure 5.16: An executor hierarchy encompassing a CPU-GPU platform with uniform partitioning. *UE:* unfolder executor. *ME:* mapper executor. *BE:* bottom executor. *GF:* NVIDIA GeForce GPU.

Listing 5.17: Deployment of a CPU-GPU hierarchy (Fig. 5.16).

```
1 auto& mapperOnCpuGpu = deploy( deploy_only<sequential>(), deploy_first<cuda_device>() );
2 auto& mainExecutor = deploy( mapperOnCpuGpu );
```

traces, data transfer and heterogeneous device management.

Step 3: Heterogeneous task partitioning with tied partitioner executors

On heterogeneous systems, optimal performance for different devices usually correspond to different task granularities. For CPU-GPU systems, GPU typically demands coarser tasks than individual CPU cores in order to fully saturate its resources. With the STEEL-API, previous executor hierarchy can be expanded by associating an unfolder executor to generate fine-grained partitions to the CPU cores, and setting a coarse-grain partitioner as the main executor (Figure 5.18 and Listing 5.18). Only those tasks that provide a `kernel` specialization compatible for unfolder executors will be processed by the fine-grained *partition* executor. In this case, the decision is to allow only the `potrf` tasks to include that feature, and thus only diagonal Cholesky factorizations will be partitioned and mapped to individual cores. This is a common strategy found in literature [16].

Listing 5.18: Deployment for heterogeneous partitioning (Fig. 5.18).

```
1 auto& cpuUnfolder = deploy( deploy_only<sequential>() );
2 auto& mapperHeterog = deploy( deploy_first<cuda_device>(), cpuUnfolder );
3 auto& mainExecutor = deploy( mapperHeterog );
```

The associated execution trace in Figure 5.19 corresponds to a heterogeneous CPU-GPU execution with the same thread setting as in the previous step, for a problem size $n = 16384$. Task

Figure 5.17: Step 2. Execution trace for a heterogeneous CPU-GPU execution.

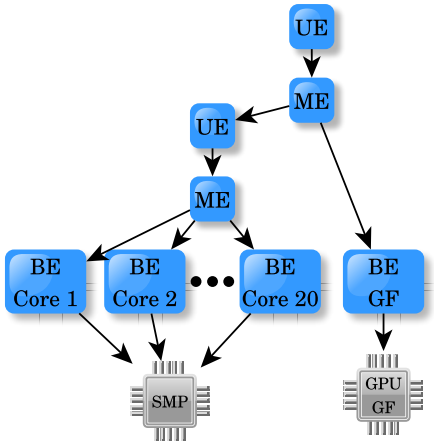


Figure 5.18: An executor hierarchy encompassing a CPU-GPU platform with non-uniform partitioning. *UE:* unfolder executor. *ME:* mapper executor. *BE:* bottom executor. *GF:* NVIDIA GeForce GPU.

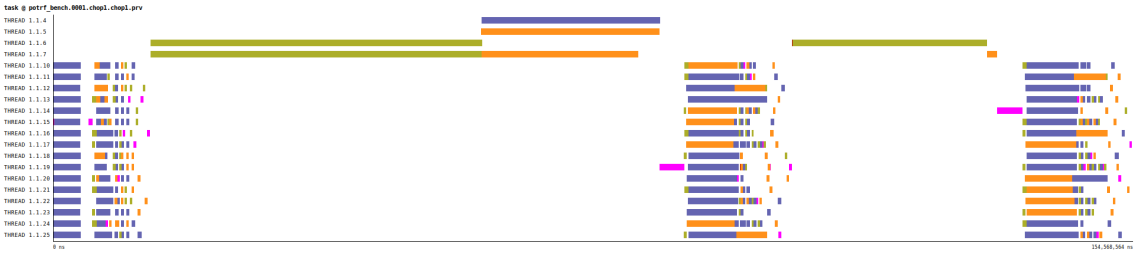


Figure 5.19: Step 3. Excerpt of the factorization using a heterogeneous-partitioning strategy.

granularity is heterogeneous: $nb = 4096$ in the first partitioning level, $nb = 512$ in the second partitioning level. The trace correspond to the last steps of the Cholesky factorization. Note how task granularity differs on GPU threads (top four rows) and CPU threads (bottom rows); actually, the fine-grained tasks observed in three successive groups of the CPU correspond to three `potrf` tasks decomposed automatically by STEEL-RT into smaller tasks following the double-partitioner hierarchy.

5.4.2.1 Step 4: Heterogeneous forking by means of multiple *forker* executors

Finally, the previous hierarchy is expanded by appending an SMP and a GPU (NVIDIA Volta V100), each of which handled by an independent bottom executors (See Figure 5.20 and Listing 5.19).

The execution trace for the same problem dimension and thread granularities as those for the previous step is shown in Figure 5.21. The first row corresponds to the thread-forking tasks for the second SMP (socket). Thus, those tasks will be internally threaded using, in this case, 20 threads. Rows 2 and 3 correspond to two CUDA streams delivering tasks to the GTX1080, and the following 4 rows to the corresponding CUDA streams associated to the V100 GPU. The remaining threads correspond to individual cores in the first CPU socket.

Note that the aim of this section is neither to provide absolute performance results nor a thorough performance optimization procedure. Optimization techniques (number of threads per task; task partitioning granularity on unfolder executors; advanced locality-aware policies to assist *mapper* executors, ...) have been previously applied in the literature and can be also integrated into

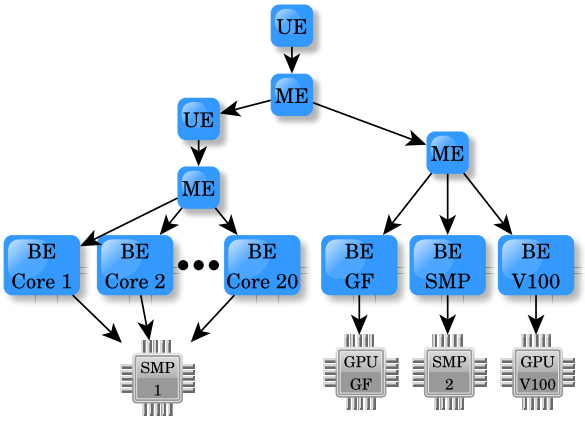


Figure 5.20: An executor hierarchy encompassing a CPU-GPU platform with non-uniform partitioning and forking. *UE:* unfolder executor. *ME:* mapper executor. *BE:* bottom executor. *GF:* NVIDIA GeForce GPU. *V100:* NVIDIA Volta GPU.

Listing 5.19: Deployment for heterogeneous partitioning and forking (Fig. 5.18).

```
1 auto& heterogMapper = deploy(
2     deploy_first<cuda_device>(),
3     deploy_first<multicore>(),
4     deploy_first<cuda_device>() );
5 auto& finePartitioner = deploy( deploy_only<sequential>() );
6 auto& coarseMapper = deploy( heterogMapper, finePartitioner );
7 auto& coarsePartitioner = deploy( coarseMapper ); /* mainExecutor */
```

STEEL-RT. Nevertheless, the goal was to illustrate how simple modifications in user codes can be performed to expose sophisticated executor hierarchies (hence, complex heterogeneous architectures) and yield dramatic changes in the underlying runtime execution.

5.5 Summary

This chapter has exposed by means of simple *data*-related and *compute*-related and executor deployment cases a set of functionalities of STEEL-RT and its API. Data-related cases revolved around some of the concepts exposed in Chapter 4, in particular data visibility, data partition, and accessing modes were jointly exposed in the form of codes written according to STEEL-API, and a set of simple execution traces. In particular, both data- and compute-related cases showed how the STSE model –in which a single executor (abstracting the full parallel platform) receiving a single task (representing the full application), reading and manipulating data residing in a non-

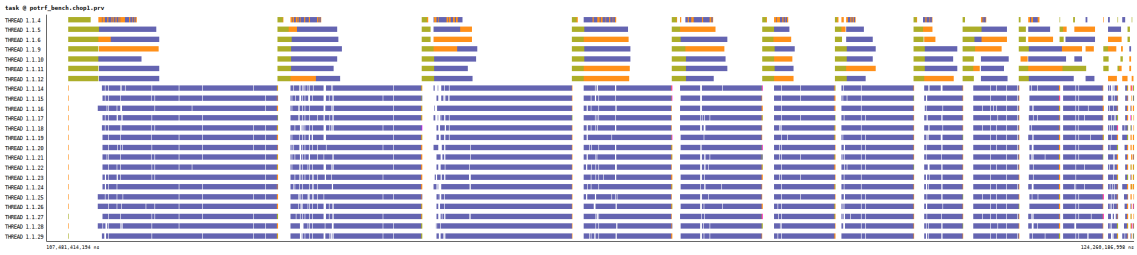


Figure 5.21: Step 4. Execution trace for a complex executor hierarchy exploiting full heterogeneity.

volatile memory space–, yield rich parallel executions with minimal incremental additions in code. Regarding the executor API, it has been shown how *explicit* executor deployment together with specific task featurizations could help to expand the execution possibilities accessible at run-time in terms of heterogeneous executions, heterogeneous partitioning, or runtime task moldability.

Relations that lace
the fabric of space
from a source hologram,
may transform with grace
at a gentle pace
into a functional program.

6

STEEL as a functional model

In this chapter, part of the abstract model explained in Chapter 3 and the STEEL-API exposed in Chapter 4, are represented in terms of a model written in functional language. As mentioned, one of the motivations of STEEL-PM is to increase programmability by letting the user abstract away and delegate the execution control flow to a runtime system. In particular, Chapter 4 explained how abstraction is raised by means of declarative patterns that express execution opportunities to be exploited by the runtime. Also, as application correctness at run-time and debuggability are major concerns, specially in concurrent and parallel scenarios, these patterns are also designed to favor proper side-effect encapsulation (see Section 1.2.1.4).

In a broader sense, functional programming patterns are central elements in STEEL-PM in three clear ways, namely:

Template metaprogramming. As explained in Chapter 4, the building process of applications following STEEL-API rules strongly relies on a two-way interaction between user code and STEEL-API template code, in the sense that user declares functions to be called by STEEL-RT, and implicitly instantiates functions in the STEEL-RT template library. In essence, this interaction between user code and STEEL-RT code is parsed by the C++ compiler under *strong typing* and functional rules. The compilation process ultimately results in a binary executable, built from specific template instantiations that represent application-platform bindings. In other words, STEEL-API can be contemplated as a small domain-specific language, built from stricter functional C++ template metaprogramming language rules, designed to develop HPC applications under the *execution relaxations* paradigm.

STSE model and pure functions. The Single Task-Single Executor model partially relies on the idea that all the computational components or building blocks of an application can be abstracted away into a single task with specific input and output arguments. This is done by declaring ways –in terms of user-defined pure functions– in which the application building blocks can be decomposed and executed in different architectures. On the other hand, the STSE model relies on the idea that different execution contexts can be composed and accessed via a single interface –the top-most executor–, which is a *de facto* pure function from the user perspective –i.e., the eventual state changes of the components abstracted by the top-most executor during a task execution are not accessible by the user under any circumstance–.

Execution flow is declared, not imposed. Execution flow is constrained in two ways: *(i)* by the deployed executor tree and *(ii)* by specific user declarations (i.e., *execution relaxations*). Both constraints are imposed at compile-time, so that the *flow of execution* or *workflow* is driven by the runtime system across the deployed executors. This is a declarative-like paradigm shift, as the user no longer specifies –at least in a *direct* sense– how the application execution is mapped to the underlying hardware.

With these ideas in mind, in the functional model presented next, these three characteristics are clarified thanks to a more terse and concise functional notation. Thus, the primary goal of this chapter is to present a functional model that clearly captures the relations and dependencies between some of the modules –executors, tasks, scheduling policies, etc.– explained in previous chapters. Moreover, the secondary goal is to express with greater precision how the different stages of the development process –development, installation, compilation and execution– are related with some important functions.

This chapter is structured as follows. Section 6.1 exposes some basic concepts of functional programming. Section 6.2 introduces the functional model of STEEL in three separated parts. First, Section 6.2.1 explains how execution paths from an execution context are generated (or *bursted*) from user-defined functions, and ultimately composed from an executor tree hierarchy. Second, Section 6.2.2 exposes how asynchronous delegation of tasks related by data dependencies is expressed in the context of an executor. Finally, Section 6.2.3 gathers previous ideas to build a functional framework that helps to visualize how a single task running on the top-most executor of the tree results into an *asynchronous* and *expanded* flow of computation. Finally, the last Section 6.3 exposes some limitations of the current model, which will motivate future improvements focused on a purity considerations.

6.1 Introduction

In the following, some notation and fundamental ideas are presented. They are general concepts in the functional programming paradigm, and the exposed ones are a minimal subset of what can be found in a standard library of a functional language implementation. The specific implementations of the following types, data structures and functions depend on programming language specifications and are irrelevant in the context of this chapter.

As a side note, runtime implementations of purely functional programming specifications usually exploits *referential transparency* (see section 1.2.1.1), which implies that once a function is applied, its result may be memoized. Consequently, eventual calls of a function with the same arguments returns the value computed in the first call, not function re-application –recall that "=" operator in functional programming is used under *equality* semantics–.

6.1.1 Basic elements

The basic elements to be exposed in this section consist of types, functions, functors and monads. After exposing their syntax and rules, a set of data structures –pair, set, list, stream, tree– will be defined.

6.1.1.1 Types

Type definition is expressed by the "::" syntax, in the line of Haskell notation [80], so that the expression $A :: B$ defines a type A from another defined type B.

Types can be defined by composition to form *algebraic data types* using "+" and "×" operators:

Sum types: $A :: B + C$ defines type A as a *sum type* of types B and C, so that values of type A can be either of type B or C. The operator "|" is often used instead of "+". The cardinality –i.e., the number of elements– of type A is the sum of the cardinalities of B and C.

Product types: $A :: B \times C$ defines type A as a *product type* of types B and C, so that values of type A are a composition of values of types B and C. The cardinality of type A is the product of the cardinalities of B and C.

Type constraint: A type definition based on other types can be constrained with an expression inside parenthesis in the form (*Constraint* $\Rightarrow$ *Type name*) after "::". For example, a type A defined from existing data types B and C, enforcing B to be an integer, could be expressed as $A :: (Integral \Rightarrow B)B \times C$.

Recursive types: Types can be defined recursively, in the sense that the defined type could exist in the expression that defines that type. For example, type $A :: B + A$ represent values that can take a value of type B or a value that can be either B or A. This is useful to express recursive data structures, for example graphs composed by nodes that reference other nodes.

6.1.1.2 Functions

Function signatures: A single-valued function of type F mapping values of type A into values of type B defines its type as $(F :: A \longrightarrow B)$. The type of a function is also named as *signature*. Functions receiving multiple parameters have signatures built by chaining the types of the arguments. (e.g., $(F :: A \longrightarrow B \longrightarrow ... \longrightarrow C)$ is a function returning values of C from multiple arguments of types A, B, ...).

In particular, "$\longrightarrow$" is an operator that defines a function type from the two types, and in this *infix* form the types at each left and right side represent the function input and output respectively. The "$\longrightarrow$" operator can also be used in *prefix* form with parenthesis (e.g., $(\longrightarrow)(A\,B) \equiv A \longrightarrow B$).

Function implementation: A function implementation is expressed with "=", with the function body at the right of "=". The arguments are represented within parenthesis at the left of "=" and they follow function signature definitions, so the previous implementation of $f \in F$ is expressed as $f(a, b) =$ '$body$', and consequently $a \in A$ and $b \in B$ are implicitly assumed.

Function application: In functional programming, functions are considered *first-class citizens*, in the sense that functions are just values of a certain *type* (or *signature*) that can be passed to other functions. Also, operator "$\longrightarrow$" is right-associative and function application is left-associative, so functions with several inputs can be subject to partial application. For example, a function F receiving two arguments $F :: A \longrightarrow B \longrightarrow C$ can be thought as a function receiving one argument of type A and returning a function of type $(B \longrightarrow C)$, $F :: A \longrightarrow (B \longrightarrow C)$.

To express the independence of a specific function implementation on an input parameter, an underscore "_" is used. Regarding the previous example, $f(a, \_) =$ '$body$' means that the input value of type B is discarded.

Function composition: Two functions can be composed to form a third function as long as the output type of the first function matches the input type of the second function. The operator "$\circ$" in infix form receives the first function to be applied on the right and the second on the left, and has a signature

$$\circ :: (B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow (A \longrightarrow C),$$

and different equivalent syntax, e.g., if $f \in (A \longrightarrow B)$ and $g \in (B \longrightarrow C)$ and $h \in (A \longrightarrow C)$ its composition,

$$g \circ f \equiv (\circ)gf \equiv g(f) \equiv h.$$

The first and second expressions correspond to *infix –in between* arguments– and *prefix –preceding* arguments– notations, respectively.

Lambdas: Lambdas are anonymous functions present in a function implementation. A *lambda* with input arguments $arg_1, arg_2, \ldots$, and implementation $body$ is defined as $\lambda(arg_1, arg_2, \ldots)(body)$. Lambdas are values of a type or *signature* that can be inferred from the types of $args$ –which can derived from the body in which the lambda is expressed– and the return type of the $body$ definition.

6.1.1.3 Functors

A functor F consists of a *type parametrization* (or a *type constructor*), and a function $FMap$. Type constructor of functor F defines (":::") a new type $F\,A$ from a primordial type A.

$$F ::: A \longrightarrow FA.$$

It is often said that the type A is *embellished* by a functor F to the type $F\,A$.

Function $FMap$ maps (or *lifts*) functions between two types to functions between respective embellished types,

$$FMap :: (A \longrightarrow B) \longrightarrow (FA \longrightarrow FB),$$

Functor rules: In order for the type constructor together with $FMap$ to qualify as a Functor, first, $FMap$ definition must preserve *identity*,

$$FMap(Id) = Id,$$

i.e., the identity function $Id :: X \longrightarrow X$ such that $Id(a) = a$, has to be mapped to the identity function acting on embellished types FX, $Id :: FX \longrightarrow FX$ that satisfies $Id(fa) = fa$.

Secondly, $FMap$ must preserve function composition between any two composable functions f and g,

$$FMap(f \circ g) = FMap(f) \circ FMap(g).$$

6.1.1.4 Monad pattern

One of the definitions of the monad pattern (see Section 1.2.1.1) is based on three definitions in terms of a functor and two functions that must obey particular rules, as explained next.

Type constructor: As said, a functor is a type parametrization to be instantiated by a type, so the values of the new type are a form of encapsulation of the values of the original type X. A monadic *type constructor* defines a monadic type $M\,X$ from other type X. Values of type $M\,X$ are *monadic values* that encapsulate values of the primitive type X.

Unit function: Together with a functor, an implementation of a *Unit function* is required, which returns monadic values from values of the original type,

$$Unit :: X \longrightarrow M\,X.$$

Bind function: Also, an implementation of a *Bind* function with signature

$$Bind :: M\,X \longrightarrow (X \longrightarrow M\,Y) \longrightarrow M\,Y,$$

needs to be provided. Specifically, $Bind$ is a function that receives a monadic value $M\,X$ and a *continuation* function with signature $(X \longrightarrow M\,Y)$. Internally, the $Bind$ implementation unwraps the input monadic value and passes the contained value to the continuation, so a new monadic value of type $M\,Y$ is returned. The $Bind$ function is alternatively expressed using the infix operator ">>=".

Monad rules: In particular, $Bind$ and $Unit$ implementations associated to a type constructor have to satisfy three *monad rules* in order for them to qualify as a monad.

First, *Unit* is a left-identity for *Bind*

$$Bind(Unit(a), \lambda(x)(f(x))) \equiv f(a), \tag{6.1}$$

Secondly, *Unit* is a right-identity for *Bind*

$$Bind(ma, \lambda(x)(Unit(x))) \equiv ma, \tag{6.2}$$

Third, *Bind* is associative

$$Bind(Bind(ma, f), g) \equiv Bind(ma, \lambda(x)(Bind(f(x), g))). \tag{6.3}$$

Previous rules provide a fixed framework that permits composability of any pair of functions that return monadic values $M\,X$ and $M\,Y$, and will be particularly useful when composing execution contexts and asynchronous computations.

Monad patterns do not only represent a generic abstraction layer for function composition; they are also useful for proper encapsulation of side-effects (see Section 1.2.1.1). In this sense, it will be exposed how some features of the relaxed execution model –e.g., executor composition and hierarchical task partitioning– can be expressed in terms of monadic patterns.

6.1.2 Data structures

This section explains how common data structures and related functions to be used in subsequent developments can be expressed from a functional programming approach. These data structures and functions are simply enough to be typically built-in types present in standard libraries of many functional languages. As said, the specific implementation details of these functions and subtleties concerning type constraints are irrelevant and omitted for the sake of brevity and simplicity.

6.1.2.1 Pair

A *pair type* is a product type of any two types X and Y, $X \times Y$. Its values are expressed as $\langle x, y \rangle$, where $x \in X$, $y \in Y$.

Element access: Fst and Snd functions act on pairs (returning the *first* and the *second* element, respectively) and present the following signatures and definitions:

$$\begin{aligned} Fst :: X \times Y &\longrightarrow X \\ Fst(\langle a, b \rangle) &= a \\ Snd :: X \times Y &\longrightarrow Y \\ Snd(\langle a, b \rangle) &= b. \end{aligned} \tag{6.4}$$

6.1.2.2 Set

A data type representing a *Set* data structure that stores unique values of type X without any particular order is referred as $Set\,X$. Values of type $Set\,X$ are expressed within brackets –e.g., $\{x_1, x_2, ...\}$–. For implementation performance reasons, some functions applied on $Set\,X$ values require type X to be a *totally ordered* data type. This constraint on X is not specified on the following function signatures acting on sets for the sake of clarity, as the implementation details with regard to algorithmic complexity are out of the scope of the discussion.

Set composition: To compose two sets of same type, the binary infix operator ":" is used. The fact that set structures do not refer to any order makes ":" commutative on any pair of sets,

$$\begin{aligned} : \quad &:: Set\,X \longrightarrow Set\,X \longrightarrow Set\,X \\ &a : b = b : a \quad \forall a, b \in Set\,X. \end{aligned} \tag{6.5}$$

Note that the empty set $\{\}$ is the identity element of the ":" operator.

Function $Singleton$ creates a singleton set from an element,

$$\begin{aligned} Singleton &:: X \longrightarrow Set\,X \\ Singleton(a) &= \{a\}, \end{aligned} \tag{6.6}$$

enabling element insertion into a Set.

Map: A function *map* acting on a function $F :: X \longrightarrow Y$ and a Set $Set\,X$ presents the signature

$$Map :: (X \longrightarrow Y) \longrightarrow Set\,X \longrightarrow Set\,Y,$$

and creates a secondary Set $Set\,Y$ by mapping via $f \in F$ the images of the elements of the input Set. Note that the output Set could be smaller than the input Set if $f(x_1) = f(x_2)$ for some pair x_1, x_2 in the input Set and $f \in F$.

Filter: Similarly, a function *filter* receives a predicate on X and a $Set\,X$ and returns another Set including the elements that satisfy the predicate, having a signature

$$Filter :: (X \longrightarrow Bool) \longrightarrow Set\,X \longrightarrow Set\,X.$$

Fold: A function to *fold* or *reduce* a $Set\,Y$ into a single value of type X has a signature:

$$Fold :: (X \longrightarrow Y \longrightarrow X) \longrightarrow X \longrightarrow Set\,Y \longrightarrow X, \tag{6.7}$$

where the first argument $(X \longrightarrow Y \longrightarrow X)$ is a function that returns an element X from a pair X and Y, the second argument X represents the type of the initial value, and the third argument $Set\,Y$ represents the set to be reduced.

6.1.2.3 List

A $List$ is a functor that, for a type X, creates a type $List\,X$ representing an ordered set of X elements. Its definition is recursive,

$$List\,X :: Void + X \times List\,X,$$

meaning that a list is either empty ($Void$) or has a head element coupled ($\times$) with another list representing the tail. Specifically, $Void$ type is a special type with a single possible value "[]". Values of type $List\,X$ are expressed with brackets –e.g., $[x_1, tail]$–.

Although lists can store repeated elements, the way lists are used in the following discussion consists on storing different elements in a specific ordering, so a list in this context can be viewed just as a set endowed with a specific ordering.

In analogy to the Set structure, functions

$$Map :: (X \longrightarrow Y) \longrightarrow List\,X \longrightarrow List\,Y,$$

$$Filter :: (X \longrightarrow Bool) \longrightarrow List\,X \longrightarrow List\,X,$$

$$Fold :: (X \longrightarrow Y \longrightarrow X) \longrightarrow X \longrightarrow List\,Y \longrightarrow X,$$

and the *infix concatenation* operator ":"

$$: \quad :: List\,X \longrightarrow List\,X \longrightarrow List\,X$$

are assumed to exist.

Additionally, $Head$ and $Tail$ functions are defined as:

$$\begin{aligned} &Head :: List^*\,X \longrightarrow X \\ &Head([x, tail]) = x \\ &Tail :: List\,X \longrightarrow List\,X \\ &Tail([x, ...]) = [...]. \end{aligned} \tag{6.8}$$

The type $List^*X$ refers to non-empty lists. In the following, $Head$ receiving a general list with type $List\,X$ is assumed to perform an intermediate pattern matching to assert that $List\,X$ is convertible to $List^*\,X$, which will be omitted for simplicity.

6.1.2.4 Stream

A $Stream$ is a functor that embellishes a type X to represent an infinite ordered sequence of elements of type X,

$$Stream\,X :: X \times Stream\,X.$$

Similarly to $List$, $Head$ and $Tail$ functions return the head and the rest of the stream, respectively,

$$\begin{aligned} &Head :: Stream\,X \longrightarrow X \\ &Tail :: Stream\,X \longrightarrow Stream\,X. \end{aligned}$$

The fact that a stream abstracts an infinite sequence of elements does not imply that its manipulation is not practical in a real computer program. Functional-based *lazyness* and recursion characteristics permit the manipulation of these infinite abstractions in a real finite computer.

6.1.2.5 Tree

A tree structure stores elements of type X into a hierarchy accessible from one root node. The associated $Tree$ data type can be defined as a recursive sum type

$$Tree :: X + Set\,Tree, \tag{6.9}$$

which means that a particular $tree \in Tree$ is either a leaf node storing a value of type X or a Set of trees –a non-leaf node that encompasses other $tree$ values–.

6.2 Functional relaxed execution model

In this section, the previously exposed functional concepts are applied to translate most of the ideas presented in Chapters 3 and 4 into a functional programming model.

First, Section 6.2.1 exposes how executor composition can be used to generate a set of hierarchical execution paths. Next, Section 6.2.2 presents how asynchronous computations can be composed, and in particular how the execution of a list or a stream of dependent tasks can be represented in the context of an unfolder executor. Finally, in Section 6.2.3, run-time scheduling concepts are introduced and joined with previous concepts. Particularly, scheduling functions will collapse a subset of execution paths –representing abstract execution possibilities– into particular paths that instantiate real executions.

6.2.1 Execution path composition

This section exposes, by means of a functional language, how a set of execution paths associated to a task and a particular executor are generated. As explained in previous chapters, executor composition and specific user definitions lead to *execution path expansions* (see Section 3.2.1). According to the STSE model, it is explained how executor composition endowed with monadic patterns enable *executor path composition*, from which a rich ensemble of execution paths are expanded by user definitions.

Following the naming of previous chapters, a type EX categorizes executors and regarding a function acting on executor-typed values, a type constraint $(Bottom \Rightarrow EX)$ is used to restrict the function application only to bottom executors values. Equivalently, $(Mapper \Rightarrow EX)$ and $(Unfolder \Rightarrow EX)$ clauses force the type EX in the following expression to be a mapper executor and unfolder executor only, respectively.

6.2.1.1 Burst bottom executor paths

An **e***xecution* **f***eature* is represented by a type EF whose values represent capabilities exposed within an execution context represented by a bottom executor. For example, features of a bottom executor may refer either to *(1)* architectural features of the processing device it is referring to –e.g., floating point units, instruction set extensions, etc.– and *(2)* software features –e.g., GPU-specific libraries, DVFS API, etc.–. These features are defined in a function $Features$, which maps every instantiable bottom executor in the system to a *Set* of execution features:

$$Features :: (Bottom \Rightarrow EX)EX \longrightarrow Set\,EF. \tag{6.10}$$

In the STEEL-PM implementation, these functions are defined at install-time for all the processing devices of the target platform.

In addition, type T is used to represent tasks that can be run on any executor. Values of T represent user-defined task identifiers, so with respect to Chapter 4 definitions, T can be thought as the `app::impl` enumeration type.

The $Allowed$ function represents these constraints, and it is a predicate that depends on a task t and an *execution feature*, with function signature

$$Allowed :: T \longrightarrow EF \longrightarrow Bool. \tag{6.11}$$

Its implementation is user-defined, and it is equivalent to the template constraint preceding a user-defined `kernel` class (see Section 4.2.2).

As a side note, regarding that different execution features could simultaneously characterize a task execution, the composition of simultaneous features is also considered a feature. For instance, when single and double floating-point processing capabilities are available and two sets of vector extensions are also available, the four combinations between floating-point capabilities and vector extensions are implicitly considered as elements of EF type. The construction of a set of this kind of composite features from other basic features is omitted from the sake of brevity.

From the previous definitions, a $Burst$ function is defined as a map that generates a Set of all possible execution features allowed for a given task $t \in T$ running on an executor constrained to be a bottom executor $(Bottom \Rightarrow EX)$. It presents a signature

$$Burst :: (Bottom \Rightarrow EX)EX \longrightarrow T \longrightarrow T \times Set\, EF, \tag{6.12}$$

and definition,

$$Burst(e, t) = \langle (t, Filter(Allowed(t), Features(e)) \rangle. \tag{6.13}$$

In other words, $Burst(e, t)$ returns a set of *execution possibilities* or *paths* that the runtime is free to take when a task t is running on an bottom executor e.

6.2.1.2 The *Burst* Monad

Any executor hierarchy spans a set of execution possibilities for a given task $t \in T$. This set of hierarchical execution possibilities can be modeled as a tree structure of type $ExTree$. Values of type $ExTree$ can be defined as a particular tree structure (see Definition 6.9) with *execution features* representing leaf nodes, so the $ExTree$ type can be defined as a recursive sum type in the form

$$ExTree :: EF + Set\, ExTree. \tag{6.14}$$

The infix concatenation operator ":" can be particularized to compose trees from EF values and other trees:

$$\begin{aligned}
: \quad &:: EF \longrightarrow Set\, ExTree \longrightarrow Set\, ExTree \\
&\quad ef : tree = Singleton(ef) : tree \\
: \quad &:: Set\, ExTree \longrightarrow EF \longrightarrow Set\, ExTree \\
&\quad tree : ef = ef : tree \\
: \quad &:: ExTree \longrightarrow ExTree \longrightarrow ExTree \\
&\quad a : b = \{a, b\} \\
&\quad b : a = a : b,
\end{aligned} \tag{6.15}$$

where the operator ":" in the right-hand side of the first definition refers to the Set concatenation operator (see Definition 6.5).

From these definitions, recalling the Monad triple in Section 6.1.1.4, a *Burst* Monad associated to a task type T can be first defined from a *type constructor* B that embellishes type T with a tree type $ExTree$

$$B ::: T \longrightarrow T \times ExTree. \tag{6.16}$$

Secondly, a $Unit_B$ function

$$\begin{aligned} Unit_B :: T &\longrightarrow T \times ExTree \\ Unit_B(t) &= \langle t, \{\} \rangle, \end{aligned} \tag{6.17}$$

and finally a $Bind_B$ function, receiving a monadic value mt and a continuation f

$$\begin{aligned} &Bind_B :: T \times ExTree \longrightarrow (T \longrightarrow T \times ExTree) \longrightarrow T \times ExTree \\ &Bind_B(mt, f) = \langle Fst(f(Fst(mt))), Snd(mt) : Snd(f(Fst(mt))) \rangle. \end{aligned} \tag{6.18}$$

In particular, regarding to the first monad law 6.1,

$$\begin{aligned} Bind_B(Unit_B(a), \lambda(x)(f(x))) &= \langle Fst(f(Fst(Unit_B(a)))), \\ &\quad Snd(Unit_B(a)) : Snd(f(Fst(Unit_B(a)))) \rangle \\ &= \langle Fst(f(a)), \{\} : Snd(f(a)) \rangle \equiv f(a) \end{aligned}$$

so 6.1 is satisfied.

With respect to the second monad law (6.2),

$$\begin{aligned} Bind_B(mt, \lambda(x)(Unit_B(x))) &= \langle Fst(Unit_B(Fst(mt))), \\ &\quad Snd(mt) : Snd(Unit_B(Fst(mt))) \rangle \\ &= \langle Fst(Unit_B(t)), \quad Snd(mt) : \{\} \rangle \\ &= \langle t, Snd(mt) \rangle \equiv mt, \end{aligned}$$

so it is also satisfied.

Regarding the third monad law, the left hand side of the equivalence in (6.3) expands into

$$Bind_B(\langle Fst(f(Fst(ma))), Snd(ma) : Snd(f(Fst(ma))) \rangle, g).$$

Aliasing $f(Fst(ma)) \equiv \alpha \in T \times ExTree$ for clarity, the previous expression develops into

$$\begin{aligned} &Bind_B(\langle Fst(\alpha), Snd(ma) : Snd(\alpha) \rangle, g) = \\ &\langle Fst(g(Fst(\alpha))), Snd(ma) : Snd(\alpha) : Snd(g(Fst(\alpha))) \rangle. \end{aligned} \tag{6.19}$$

Similarly, the right hand side of the equivalence in (6.3) expands into

$$\langle Fst(Bind_B(f(Fst(ma)), g)), Snd(ma) : Snd(Bind_B(f(Fst(ma)), g)) \rangle.$$

Applying the α alias, it transforms into

$$\begin{aligned} &\langle Fst(Bind_B(\alpha, g)), Snd(ma) : Snd(Bind_B(\alpha, g)) \rangle = \\ &\langle Fst(g(Fst(\alpha))), Snd(ma) : Snd(\alpha) : Snd(g(Fst(\alpha))) \rangle, \end{aligned} \tag{6.20}$$

which is equal to the left hand side expression in (6.19). Consequently, it can be concluded that $Bind_B$ (6.18) and $Unit_B$ (6.17) definitions for the type constructor (6.16) satisfy the three monad laws (Definitions 6.1, 6.2 and 6.3).

As a side note, from a Category Theory reasoning, it could be demonstrated that $Burst$ is a monad by simply noting that it follows the same structure as the well-known *Writer monad* [67]. Specifically, note that values of $ExTree$ type that embellish the original type, together with the previous concatenation operator ":" definition and the identity element "{}", form a monoid [109].

Paths from a bottom executor: The burst function for bottom executors (Definitions 6.12 and 6.13) partially applied to a bottom executor e, $burst(e)$, is a function with signature $T \longrightarrow T \times Set\,EF$ that can be seen as a monadic function $T \longrightarrow T \times ExTree$, because $T \times Set\,EF$-typed values can be converted according to Definition 6.14 into $T \times ExTree$-typed values. Thus, signature 6.12 can be redefined as a function that returns monadic values

$$Burst :: (Bottom \Rightarrow EX)EX \longrightarrow T \longrightarrow T \times ExTree. \tag{6.21}$$

Hence, the bursted execution paths that represent all the execution options when a task $t \in T$ is associated to a bottom executor can be represented by a monadic value of type $T \times ExTree$. Specifically, the return value consists of a pair of a task and a shallow tree composed of the user-allowed execution features exposed by the $Features$ function applied on that bottom executor.

6.2.1.3 Burst mapper executor paths

The previous definition of the *Burst* Monad enables the composition of $Burst$ functions of a set of bottom executors. This composition results into an aggregation of execution paths from a set of bottom executors encompassed by a mapper executor. Consequently, the burst function $Burst$ restricted to mapper executors $(Mapper \Rightarrow EX)$, which encompass a set of executors defined in a function $Mapped$

$$Mapped :: (Mapper \Rightarrow EX)EX \longrightarrow Set\,EX, \tag{6.22}$$

with signature

$$Burst :: (Mapper \Rightarrow EX)EX \longrightarrow T \longrightarrow T \times ExTree. \tag{6.23}$$

Its implementation can be defined from previous $Fold$ definition (Definition 6.7), particularized for $X \equiv T \times ExTree$ and $Y \equiv (T \longrightarrow T \times ExTree)$.

Specifically, the first argument type $(X \longrightarrow Y \longrightarrow X)$ translates into

$$T \times ExTree \longrightarrow (T \longrightarrow T \times ExTree) \longrightarrow T \times ExTree,$$

which matches the signature of $Bind_B$ (Definition 6.18). The second argument is the initial monadic value built from $Unit_B$, and the third argument is the Set of $Burst$ functions to be folded, which is the result of partial application of $Burst$ on each of the elements of $Set\,EX$

$$Burst(e, t) = Fold(Bind_B, Unit_B(t), Map(Burst, Mapped(e))), \tag{6.24}$$

The result of this function is an execution tree of paths composing the nodes bursted by the mapped executors in $Mapped(e)$. Note that implementations 6.13 and 6.24 make function $Burst$ polymorphic (behavior depending on $Mapper$ or $Bottom$ type constraints). Thus, the recursive definition 6.24 allows other EX types that qualify as $Mapper$ included in $Mapped(e)$ (i.e., not necessarily $Bottom$).

6.2.1.4 Burst unfolder executor paths

By definition, an unfolder executor delegates a set of tasks $Set\,T$ to any executor with type EXD defined in $Delegate$ function,

$$Delegate :: (Unfolder \Rightarrow EX)EX \longrightarrow EXD. \tag{6.25}$$

To represent the execution paths bursted from an unfolder executor, the execution paths for each task T in a $Set\,T$ have to be composed.

Specifically, a function $Burst$ associated to an unfolder executor has a signature:

$$Burst :: (Unfolder \Rightarrow EX)EX \longrightarrow T \longrightarrow T \times ExTree. \tag{6.26}$$

In order to calculate the bursted execution paths resulted from a task partition on a context of an unfolder executor, provided a user-defined function

$$MakeTaskList :: T \longrightarrow List\,T, \tag{6.27}$$

A function $BurstTaskList$ presents a signature (with the $(Unfold \Rightarrow EX)$ constraint implicitly assumed)

$$BurstTaskList :: EX \longrightarrow T \longrightarrow (T \longrightarrow List\,T) \longrightarrow T \times ExTree, \tag{6.28}$$

and implementation

$$\begin{aligned} BurstTaskList(e,t,f) = Fold(&\lambda(mt',t')(Bind_B(mt', Burst(Delegate(e),t'))),\\ &Unit_B(t),\\ &f(t)), \end{aligned} \tag{6.29}$$

where the first argument of $Fold$ has signature $(T \times ExTree \longrightarrow T \longrightarrow T \times ExTree)$, the second argument is a monadic value $Unit_B(t)$, and the third argument is the $List\,T$ to be folded.

If the user wants to expose a set of valid task partitioning possibilities or *parameters* (with type PP) for a given task, so that the runtime is free to choose among a set of partitions, a function $PartitionParameters$ with signature

$$PartitionParameters :: T \longrightarrow Set\,PP \tag{6.30}$$

must first be implemented. Then, for each valid pair of a T and a PP, a function to generate a set of tasks from partition parameters, with signature

$$Partition :: T \longrightarrow PP \longrightarrow List\,T \tag{6.31}$$

must also be provided.

Another version of $BurstTaskList$, $BurstTaskListParam$, receiving a partition parameter and composing the result monad with other monadic value, would have a signature

$$BurstParamCompose :: EX \longrightarrow T \longrightarrow T \times ExTree \longrightarrow PP \longrightarrow T \times ExTree, \tag{6.32}$$

and can be implemented as

$$\begin{aligned} BurstParamCompose(e,t,mt,pp) = Bind_B(&mt,\\ &BurstTaskList(e,t,\lambda(t')(Partition(t',pp)))). \end{aligned} \tag{6.33}$$

With previous definitions, an final implementation for $Burst$ targeting unfolder executors (Definition 6.26), can be implemented as

$$Burst(e,t) = Fold(BurstParamCompose(e,t), Unit_B(t), PartParameters(t)), \tag{6.34}$$

which returns a monadic value representing the bursted execution paths for a task t being executed by an unfolder executor.

In definition 6.34, the $Fold$ function presents a signature

$$(B\,T \longrightarrow X \longrightarrow B\,T) \longrightarrow B\,T \longrightarrow Set\,X \longrightarrow B\,T,$$

with

$$B\,T \equiv T \times ExTree,$$

and

$$X \equiv PP.$$

Note that partial application of $BurstParamCompose$ results into a function with signature $B\,T \longrightarrow PP \longrightarrow B\,T$, thus correctly matching the signature of the first argument of $Fold$.

6.2.1.5 Summary

The *Burst* Monad consisting on the triple *type constructor* (6.16), $Unit_B$ function (6.17), and $Bind_B$ function (6.18), enabled the composition of a set of polymorphic $Burst$ functions, for each kind of executor bottom executor, mapper executor and unfolder executor, that return a monadic value consisting on a task identifier of type T and a tree of execution paths $ExTree$.

In Table 6.1, the associated references to $Burst$ and required user definition for each executor type are summarized. Note that $Burst$ function for mapper executors does not require additional user-definitions, which is illustrated by the fact that task featurization for mapper executors is implicit and derived from the lower-level executors (See section 4.2.2.5).

	Burst Signature	*Burst Implementation*	*Required user-defined function signatures*
bottom executor	(6.21)	(6.13)	(6.11)
mapper executor	(6.23)	(6.24)	
unfolder executor	(6.26)	(6.34)	(6.30, 6.31)

Table 6.1: References to *Burst* function definitions and signatures of required user-defined implementations.

As a final note, recall that in Section 3.1.3 *task reimplementations* are also possible user-defined relaxations to be done in the context of unfolder executors. Despite that, in this section only *partitions* resulted into lists of tasks were mentioned, because it is the most general scenario in an unfolder executor: a task reimplementation results into a particular singleton-list case. In the following, whenever *task partitioning* is mentioned, *task reimplementation* is also implicitly considered.

6.2.2 Asynchronous and data-flow computations

Asynchronous call abstractions and Continuation-Passing-Style patterns have provide great utility to chain computations irrespective of their termination [108, 141]. As mentioned in Section 1.2.2, these patterns are used to enable high levels of parallelism in several runtime systems.

With this regard, the *future* abstraction, implemented as a built-in feature in many programming languages and frameworks [138, 59, 49, 117, 70], is used within STEEL-RT to compose asynchronous executions of tasks seamlessly. In particular, this abstraction is completely STEEL-RT-internal, so the user does not need to know any interface of *future* objects –and not even what a *future* is–.

In previous section, tasks were only elements by which a mere *execution feature filtering* (Definition 6.11) on the features exposed by a bottom executor (Definition 6.10) was done.

In practice, and according to STEEL-PM, a task identifier is also associated to a *kernel* function that returns output data of type D from input data of type D.

$$Kernel :: D \longrightarrow D, \tag{6.35}$$

whose implementation is user-defined. In relation to section 3.1.3, values of type $Kernel$ are equivalent to *task instantiations*.

Moreover, in order for a set of tasks to be concurrently distributed across several execution contexts –so that task parallelism is enabled–, a way to asynchronously dispatch the tasks to different contexts has to be provided.

In this sense, a *future* represents an abstraction of a computation that has not necessarily been completed. A *future type* is associated to another type and wraps its values into *future values*. In order for a kernel instantiation to be asynchronous, the previous $Kernel$ signature has to be upgraded to a *futurized* kernel with signature

$$AsyncKernel :: D \longrightarrow FutD, \tag{6.36}$$

in which $FutD$ represents a *future type* derived from D type.

Contrary to function 6.35, $AsyncKernel$ function is an abstraction automatically constructed by STEEL-RT at compile-time.

In practical scenarios, kernel instantiations receive input data to be returned by other instantiations, so there are dependency relations that serialize the executions. In order to maintain the asynchronous properties across kernel compositions –i.e., if f and g are inter-dependent asynchronous kernels, its composition $h = g(f)$ must also be asynchronous–, a way to systematically compose futurized kernels has to be provided.

6.2.2.1 *Future* monad

The *Future* monad enables composition of asynchronous computation. Type $Fut\,D$ in signature 6.36 actually represents a type constructor for the future monad Fut that lifts a data type D to a *future type* $Fut\,D$. In general, the *future* monad is based on a Functor that embellishes any type X:

$$Fut ::: X \longrightarrow FutX. \tag{6.37}$$

A $Unit_F$ function exposes a signature

$$Unit_F :: X \longrightarrow FutX, \tag{6.38}$$

and a $Get :: FutX \longrightarrow X$ function could also be provided to retrieve the value encapsulated by a *future* monadic value. In general, application of Get on a future will block until the function responsible for computing the wrapped result of type X has finished.

An actual implementation for $Unit$ would wrap a value $x \in X$ into a *ready future* monadic value –i.e., by definition, $Get(Unit(x))$ never blocks for any x–. A *ready future* is also said to be *resolved*.

An implementation of $Bind_F$ function with signature

$$Bind_F :: FutX \longrightarrow (X \longrightarrow FutY) \longrightarrow FutY, \tag{6.39}$$

unwraps a future representing the result of an unfinished computation and injects it in the continuation $(X \longrightarrow FutY)$, so a new future $FutY$ is returned.

$AsyncKernel$ with signature 6.36, can be defined internally in STEEL-RT from a user-defined implementation for $Kernel$ and $Unit_F$,

$$AsyncKernel = \lambda(d')(Unit_F(Kernel(d'))).$$

6.2.2.2 Function composition

From previous definitions, the *futurized* versions of functions $X \longrightarrow Y$ and $Y \longrightarrow Z$ result into $f \in X \longrightarrow FutY$ and $g \in Y \longrightarrow FutZ$, and the bind function of the future monad provides a way to seamlessly compose an asynchronous function $h \in X \longrightarrow FutZ$, from f and g:

$$h(x) = Bind_F(f(x), g).$$

A function $j(y_1, y_2)$ with signature

$$(Y_1 \longrightarrow Y_2 \longrightarrow FutZ)$$

can be composed with a set of functions

$$k_1, k_2 \in K_1, K_2 \equiv (X_1 \longrightarrow FutY_1), (X_2 \longrightarrow FutY_2)$$

to form a function m with signature

$$X_1 \longrightarrow X_2 \longrightarrow FutZ$$

by nesting $Bind_F$ calls:

$$m(x_1, x_2) = Bind_F(k_1(x_1), \lambda(y_1')(Bind_F(k_2(x_2), \lambda(y_2')(j(y_1', y_2'))))).$$

Application of m immediately returns a future of the result, and this composition can be easily generalized to multiple arguments $X_1, X_2, ..., X_n$ by deeper $Bind_F$ nesting to compose multiple functions.

This composition of futures represents a *when-all* abstraction, which is a way to encapsulate a set of pending results that can be computed concurrently. To this encapsulation, a *continuation function* (in previous example j) that represents a computation that requires all the values to proceed, can be attached. In general, the return value of this function, built from composition of n k_i functions and j function that gathers the results, returns an abstract future that becomes ready when all the futures $FutY_1, ..., FutY_n$ become ready.

A similar *when-any* abstraction represents an abstract computation that encompass a set of results but only requires a single ready future in order to be ready –i.e., it becomes resolved *when any* of the encapsulated futures is resolved–. Its implementation details are out of the scope of this discussion, as asynchronous non-redundant tasks in DAG structures can be sufficiently modeled with *when-all* abstraction.

6.2.2.3 Data guarding

The process of data guarding seen in the previous chapter consisted of guarding a data handle –i.e., building a *data guard* from a data argument– and providing ways of accessing to subdata elements (see Section 4.2.4). The returned subdata elements consist of representations of data subregions that provide protection against concurrent access.

As a side note, for the sake of simplicity, the previous scenario in which a data may depend on multiple input and output arguments is reduced into a data that just accepts a single datum D and outputs another datum of the same type. Note that this simplification is just syntax-related for the sake of clarity and generality is not sacrificed: no restriction was imposed to type D, so a set of multiple arguments can be thought of a list of elements belonging to D. Also, the possibility of expressing task dependencies by fine-grain synchronization characteristic of data-flow pattern is still intact but hidden in the abstract data type D.

$Guard\, D$ represents the state of the guard container that changes when access to a subdata element is retrieved and consequently passed to a task.

In particular, four helper functions related to $Guard\,D$ type are assumed to exist. First, a function

$$Getd :: T \longrightarrow Guard\,D \longrightarrow Fut\,D \times Guard\,D, \tag{6.40}$$

returns a subdata element contained in $Guard\,D$ wrapped in a future that represents the input dependence of a task T. Secondly, a function

$$Setd :: Guard\,D \times Fut\,D \longrightarrow Guard\,D \tag{6.41}$$

inserts a data future representing a pending result of a task execution into the state guard. Finally,

$$CreateGuard :: D \longrightarrow Guard\,D \tag{6.42}$$

initializes a $Guard\,D$ state from a datum, and

$$ReturnGuard :: Guard\,D \longrightarrow Fut\,D \tag{6.43}$$

aggregates the pending output data that has been stored in a guard by $Setd$ function and retrieves a future that represents a *when-all* aggregation of them.

As a side note from Category Theory, $Guard$ functor together with $Setd$ and $Getd$ functionalities can be grouped as an instance of the well-known *State monad* [67], which is designed to provide general composition rules for functions that read and modify a generic state. In this sense, previous and ongoing developments can be translated in terms of a $State$ functor and specific $Unit$ and $Bind$ functions, but formulation in terms of $Guard\,D$ and $Getd$ / $Setd$ functions is preferred to remark the similarity with the STEEL-API functions.

6.2.2.4 Finite list of tasks

Section 6.2.3 will expose how functions $ExecuteAsync$ with signature

$$ExecuteAsync :: EX \longrightarrow T \longrightarrow D \longrightarrow Fut\,D \tag{6.44}$$

abstract the asynchronous execution of a task into any executor context. Meanwhile, it is assumed that these functions partially applied to pairs $e \in EX$ and $t \in T$ return an $AsyncKernel$ function object with type $D \longrightarrow Fut\,D$.

Since the goal of this section is to express how a set of tasks are executed asynchronously, in the following it is exposed a specific implementation of $ExecuteAsync$ when EX is constrained to an unfolder executor.

By definition, an unfolder executor is always stacked on top of a delegate executor (see Section 3.1.2), and according to the present functional model it is defined in the implementation of a function

$$Delegate :: EX \longrightarrow EX. \tag{6.45}$$

Assuming a function implementation for $MakeTaskList$ (6.27), which returns a list of tasks generated after a task partition, then

$$\begin{aligned} &ExecuteAsync(ue, t, din) = \\ &\qquad ExecuteListAsync(Delegate(ue), MakeTaskList(t), CreateGuard(din)), \end{aligned} \tag{6.46}$$

where $ExecuteListAsync$ has a signature

$$ExecuteListAsync :: EX \longrightarrow ListT \longrightarrow Guard\,D \longrightarrow Guard\,D,$$

with recursive implementation

$$ExecuteListAsync(e, [t, tail], gd) = ExecuteListAsync(e, [tail], \mu(e, t, gd)). \quad (6.47)$$

Function μ returns an updated $Guard\,D$ that is passed to the next recursion and is defined as

$$\mu(e, t, gd) = Setd(\langle Snd(Getd(t, gd)), \\ Bind_F(Fst(Getd(t, gd)), ExecuteAsync(e, t))\rangle).$$

Finally, an end-case (empty list) implementation returns the $Guard\,D$ state unaltered,

$$ExecuteListAsync(\_, [], gd) = gd.$$

6.2.2.5 Infinite stream of tasks

Many applications are more conveniently expressed in terms of an infinite sequence of operations.

An unfolder executor context is not only used to unfold a finite list of children tasks, but also to unfold an infinite stream of tasks. Specifically, the user must implement a function $MakeStream$ with signature

$$MakeStream :: T \longrightarrow Stream\,T$$

which creates a stream of tasks, in which tasks are in general fed from data generated by previous tasks, and possibly from initial data encapsulated in a guard returned by $CreateGuard$.

In particular, $ExecuteStream$ presents a signature

$$ExecuteStream :: (Unfolder \Rightarrow EX)EX \longrightarrow T \longrightarrow D \longrightarrow Stream\,Fut\,D$$

which returns an infinite stream of futures. It can be implemented from a helper function $DataStream$ with signature

$$DataStream :: EX \longrightarrow StreamT \longrightarrow Guard\,D \longrightarrow Stream\,Fut\,D,$$

as

$$ExecuteStream(ue, t, din) = DataStream(Delegate(ue), \\ MakeStream(t), CreateGuard(din)).$$

Additionally, $DataStream$ can be implemented recursively as

$$DataStream(e, st, dg) = \langle Fst(\nu(e, Head(st), dg)), \\ DataStream(e, Tail(st), Snd(\nu(e, Head(st), dg)))\rangle,$$

where ν with signature $(EX \longrightarrow T \longrightarrow Guard\,D \longrightarrow Fut\,D \times Guard\,D)$, first calculates the output future

$$fdo = Bind_F(Fst(Getd(t, dg)), ExecuteAsync(e, t)),$$

and returns a $Fut\,D \times Guard\,D$ pair

$$\nu(e, t, dg) = \langle fdo, Setd(dg, fdo)\rangle.$$

Finally, the user is free to *channel* the returned $Stream\,Fut\,D$ in any way, and eventually finalizing the stream computation when a terminate condition is met.

6.2.2.6 Relation with STEEL API

Previous definitions provide a functional model for a typical task partition in the context of an unfolder executor. $CreateGuard$ function is a direct analogy of the *guard creation* mechanism explained in Section 4.2.4. The inner workings of $CreateGuard$, from the perspective of STEEL-RT, involve the wrapping of a subdata element into future values that also encapsulate the necessary information needed for data-hazards prevention.

Moreover, $Getd$ reflects the behavior of the member function `get` of a STEEL-PM data guard with a minor difference. In this case, $Getd$ is a simplified version, in the sense that it directly associates a task identifier with a future encapsulating its single input data. Contrary, in STEEL-API each argument of the task has to be encapsulated by respective guards, and the access to subdata elements encompassed by the guard is done by user-defined keys (see details in Sections 4.2.4 and 4.2.7).

On the other hand, in the simplified functional model just presented, an explicit $ReturnGuard$ function is needed to aggregate the partial data returned by children tasks into a future value that represents the result of the partitioned task, which is then returned to the caller of the unfolder executor. In STEEL-API, RAII semantics for data guards make this wrapping hidden from the user view.

Finally, the functional model for unfolding an infinite stream is not explicitly translated into *stream*-like structures in STEEL-API, but the imperative nature of the built-in language in which STEEL-PM is built upon (C++), makes the implementation of stream-like computations indistinguishable from the finite-partitioned task –i.e., both cases consist on an asynchronous delegation of tasks inside a loop in which tasks are unfolded and delegated–.

6.2.2.7 Summary

In this section it was exposed a *preliminary* implementation for $ExecuteAsync$ designed for unfolder executors in terms of the well-known future abstraction and user-defined $MakeTaskList$ function, together with additional helper functions $Getd$, $Setd$, $CreateGuard$ and $ReturnGuard$, whose functionality was related to explanations in Chapter 4. It was also exposed, for the sake of generality, a function $ExecuteStream$ in which task is not partitioned, but unfolded into a stream of future data elements.

The implementation $ExecuteAsync$ is said to be *preliminary* because it does not take into account eventual partition possibilities declared by the user. In this line, the next section exposes an implementation that does consider this. Also, $ExecuteAsync$ implementations will be defined for the rest of the executor kinds.

6.2.3 Scheduling and execution of tasks

In this section, previous developments regarding execution path creation and asynchronous computations are merged to define a set of $ExecuteAsync$ functions for each executor type. Equivalently to $Burst$ functions, $ExecuteAsync$ functions expose the same signature $EX \longrightarrow T \longrightarrow D \longrightarrow Fut\ D$ with only the EX type constrained for each executor type.

Moreover, a set of functions for each executor type $BottomSchedule$, $MapperSchedule$ and $UnfolderSchedule$ are used to model the internal workings of STEEL-RT in terms of scheduling. As explained in Section 3.2, the concept of *scheduling* employed in this development is *generalized* or *broaden* from the classic concept consisting on the mere *assignment of work to resources* (see Section 3.2.2). In this line, scheduling functions have the inverse effects of $Burst$ functions implemented in Section 6.2.1: for a specific executor-task pair, they reduce a set of execution possibilities generated by corresponding $Burst$ calls. In the particular case of the $MapperSchedule$ function, it will assign the execution of a task to an executor, hence recovering the classic *scheduling* semantics. On the contrary, $BottomSchedule$ could select one of the

allowed *bursted* execution features, while $UnfolderSchedule$ could pick a particular partition granularity or reimplementation.

6.2.3.1 Execution in a bottom executor

In relation to Section 4.2.2, a $RelaxKernel$ function corresponds to user-defined kernel instantiations meant to be run in bottom executors,

$$RelaxKernel :: T \longrightarrow EF \longrightarrow D \longrightarrow D. \tag{6.48}$$

During compilation, these functions are internally transformed to asynchronous functions with signature

$$RelaxAsyncKernel :: T \longrightarrow EF \longrightarrow D \longrightarrow Fut\, D. \tag{6.49}$$

where the output data type D is internally embellished at compile-time to a $Fut\, D$. Additionally, the second argument of type EF refers to the `param` appearing for example in Listings 4.9 and 4.10, whose value is meant to be set internally by a STEEL-RT scheduler.

In this line, the $BottomSchedule$ function represents this scheduling action, which outputs a specific execution feature EF from a shallow tree generated by the $Burst$ function (6.13).

$$BottomSchedule :: T \times ExTree \longrightarrow EF. \tag{6.50}$$

Then an $ExecuteAsync$ associated to a bottom executor can be defined with signature

$$ExecuteAsync :: (Bottom \Rightarrow EX)EX \longrightarrow T \longrightarrow D \longrightarrow Fut\, D, \tag{6.51}$$

and implementation

$$ExecuteAsync(e, t, din) = RelaxAsyncKernel(t, BottomSchedule(Burst(e, t)), din). \tag{6.52}$$

In summary, $ExecuteAsync$ returns a future to the result of the user-defined kernel $RelaxKernel$, whose execution is particularized by the result returned by calling

$$BottomSchedule(Burst(e, t)).$$

6.2.3.2 Execution in a mapper executor

Similarly, the $MapperSchedule$ function assigns a task to an executor encompassed by a mapper executor, based on the possibilities expanded by the associated $Burst$ function.

$$MapperSchedule :: T \times ExTree \longrightarrow Set\, EX \longrightarrow EX \tag{6.53}$$

An equivalent $ExecuteAsync$ particularized for mapper executors with signature

$$ExecuteAsync :: (Mapper \Rightarrow EX)EX \longrightarrow T \longrightarrow D \longrightarrow Fut\, D \tag{6.54}$$

is implemented as:

$$ExecuteAsync(e, t, din) = ExecuteAsync(\langle MapperSchedule(Burst(e, t), Mapped(e)), t\rangle, din). \tag{6.55}$$

In this case, the execution is simply forwarded to the executor selected by calling to $MapperSchedule$.

6.2.3.3 Execution in an unfolder executor

The scheduling function $UnfolderSchedule$ picks a specific list of tasks from the set of possibilities expanded by the corresponding $Burst$ function.

$$UnfolderSchedule :: T \times ExTree \longrightarrow Set\,List\,T \longrightarrow List\,T. \tag{6.56}$$

Then, $ExecuteAsync$ with signature restricted for unfolder executors

$$ExecuteAsync :: (Unfolder \Rightarrow EX)EX \longrightarrow T \longrightarrow D \longrightarrow Fut\,D \tag{6.57}$$

can be implemented as

$$\begin{aligned} ExecuteAsync(ue, t, din) = \\ & ExecuteListAsync(Delegate(ue), \\ & UnfolderSchedule(Burst(e, t), \eta(t)), \\ & CreateGuard(din)), \end{aligned} \tag{6.58}$$

where η generates all the possible partitions (in the form of a $Set\,List\,T$) for a task t:

$$\eta(t) = Map(Partition(t), PartitionParameters(t)).$$

In essence, Definition 6.58 applies $ExecuteListAsync$ (Definition 6.47) to a list that represents a specific partition (or to a singleton list representing a reimplementation), decided by $UnfolderSchedule$ based on the possibilities returned from $Burst$.

6.3 Summary

In the following (Section 6.3.1), the relation of a subset of functions previously listed with functions exposed in the STEEL-API is shown, stressing the moment –during compilation or execution– in which these functions are called.

Secondly, some limitations of the previous functional model are exposed in Section 6.3.2. It is worth noting that these limitations are exclusive of the functional model –in close relation with its *functional purity*–, and do not pose any problem from the implementation perspective in terms of the STEEL-PM implementation presented in the previous chapter, as it is not a purely functional implementation.

6.3.1 Relation with STEEL-PM

Among all the set of functions previously exposed, it is summarized next a subset of functions whose utility relates to what was explained in Chapters 3 and 4. These functions are classified by *(i)* their main usage, *(ii)* the implementer and *(iii)* the caller.

6.3.1.1 System – Executor matching

In Table 6.2, $Features$, $Mapped$ and $Delegate$ functions endow each of the executor types – bottom, mapper, unfolder– with their respective attributes.

In particular, function $Features$ is implemented as a template specialization of a parametrized function belonging to the runtime support template functions. Functions of this set are meant to be specialized to account for newer architectural characteristics of processing devices (see Figure 4.2), and defined from the detected hardware characteristics during installation. Eventual bottom executors to be instantiated at compile-time will then expose these characteristics as *execution features* to be considered in a user-defined kernel implementation of a task.

Functions $Mapped$ and $Delegate$ establish the relations between executors, fully characterizing the executor tree. Consequently, their *implementer* is the user if the executor tree is explicitly built by composing executor references from `deploy` calls. On the contrary, if the executor tree is automatically inferred by `deploy_main` call, the implementer is said to be the compiler, as these functions –hence the full executor tree– are derived from a metaprogram executed at compile-time.

Function	*Usage*	*Implementer*	*Caller*
$Features$ (6.10)	Device arch. detection	Installer	Compiler
$Mapped$ (6.22)	Executor arch. definition	User or compiler	Compiler
$Delegate$ (6.25)	Executor arch. definition	User or compiler	Compiler

Table 6.2: Related functions for matching system platform with executor properties and tree architecture.

6.3.1.2 Application development

Table 6.3 summarizes functions related to application development. In these functions, the *user* –or *application developer*– plays a role either as the implementer or as the caller.

First, $ExecuteAsync$ represents both the program entry point and the runtime binding between executors, in the sense that tasks flow across the executor tree by internal $ExecuteAsync$ calls. Its caller is the user in the program entry point case –when dispatching the main task to the top-most executor– and in the unfolder case –when dispatching tasks to a *delegate* executor–. On the contrary –dispatching a task from a mapper context–, then the runtime is the caller. In relation to the STEEL-API, calls to `execRef` (see Listing 4.18 in Section 4.2.3) perform an equivalent functionality.

Functions $Allowed$ and $PartitionParam(eters)$, in second and third rows, represent *featurization* options and they are resolved (or *called*) at compile-time. These functions return a set of execution possibilities to be resolved by the runtime and their definition is equivalent to the setting of the type `expanded` in Listings of Section 4.2.2, which is a type interpreted by the compiler.

Functions $RelaxKernel$ and $Partition$ are equivalent to the user-defined `run` functions defined within a `kernel` class, each representing a kernel to be run on an unfolder executor and a task partition (or reimplementation), respectively.

As explained in Section 6.2.2, the four remaining functions provide data guarding, partitioning, and fine-grained task-synchronization capabilities. These functions have clear correspondence to the API exposed in Section 4.2.4. For $ReturnGuard$ and $Setd$, the caller is marked as the *Runtime** and not the user because their equivalent STEEL-RT functions counterparts are *implicitly* triggered by the runtime. In other words, RAII-semantics and the internal mechanism of `get` interface in the objects returned by `handle` functions make the STEEL-API-equivalent counterparts of $ReturnGuard$ and $Setd$ functions invisible from user code.

6.3.1.3 Functions in runtime scheduling

In Table 6.4 STEEL-RT internal functions related to execution path generation (or *bursting*) and scheduling are shown.

First, $Burst$ function is polymorphic, with implementations depending on the executor type, and always resolved (or *called*) at compile-time. Hence, all the execution path possibilities for a given *task – executor* pair do not have to be computed at run-time each time a task is dispatched to an executor. Secondly, as explained in Section 6.2.3, scheduling functions for each executor type are resolved during execution to select one of the options generated by their corresponding $Burst$ function.

Function	*Usage*	*Implementer*	*Caller*
$ExecuteAsync$ (6.52,6.55,6.58)	Task execution	RT-developer	User/Runtime
$Allowed$ (6.11)	Task featurization	User	Compiler
$PartitionParam$ (6.30)	Task featurization	User	Compiler
$RelaxKernel$ (6.48)	Kernel execution	User	Runtime
$Partition$ (6.31)	Task unfolding	User	Runtime
$CreateGuard$ (6.42)	Data part. & sync.	RT-developer	User
$ReturnGuard$ (6.43)	Data part. & sync.	RT-developer	Runtime*
$Getd$ (6.40)	Data part. & sync.	RT-developer	User
$Setd$ (6.41)	Data part. & sync.	RT-developer	Runtime*

Table 6.3: Related functions during the application development process.

Function	*Usage*	*Implementer*	*Caller*
$Burst$ (6.13,6.24,6.34)	Generate execution paths	RT-developer	Compiler
$BottomSchedule$ (6.50)	Select execution feature	RT-developer	Runtime
$MapperSchedule$ (6.53)	Select executor	RT-developer	Runtime
$UnfolderSchedule$ (6.56)	Select partition / reimpl.	RT-developer	Runtime

Table 6.4: Related functions during runtime task scheduling.

6.3.2 Model limitations

The functional model presented in this chapter had the *sole* purpose of providing an alternative functional representation –yet using the same concepts when possible– to what was exposed in Chapters 3 and 4. However, from a stricter theoretical lens, this model presents several loopholes that should be highlighted. Specifically, some limitations of the model in terms of its *purity* (see Section 1.2.1.1) are exposed next. These limitations would need to be resolved in order to implement STEEL-PM in a *pure* functional language –e.g., Haskell–, to endow the current implementation with more *functional-like* features, or to permit the model to be subject to formal verification tools.

6.3.2.1 *Future* wrappers are not immutable

Section 6.2.2 exposed how tasks related by data dependencies encapsulated in *Guard* containers are synchronized in the STEEL model. The main purpose of these containers is to serve as concurrent-safe storage for subdata regions belonging to larger data. In particular, a basic interface via $Getd$ and $Setd$ was exposed, by which *future* objects wrapping subdata elements can be retrieved and injected in the $Guard\ D$ object as new tasks are delegated to a lower-level executor.

Despite this model represents with fair accuracy the task synchronization model followed by STEEL-RT, it is not purely functional. Specifically, *future* objects are wrappers of a shared and mutable state, and their usage clearly violates referential transparency: by design, the state encapsulated by a future –i.e., the result of a computation– depends on *time* –e.g., despite $Getd :: Guard\ D \longrightarrow Fut\ D$ function is referentially transparent, extracting the value from the returned future is not–.

6.3.2.2 Executors are stateless

As mentioned in Section 3.2, in an executor tree architecture, executors are abstractions of *mutable* execution contexts –i.e., each executor holds a varying *state*–. In an executor tree, executor state visibilities are propagated *bottom-top*, so that schedulers in higher-levels in the hierarchy

perform scheduling decisions based on the actual executor contention underneath –e.g., processor occupations, length of work queues storing pending tasks, data distribution, etc.–. However, in the functional model presented, executors are stateless: the only effect of an $ExecuteAsync$ call is to return the *futurized* result.

In order to account for that, a kind of state representation for executors should be defined, in a way in which executor states are only writable by scheduling functions applied to them. Additionally, the states of the lower-level executors would be propagated bottom-top so that they can be accessed by upper executors above.

This problem is however a particular case of the general limitations exposed in Section 1.2.1.4, regarding the difficulty to account for a self-referential execution state within a functional program.

6.3.2.3 Scheduling is static

In relation to the previous section and developments in Section 3.2.4, the scheduling functions just presented in this chapter (see Section 6.2.3) are static –i.e., their output is based on information (expanded execution paths) not varying at run-time–. As mentioned, static schedulers are inherently short-sighted and greedy, so they are not expected to be competitive in the kind of scenarios in which an implementation of STEEL-PM is intended to operate. Ideally, all the executor states which might be influenced by the scheduler decision, together with a representation of the work queue pending to be dispatched, should be fed to the scheduler input.

6.3.2.4 Related work

The previous issues could be re-evaluated under the light of several existing approaches in functional programming.

In relation to the mutability of futures, the *continuation* abstraction, based on the CPS pattern [141, 151], could be employed instead of *futures*. In particular, the *continuation* monadic pattern [67] could be used to compose asynchronous computations in a pure way.

The problem of synchronization in a multiple task context has been already treated from a functional approach. For example, in [66] it is exposed how a context in which tasks in a OS kernel running concurrently can be modeled in terms of *Resumption* and *State* monads; the former used to model concurrency [120] and the latter to model read-modify operations on a general state. Additionally, a purely functional implementation of an OS kernel is shown, in which operations like task preemption, process forking, synchronization and message passing are considered.

In [87], it is exposed how data-oriented workflows and a set of dependent tasks related by a DAG can be modeled with a set of functional constructs based on lambda-calculus. Although it does not target any HPC-related application, some of these ideas could be considered to enable fine-grained task synchronization in a purely functional way.

Related to making executors stateful and enhancing scheduling, so that the scheduling is explicitly sensible to *computational effects*, developments that explore incremental [27] and adaptive [3] functional programming patterns could also provide utility.

In this line, also in relation with Section 1.2.1.1, recall that the monad pattern, originally devised as a mathematical concept within Category Theory and then applied to Functional Programming Theory, was successfully accepted as a mechanism to incorporate side-effects and a systematic framework for scalable function composition into functional programming languages. Its category-theoretic dual, the *co*monad [96], is a pattern that when applied to programming language theory, it is able to model *contextual computations* [109]. In this sense, several theoretical works [93, 121, 148, 118], have exposed the appeal of comonadic patterns to program environment- or context-dependent computations from a purely functional approach.

Any HPC-oriented runtime system must work and respond in strongly context-dependent situations, as the underlying hardware system must be constantly monitored in order to mitigate usual

load-balancing, contention and data locality problems. For this reason, any *purely* functional runtime system intended to work as a middleware for HPC applications would presumably need to incorporate comonadic patterns as a core component.

In relation to an hypothetical purely functional representation of STEEL-PM, comonadic patterns could be employed to approach the previously mentioned problems, as they are related with a form of *environment* or *context* –either in the form of task dependencies or in the form of the underlying hardware state and stateful scheduling–. Specifically, asynchronous computations of inter-dependent tasks could be modeled in terms of the *Store* and *Stream* comonads, following the line of [148].

Future eyes
futurize
future rise.

7 Conclusions

User-defined execution relaxations are fundamentally a new approach to HPC application development. The core idea consists of declaring relaxations to let the runtime system access to new regions in the space of execution possibilities. These regions may be worth exploring for two reasons: *(i)* to steer the execution according to some optimization criteria, and *(ii)* to *widen* or to *make more flexible* the execution, so that it can be automatically adapted to other application instances or environments (e.g., different application parameters or parallel computing platforms).

To view the user-defined execution relaxations from a bigger picture, simultaneously encompassing performance, efficiency, programmability and portability considerations, this chapter summarizes the contributions of this thesis, also briefly exposing future extensions of the current STEEL-RT implementation and interface, together with future research paths.

7.1 Contributions

A high-level, general-purpose parallel programming model. The proposed programming model and interface do not require any low-level programming instruction nor the use of any parallel tool for synchronization, yet the API has shown to be general enough to let the user (implicitly) represent generic parallel programs instantiated to different parallel contexts: departing from the task-asynchronous and data-flow execution models, the computation of parallel tasks is derived from a code that looks sequential, yet different levels of parallelism are accessible and automatically unfolded by the runtime. Also, the proposed STSE model, in which the execution entry point is reduced to a *single task*, representing the full application, that is assigned to a *single executor*, representing the complete parallel system, is one of the main contributions of the thesis. Nevertheless, the use of C++- concurrent and parallel programming tools is still available (although in general discouraged, regardless of the scale target platform) when needed by the user.

A development paradigm in which *user-defined execution relaxations* yield *runtime-driven rich execution expansions*. The proposed parallel programming model lets the user to express *execution relaxations* to remove the decision cost from the development stage, and to expose execution opportunities to be automatically exploited at run-time. Results presented in Chapter 5 show how either simple and incremental task featurizations and platform-dependent executor deployment were enough to yield rich executions.

Decoupling between runtime scheduling and application development. The exposed interface showed user-oriented mechanisms to formalize execution complexity which, once declared and compiled, they are translated and interpreted by a runtime system. This *decoupling* between runtime scheduling and application development follows a declarative philosophy, in which the *when*, the *where* and the *how*, regarding task execution, are no longer important during application development. In this model, the user just cares about writing or reusing the fundamental building blocks (or kernels) of the application, and expressing their dependencies. In this model, the execution orchestration is deferred during development as a problem to be solved by the runtime. Although complex, this task orchestration is a highly parametrized and well-defined problem (thus subject to full automatization) where *inputs* (i.e., the useful information needed to take a decision) and *outputs* (i.e., the set of actions to take) are clearly defined for all user-defined relaxations.

Strongly-typed, clear and exception-safe API. The proposed model and runtime system implementation rely on a *strongly-typed*, *hard-to-misuse* and *exception-safe* interface. Also, RAII-based patterns have shown to be of great importance in user code, guaranteeing a deterministic, automatic and safe release of resources without the need of any unpredictable garbage collection-based mechanisms. Also, the proposed STEEL-RT-callable interface encourages the use of free functions within parametrized stateless structures, although the user is still free to have its own state.

A feature-scalable and portable framework. The principles of parallel programming are the same whether the target platform is a supercomputer or a hybrid CPU-GPU system: same problems (although in different scales) in terms of memory management, load balancing, runtime overhead and resource starvation arise in both scenarios. Portability demands that the program should look nearly the same regardless of the target system. What needs to be different is the actual execution that is performed *under the hood* by the runtime system. The proposed hierarchical *executor-allocator* architecture and the API deeply based on generic programming clearly follow this philosophy: the entry point is always a single executor regardless of the platform, which can be automatically identified at install-time. Also, it was shown how heterogeneous processors and deep-memory hierarchies could be seamlessly exploited from the STSE model.

7.1.1 *Execution relaxations*-leveraged programmability

Figure 7.1 illustrates the *traditional* development process of HPC applications. Three loops can be distinguished, for which knowledge of concurrency and parallel programming knowledge are mandatory. The first *correctness loop* abstracts the development process seeking program correctness, which, once attained, is typically followed by a sort of performance evaluation which should highlight the main bottlenecks that harm the application performance. The correctness stage is much harder when dealing with concurrency and parallelism, due to their inherent difficulties (e.g., non-deterministic race conditions, deadlocks, or livelocks).

Second *coarse optimization* and third *fine optimization* loops require direct intervention from the user side by means of *profiling*, and code *optimization* actions, which ultimately lead to a high-performing program execution.

Coarse optimization loop commonly involves major changes that greatly impact performance and may involve ad-hoc testing (e.g., software refactors, changes in data-layouts and changes in parallel synchronization patterns). Fine optimization loop may refer to minor changes in code not requiring testing (e.g., platform-dependent parameter tuning, data granularity fixing, thread parallelism tuning, or addition of compiler-oriented hints for performance).

In general, optimization loops reflect incremental optimizations tuned for specific application and system-dependent parameters and characteristics. They rely on some form of feedback (symbolized by the *Performance report* node), in the form of performance metrics or execution traces,

that need to be properly interpreted by the application developer to identify sources of performance drops. This analysis may lead the developer to the identification of performance problems requiring major changes.

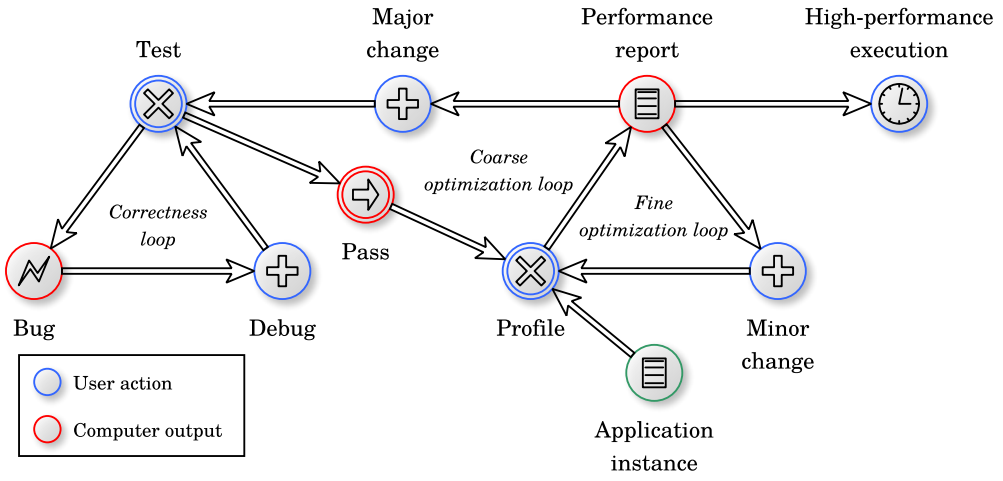


Figure 7.1: Diagram of traditional development process. Execution optimization is achieved by means of incremental application- and system-dependent optimizations. Correctness loop may involve debugging concurrency-related and non-deterministic errors.

Analogously, Figure 7.2 illustrates the development process proposed by the *execution relaxations* paradigm. Contrary to traditional development, the only user actions present in the optimization process are the user-defined execution relaxations (represented by the *Relax execution* node). Based on some knowledge of the general properties of the application and the architectural characteristics of the platform, the user would decide to incrementally relax the execution accordingly.

In practice, the *featurization loop* would not need to be traversed many times, as in practice there are not many opportunities for *relaxed execution* (e.g., concerning data / thread granularity, task reimplementation, heterogeneous task dispatching, or relaxed task precision). It is indeed restricted by the properties of the application and the architectural features of the parallel platform. In this featurization loop it could also be considered *coarse* and *fine* featurizations, but they are not illustrated for the sake of clarity. A coarse featurization could refer to adding entire new features which might require testing (e.g., adding a new architecture-specific kernel reimplementations for an abstract task), while a fine featurization could involve adding few parameters to incrementally *widen* or *relax* an already defined feature.

The process of relaxing the execution could be viewed as adding tunable knobs that represent how much actions, runtime decisions and in essence *responsibility* is delegated from the user to the runtime. Minimal feature relaxation would yield very constrained executions close to what other runtimes can do. Adding relaxations incrementally is done in response of needs of exploration in the state space of possible execution possibilities. Also, features are composable and in general orthogonal by design –i.e., making the runtime sensible to other features on top of others is easily doable under the STEEL-API–, and it is encouraged to expand and enrich the execution possibilities to which the runtime has access.

In addition, an application may be featurized by a set of architecture-dependent features that may or may not be instantiated for a specific target platform. With this regard, the same code reflecting all featurizations for an application will be instantiated differently according to the executor tree that has been deployed during compilation, also based on the architectural features on which STEEL-RT was installed.

An user-defined relaxation could also be motivated from observations regarding a bad convergence during the *automatic optimization loop*, which is composed entirely by an automatizable sequence (*Expanded execution* $\longrightarrow$ *Internal profiling* $\longrightarrow$ *Convergence analysis*). In this sequence, an internal profiling could be generated based on some user-defined optimization objectives and some metrics of execution quality which could be automatically extracted from application monitoring (e.g., time-to-solution, energy efficiency, average parallelism, peak power). Similarly, optimization convergence is also easily detectable in an automatic way.

One important feature of runtime expanded executions is that they are non-deterministic. In particular, repeatedly performing expanded executions could be thought as a kind of *execution path sampling*: in each repetition the runtime can explore more execution possibilities, so that over time the system gains knowledge regarding the paths that maximize the optimization requirements defined by the user. In general, a greater number of relaxations would increase the search space to be explored during the optimization loop, which could lead to longer automatic optimization times. Since state spaces and action spaces are totally parametrized by the application and the underlying platform, the only problem left concerns the design of *generalized scheduling agents*, working in a cooperative and hierarchical way, able to dynamically feed from these state spaces to yield generalized scheduling actions.

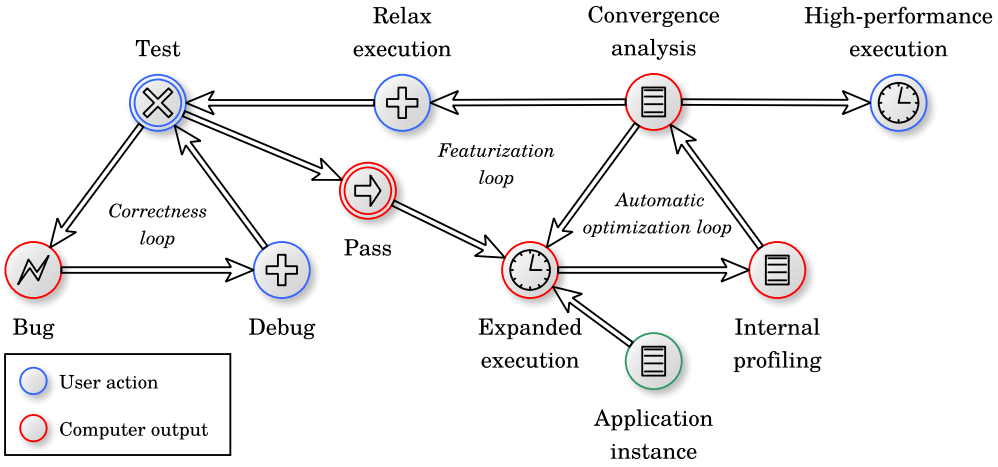


Figure 7.2: Diagram of STEEL-PM-based development process. Execution optimization is achieved by means of incremental featurizations and automatic profiling and execution path exploration. Correctness loop is also simplified because the STEEL-PM interface favors a sequential style and encourages pure functions.

7.1.2 Performance portability

To enhance portability of an application for a set of different systems, the user may declare wider / more relaxed execution opportunities to let the runtime explore a different set of relaxed scheduling configurations. If the executor tree is automatically deployed, it will reflect an architecture dependent on both the user-defined application relaxations and the parallel architecture characteristics. Hence, at compilation, simpler / complex executor trees will be delivered on simple / complex parallel platforms.

As mentioned, the user *augments* the available parallel platforms targetable for a given application by providing new ways to map the computational kernels of the application to those targets (e.g., by architecture-tuned kernel implementations): the more featurized a task into application kernels, the more portable it will be. In the same line, the more relaxed an application is for a given platform, more number of execution paths will be accessible to the runtime.

With regard to previous development diagrams, it could be observed that the optimization loop of Figure 7.1 achieves optimization by sacrificing portability, as those optimizations are normally application and platform dependent (see Figure 7.3). Contrary, the featurization loop of Figure 7.2 offers benefits in terms of both portability and programmability, delegating the optimization problem of efficient execution orchestration to an automatic system (i.e., the runtime) (see Figure 7.4). This second paradigm may be impractical if the complexity is not sufficient, as applications with simple parallel patterns running on not too complex parallel platforms may be easily optimizable by experienced application developers. On the contrary, this approach may start to be beneficial in parallel scenarios in which the execution orchestration complexity goes beyond the capacity and available time of experts.

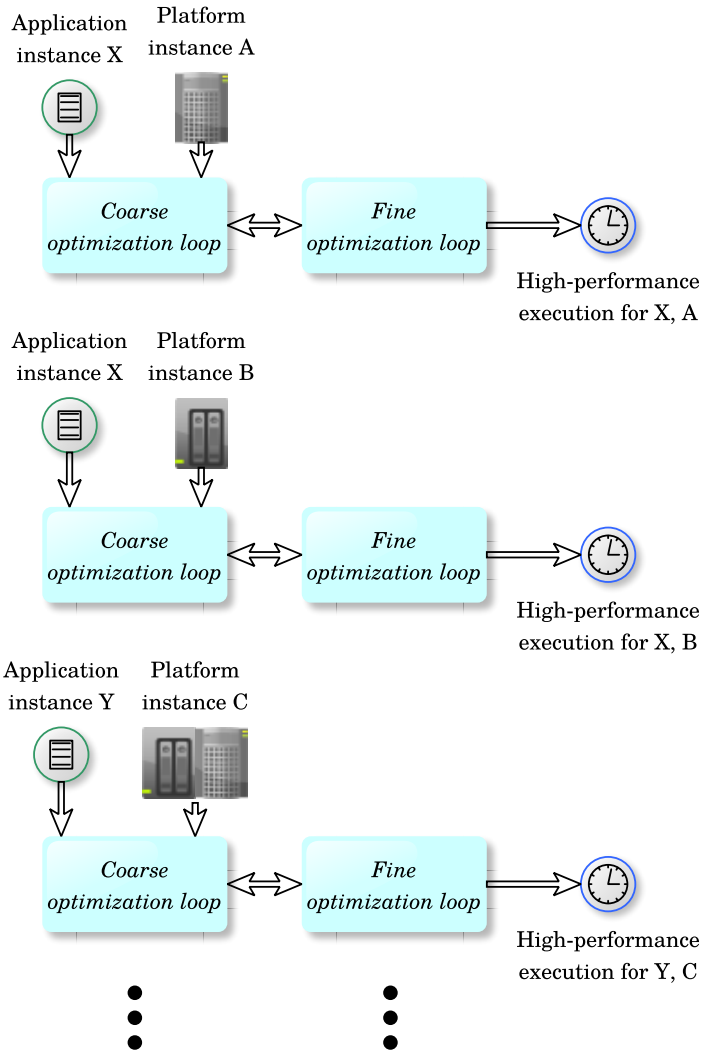


Figure 7.3: Diagram exposing ad-hoc performance tuning for each application instance / platform instance.

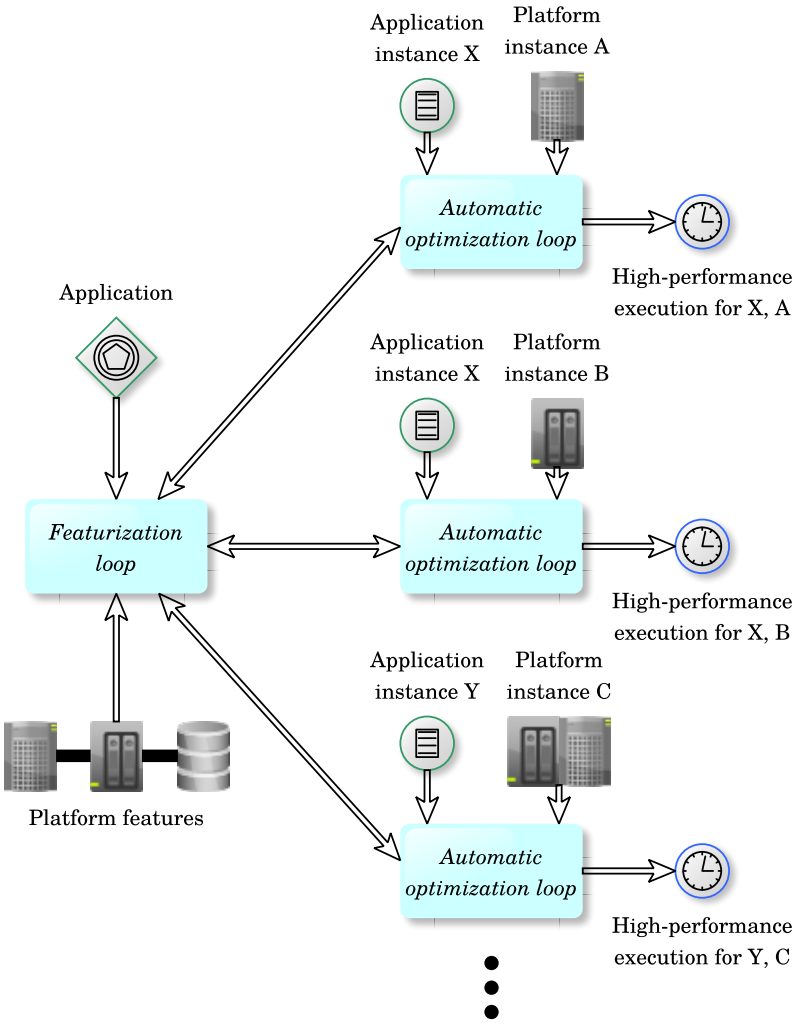


Figure 7.4: Diagram exposing STEEL-PM tuning based on general application / platform featurizations and automatic performance tuning.

7.1.3 Publications

The following papers have been published in relation to this thesis.

- Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. HeSP: A Simulation Framework for Solving the Task Scheduling-Partitioning Problem on Heterogeneous Architectures. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, pages 183–195, Cham, 2016. Springer International Publishing
- Antón Rey, Francisco Igual, Manuel Prieto Matias, and Jan Prins. Performance and Scalability Study of FMM Kernels on Novel Multi- and Many-core Architectures. *Procedia Computer Science*, 108:2313–2317, 12 2017
- Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. Variable intra-task threading for power-constrained performance and energy optimization in DAG scheduling. *The Journal of Supercomputing*, 75(3):1717–1731, Mar 2019
- Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. STEEL-RT: combining single task–single executor model and expanded scheduling to ease heterogeneity exploitation. *The Journal of Supercomputing*, Aug 2019

7.2 Future work

Several features will be included in future STEEL-RT implementations, oriented toward *(i)* enhanced scheduling, *(ii)* extended portability to other compute backends, *(iii)* resiliency, *(iv)* improved maintainability / usability and *(v)* benchmarking.

7.2.1 Stateful, generalized and multi-objective expanded execution

Current STEEL model and runtime implementation will be extended for *multi-objective*, *model-free*, and *stateful generalized scheduling* functionalities, aiming at out-of-the-box yet state-of-the-art performance from a purely STSE approach, limiting the user intervention exclusively to *execution relaxations*. Note that current STEEL-RT generalized schedulers are already fully parametrized, meaning that their *state spaces* –i.e., input– and *action spaces* –i.e., output– are already fully defined at compile-time based on the executor characteristics (and its place in the hierarchy) and the user-defined relaxations, respectively. What remains to be done is to connect these *inputs* and *outputs* by means of automatically *trainable agents* from previous execution experiences, which would ideally cooperate and converge to better solutions as they acquire more knowledge. Regarding Chapter 2, training agents through modeling and simulation will likely acquire great importance to decrease the cost and / or increase the speed of the training process (eventually helping to reduce the convergence time of the automatic optimization loop of Figure 7.2).

7.2.2 Distributed layer and additional communication backends

Current STEEL runtime implements a data coherency layer able to automatically manage data across three kinds of memory spaces: *non-volatile* or *disk*-like memory spaces, *multicore* RAM –also NUMA-aware– and *device* RAM tied to CUDA-capable GPU devices. With regard to intra-node communication, STEEL runtime will be extended to handle OpenCL-capable devices. Concerning distributed systems, an additional inter-node communication layer will be incorporated following patterns based on task asynchrony, one-sided communications and PGAS models. Additional research will be required to assess the scalability of the STSE model and the potential of *distributed* yet *expanded* executions.

7.2.3 Support for resiliency

Regarding execution resiliency, a kind of *task-checking* could be viewed as a possible user-defined execution relaxation, which could provide runtime support for resiliency under the STSE model. This could be beneficial in situations in which a set of interdependent tasks are run on a set of interconnected processors that might fail due to *soft* errors. The *open question* that could be addressed under the *relaxed execution* paradigm could be *"How often do we have to verify the result of a partial computation in a system with non-zero failure probability?"*.

If the verification is made occasionally and / or it is rather lightweight (e.g., the correctness check of a computation is not too expensive and / or it is performed only once for every million tasks) the runtime would incur in just a little *checking* overhead and low memory footprint to store the checked execution states. If soft errors are rare, this situation would be acceptable. However, in the case of too frequent failures, the runtime would need to discard a lot of invalid results too frequently before recovering the latest (and probably old) valid execution state.

On the contrary, if the runtime performs a frequent and / or costly checking (e.g., the result of *every* task is checked and this verification is relatively expensive), the runtime would incur into a significant overhead and a lot of memory footprint, which could cause an unacceptable performance degradation if errors are rare. However, in the case of failure, the amount of *corrupted* and useless results that would demand a re-computation would be small, which could be desirable in situations in which errors are common.

Thus, in the context of an *user-defined relaxed checking* and *runtime-expanded task verification*, given *(i)* a soft error failure probability for all kernel-processor pairs (possibly estimated or learned from previous experience), *(ii)* a set of functions known as *user-defined task checkers* for each *verifiable* task kernel (which would be basic predicates that tell if the result of a task execution is valid or not), and *(iii) user-defined-runtime-callable* functions to store the result of a validated computation, the *expanded execution runtime schedulers* would decide whether to check a task (hence incur into an additional cost due to *checking* and *saving* the result) or not (hence take the risk of letting future tasks to operate on corrupted results). This automatic runtime decision could be based on some prior user-defined objective (e.g., *time-to-solution*, *energy efficiency* maximization, or some criteria based on *execution robustness*-vs-*performance* tradeoff) and prior experience gathered by the runtime from similar scenarios.

7.2.4 ISO C++20 improvements

New ISO C++ language standard incorporates a set of features of great interest for future STEEL-RT improvements. First, C++20 will enable *Coroutines*, which provide support for implementing stackless resumable functions. They will affect both the runtime and the API parts of current STEEL-PM implementation, as they will serve as the main mechanism for task generation and for modeling lazy and stream-like computations. Secondly, C++20 *Modules* will be incorporated to improve compilation times and to simplify the current build architecture, which mainly relies on expensive header file inclusions for template instantiations. Finally, C++ *Concepts* (already used partially in the interface) will be extensively used to enhance runtime robustness and to simplify to greater extent the usage of STEEL-API.

7.2.5 Raising abstractions

Current STEEL-RT implementation exposes a purely C++ frontend interface, and its header-only characteristic comes with the price of a quite sophisticated compilation process that could obscure and harm the initial quest for high-level abstractions. Despite the evolution of C++ into a language focused on *high-level* yet *zero-runtime-cost* abstractions, it is commonly classified as a *system-software* rather than an *application-level* language. For this reason, regardless the interfacing improvements just exposed in Section 7.2.4, the API proposed in this thesis will be further *lifted* toward higher levels of abstractions, so that the future frontend-API layer of STEEL-RT would fully hide the current compilation and build mechanisms, and the C++-like syntax of kernel definitions, task featurizations, or executor deployment. This frontend will also provide additional safety and error-checking layers and clearer report of compilation errors. Specifically, these additional layers could be implemented in a high-level and interpreted language such as Python; and education- and research-oriented tools such as JupyterLab [82] could be employed to popularize the use of the current framework, for example by introducing the concept of execution relaxations and their effects in an interactive way.

7.2.6 Benchmarking

A set of benchmarks will be implemented in order to assess how STEEL-RT responds to realistic workloads. In particular, *highly-coupled* / *HPC-characteristic* workloads will be considered for study in favor of less challenging and easy parallelizable *loosely-coupled* / *data-centric* workloads. Despite it was already demonstrated how rich executions are achievable (actually *expanded*) by means of user-defined execution relaxations, it remains to be proven whether it is possible to achieve high-performance in these realistic applications using a programming style purely based on a STSE-frontend and the declarative patterns presented in this thesis. In addition, the *practicality* must as well be assessed, i.e.: to what extent the general *execution efficiency* is achieved and how much effort –in terms of programming and automatic training– will be required for that.

Benchmark suites like NAS [14] and more modern Rodinia [33] or PARSEC [21] have been used to measure parallel performance for well-known computational kernels and applications, some of which will be considered to evaluate STEEL-RT performance. The goal of these suites is to provide a wide range of realistic workloads requiring different parallel execution and communication patterns. Ultimately, benchmark suites motivate parallel computing research regarding both hardware architectures and parallel programming techniques.

In future benchmarks targeting STEEL-PM, the *problem domain* will be broadly classified according to two dimensions: *execution predictability* and *arithmetic intensity*. With regard to the first dimension, applications will be distinguished according to how *(i) deterministic / data-independent* or *(ii) non-deterministic / data-dependent* are their algorithms. Secondly, applications will be classified according to their *arithmetic intensity*, which refers to the relation between the number of processing operations and data movement operations.

Those applications requiring low arithmetic intensity will face more challenges regarding parallel scalability, as the saturation of the computing units will be harder to achieve due to communication bottlenecks. Applications exhibiting an *algorithmic freedom* (see Section 1.3.3.5), for which a set of implementations with different arithmetic intensity demands may be employed, will be benchmarked to study the effect of the proposed *algorithmic reimplementation* runtime decisions presented in Section 3.1.3.2. Also, algorithms with data-dependent parallel patterns (in which tasks are generated depending on previous results) will be more challenging in general, and mechanisms exposed in Section 4.2.2.6 for *data-feedback-driven execution* will be employed to assist the runtime.

Numerical linear algebra applications (e.g., *dense / sparse* and *direct / iterative* solvers) will provide an excellent test-bed to explore these two dimensions, together with mixed-precision considerations. Also, several applications from computational physics / engineering / vision and biology will be implemented to explore the effect of *particle-*, *grid-*, *stencil-* and *wave-front*-based algorithms, jointly considered with stream-like, adaptive and hierarchical patterns from a STSE standpoint.

Appendices

A

Prerequisites of C++

This appendix exposes a minimal subset of C++ on which most of the STEEL-API exposed in Section 4.2 is based.

A.1 Interfaces

C++ namespaces provide named-encapsulations for types and functions. In particular, the C++ standard library exposes a set of types and functions within the namespace `std`, and the entire STEEL-API and STEEL-RT implementation is encapsulated in `steel` namespace.

Structures (`struct`) and classes (`class`) are also designed to provide encapsulation, not only to types and functions, but also serving as building blocks for Object-Oriented programming. Functions defined within the scope of a `class` or `struct` are called *member functions*, and a `static` qualifier in the function signature indicates that the function is not tied to any class object, neither tied to any non-static data member of the class. Structures and classes in which all functions are `static`-qualified, can be thought of as interfaces. In particular, if its functions do not have side effects –i.e. they are *free* or *pure* functions–, the interface is considered *pure*. As it will be exposed, the STEEL-API relies mainly on these kind of classes.

Structures and classes working as interfaces offer an additional benefit with respect to namespaces, in the sense that they can be parametrized with types and compile-time values via the `template` qualifier.

The parameters (types or values) exposed after the `template` express a set of instantiation possibilities for the structure or class definition that comes after it. A template class that is not instantiated will not generate any run-time code by the compiler: only template instantiations for specific types and / or values will generate actual run-time code by the compiler.

As STEEL-RT is deeply rooted on functional and generic patterns, STEEL-API requires the user to provide *template* and (optionally but recommended) *pure* interfaces. For this reason, classes and structures definitions presented in the STEEL-API within the user namespace `app` are encouraged to be stateless by design –i.e. they only expose static functions with no side effect, or at least with *not-observable* side effect–.

By *not-observable side effect* it is meant that if there is any side effect, it cannot be accessible by any means within the user code (see Section 1.2.1.1). For example, the user might provide a pure function in which a call to an external third-party runtime library is done. As this internal library is expected to carry its own internal state, this is hidden and not-observable from the user side, hence the user function is *de facto* pure. As mentioned in Section 4.1.2, regardless of its in-

Listing A.1: Basic example `constexpr` and interface constrained parametrization.

```
1  enum class my_enum : int {foo=13, bar=21}; /* Enumeration definition */
2  /* Compile-time predicate restricting template parameters. */
3  template <my_enum EnumValue, typename T>
4  constexpr bool my_predicate =
5      EnumValue == my_enum::bar && std::is_floating_point_v<T>;
6
7  /* Interface parametrization constrained by my_predicate. */
8  template <my_enum EnumValue, typename T>
9  requires my_predicate<EnumValue, T>
10 class interface {
11   static int rand() { return /* Random integer */; }
12 public:
13   static int get_rt(int a) { return  a * rand(); }
14   static constexpr int get_ct(int a) { return a / (int)EnumValue; }
15 };
16
17 void main(void) {
18   /* User sets an integer in command line. */
19   int userInteger; std::cin >> userInteger;
20
21   /* Forbidden call (compilation error is returned). */
22   int a = interface<my_enum::foo, char>::get_rt(userInteger);
23
24   /* Run-time call will successfully compile. */
25   int b = interface<my_enum::bar, double>::get_rt(userInteger);
26
27   /* Valid call valuable at compile-time. */
28   constexpr int b = interface<my_enum::bar, float>::get_ct(42);
29   static_assert(b == 2); /* Assertion available at compile time. */
30 }
```

ternal management of state and resources, the use of third-party libraries for computational kernels is encouraged in STEEL model.

Also, the user might require a call to request a managed memory allocation to STEEL-RT. The fact that STEEL-RT might recycle a pointer or request an actual memory allocation to the OS, is also considered invisible from the user side. Contrary, if the user modifies global variables defined outside the scope of a user-defined kernel, this is discouraged –though allowed, as it is in C++ language– as it breaks the functional-based guidelines of STEEL-PM development.

A.2 Template specializations and constraints

Templates are the main mechanism in which powerful generic programming patterns can be expressed in C++. Apart from structures and classes, types, values and functions can be parametrized by templates. A template instantiation is done during compilation following certain strongly typed and functional rules. Moreover, template definitions can be further constrained by the user by means of C++ Concepts idiom [61], which has been recently approved for standardization in upcoming ISO C++20 language. With this regard, a `requires` preceding a compile-time predicate can be written after a template parametrization, as exposed in line 9 of Listing A.1. This compile-time predicate can be based on any predefined *constant expression* `constexpr` or any compile-time predicate depending on the internal properties or a type (line 4). In this case, a `constexpr` qualifier is used in lines 4, 14 and 28 for compile-time evaluation.

Regarding Section 4.1.1, in lines 9 and 29 it is exposed two important idiomatic features – `requires` and `static_assert`– that can be employed to ensure correct use of an interface at compile-time. In particular, in Section 4.2 it is shown that `requires` qualifier is used to constraint the execution of a task depending on compile-time information exposed by an executor.

A.3 Frequently used types

The STEEL-PM application developer will be asked to expose type definitions to be visible to STEEL-RT. The `using` qualifier is used to define a type or a template-type, which could be defined from an external template-type.

Also, strongly-typed enumerations, defined by `enum class`, are extensively used within STEEL-RT and STEEL-API for template value specializations.

Specializations of standard template-types such as `std::tuple`–to represent a set of heterogeneous types–, `std::function`–to represent a callable object–, and `std::integer_sequence`–to represent a compile-time sequence of integral values– are also pervasive in STEEL-RT and STEEL-API. In particular, specializations of *integer sequences* acquire great importance to define compile-time sets of enumeration values.

A.4 *Resource Acquisition Is Initialization*

Another feature of great importance in STEEL-API is the use of the *Resource Acquisition Is Initialization (RAII)* idiom [138, 81]. RAII patterns endow variables with ownership semantics that allow to deterministically and automatically control their lifetime in a safe way. One of the areas in which RAII exhibits great importance is in the context of safe, efficient and automatic management of memory resources. In particular, standard *smart pointers* `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr` are particular instances in which RAII permits to deterministically handle allocated memory without the need of any garbage collection mechanism. With this regard, RAII is a response to two of the mentioned guidelines: (1) *"Don't pay for what you don't use"* –e.g., there is no need for garbage collecting if memory can be deterministically released–, and (2) *"Minimize the possibility of resource leaks via ownership semantics"*.

Other benefits of RAII are related to safety in concurrent programming. Thanks to RAII patterns, STEEL-API provides a safe and clear way for the user to protect arbitrary data objects for concurrent access, while also making STEEL-RT to properly detect data dependencies between tasks.

B Platforms

This appendix summarizes the main characteristics of the two machines used through the document: MACHINE1 and MACHINE2.

B.1 MACHINE1

MACHINE1 is a heterogeneous platform equipped with 2× 14 core Intel Haswell CPUs, 2× NVIDIA Geforce GTX980 GPUs and 1× NVIDIA Geforce GTX950 GPU. Tables B.1 and B.2 report the hardware and software features of the platform, respectively.

	Intel Xeon E5-2695 v3	NVIDIA Geforce GTX980	NVIDIA Geforce GTX950
Number of devices	2	2	1
Arch	Haswell	Maxwell	Maxwell
Frequency	2,300	1,126	1,024
Local RAM GB	2 × 32 DDR4	4 GB GDDR5	2 GB GDDR5
Peak GB/sec	119.2	224	106
Core count	14 (per socket)	2048 CUDA cores	768 CUDA cores
System bus		QPI	
System bus BW GB/sec		9.6	
PCIe		PCIe Gen3 x16	
PCIe bandwidth GB/sec		16	

Table B.1: Characteristics of the processing devices in MACHINE1.

Dependency	*Version*
OS distribution	Debian 4.9
Linux kernel	4.9.0
Build system	CMake 3.13
Compiler	GNU GCC 8.2
Supported compute backends	OpenMP 4.5, CUDA 10.0
Execution tracing	Extrae 3.7

Table B.2: Software used in MACHINE2.

B.2 MACHINE2

MACHINE2 is a heterogeneous platform equipped with 2× 20 core Intel Skylake CPUs and 2× NVIDIA Volta GPU (experimental results in Section 5.4.2 replaced one of the Volta GPUs by a GeForce 1080). Tables B.3 and B.4 report the hardware and software features of the platform, respectively.

	Intel Xeon Gold 6138	NVIDIA V100
Number of devices	2	2
Arch	Skylake	Volta
Frequency	2,000	1,455
Local RAM GB	2×48 DDR4	32 HBM
Peak GB/sec	119.2	900
Core count	20 (per socket)	5120 CUDA cores
System bus	QPI	
System bus BW GB/sec	19.2	
PCIe	PCIe Gen3 x16	
PCIe bandwidth GB/sec	16	
Non-volatile storage (Fast)	Samsung 970 EVO NVMe	
Capacity	900 GB	
Non-volatile storage (Slow)	INTEL SSDSC2KB240G7	
Capacity	240 GB	

Table B.3: Characteristics of the processing devices in MACHINE2.

Dependency	*Version*
OS distribution	Debian 4.9
Linux kernel	4.9.0
Build system	CMake 3.13
Compiler	GNU GCC 8.2
Supported compute backends	OpenMP 4.5, CUDA 10.0
Execution tracing	Extrae 3.7

Table B.4: Software used in MACHINE2.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 247–259, New York, NY, USA, 2002. ACM.

[4] W.B. Ackerman. Data flow languages. *Computer; (United States)*.

[5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, jul 2009.

[6] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. Mallba: A library of skeletons for combinatorial optimisation. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 927–932, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[7] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with fastflow. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 170–181, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Pedro Alonso, Manuel F. Dolz, Rafael Mayo, and Enrique S. Quintana-Ortí. Modeling power and energy of the task-parallel cholesky factorization on multicore processors. *Computer Science - Research and Development*, 29(2):105–112, May 2014.

[9] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[10] José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. Locality-aware automatic parallelization for gpgpu with openhmpp directives. *International Journal of Parallel Programming*, 44(3):620–643, Jun 2016.

[11] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

[13] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526 – 2533, 2009. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, Nov 1991.

[15] Pavan Balaji. *Programming Models for Parallel Computing*. The MIT Press, 2015.

[16] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In E. Luque, T. Margalef, and D. Benítez, editors, *Proceedings of the 14th International Euro-Par Conference*, Lecture Notes in Computer Science, 5168, pages 739–748. Springer, 2008.

[17] María Barreda, Sergio Barrachina Mir, Sandra Catalán, Manuel F. Dolz, G Fabregat, Rafael Mayo, and E S. Quintana. A framework for power-performance analysis of parallel scientific applications. In *3rd Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119, 01 2013.

[18] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: A c++ template library for high performance stream parallel processing. *International Journal of High Performance Computing Applications*, 2016.

[19] Nathan Bell and Jared Hoberock. Chapter 26 - thrust: A productivity-oriented library for cuda. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359 – 371. Morgan Kaufmann, Boston, 2012.

[20] S. Benkner, S. Pllana, J. Larsson Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, Sep. 2011.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[22] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. *Generic Parallel Programming Using C++ Templates and Skeletons*, pages 107–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[23] Jacek Błażewicz, Maciej Machowiak, Jan Wkeglarz, Mikhail Y. Kovalyov, and Denis Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, 129(1):65–80, Jul 2004.

[24] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[25] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[26] W. CARLSON. Introduction to upc and language specification. *CCS-TR-99-157*, 1999.

[27] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 26–35, New York, NY, USA, 2002. ACM.

[28] F. Cesarini and S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, 2009.

[29] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[30] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[31] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[32] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[34] Quan Chen and Minyi Guo. Task scheduling for multi-core and parallel architectures. In *Springer Singapore*, 2017.

[35] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[36] Alonzo Church. *The Calculi of Lambda-Conversion*. Oxford University Press, London, UK, 1941.

[37] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Münster Skeleton Library Muesli: A comprehensive overview. Technical report, 2009.

[38] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. Solving lattice qcd systems of equations using mixed precision solvers on gpus. *Computer Physics Communications*, 181(9):1517 – 1528, 2010.

[39] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack cap: Adaptive dvfs and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, Dec 2011.

[40] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P. Wacrenier. Resource aggregation in task-based applications over accelerator-based multicore machines. Technical report, INRIA, October 2015.

[41] NVIDIA Corporation. Nvidia brings cuda to arm, enabling new path to exascale supercomputing, 2019. [Online; accessed June-2019].

[42] Guy Cousineau and Michael Mauny. *The Functional Approach to Programming*. Cambridge University Press, New York, NY, USA, 1998.

[43] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online strategies for high-performance power-aware thread execution on emerging multiprocessors. In *Proceedings 20th IPDPS*, page 8, April 2006.

[44] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing parallel patterns out of the corner: The p3 arsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 14(4):33:1–33:26, Oct. 2017.

[45] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175. e4175 cpe.4175.

[46] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[47] Vassilios V. Dimakopoulos and Panagiotis E. Hadjidoukas. Hompi: A hybrid programming framework for expressing and deploying task-based parallelism. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 14–26, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[48] Jack J. Dongarra, Jeffrey Hittinger, John. Bell, Lorenzo Chacón, Rachel Falgout, Marcel Heroux, Paul D. Hovland, Esmond G. Ng, Chris Webster, and Stefan M. Wild. Applied mathematics research for exascale computing. 2014.

[49] A.A.A. Donovan and B.W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education, 2015.

[50] Anastasios Doumoulakis, Ronan Keryell, and Kenneth O'Brien. SYCL C++ and OpenCL interoperability experimentation with triSYCL. In *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, pages 31:1–31:8, New York, NY, USA, 2017. ACM.

[51] Alejandro Durán, Eduard Ayguadé, Rosa M. Badía, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[52] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[53] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[54] Ben-Itzhak et al. Performance and power aware cmp thread allocation modeling. In Yale N. et al. Patt, editor, *HIPEAC*, pages 232–246. Springer Berlin Heidelberg, 2010.

[55] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972.

[56] FSF. *GNU Using the Compiler Collection (GCC)*, 2018. https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc.

[57] Jose Daniel Garcia. C++ language support for contract programming. 2014.

[58] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308, May 2013.

[59] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[60] Scott Le Grand, Andreas W. Götz, and Ross C. Walker. Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374 – 380, 2013.

[61] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[62] W.D. Gropp, W. Gropp, A.D.F.E.E. Lusk, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Number v. 1 in Scientific and engineering computation. MIT Press, 1999.

[63] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[64] A. Haidar, A. YarKhan, C. Chongxiao, P. Luszczek, S. Tomov, and J. Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *HPCC, 2015*, pages 861–864, Aug 2015.

[65] Michael Haidl and Sergei Gorlatch. Pacxx: Towards a unified programming model for programming accelerators using c++14. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.

[66] William L. Harrison. The essence of multitasking. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology*, pages 158–172, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[67] HaskellWiki. All about monads — haskellwiki, 2019. [Online; accessed April-2019].

[68] Thomas Heller, Bryce Adelstein Lelbach, Kevin A Huck, John Biddiscombe, Patricia Grubel, Alice E Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, Juhan Frank, Geoffrey C Clayton, Dirk Pflüger, David Eder, and Hartmut Kaiser. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger

of two stars. *The International Journal of High Performance Computing Applications*, 0(0):1094342018819744, 0.

[69] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[70] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[71] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[72] Jared Hoberock, Michael Garland, Olivier Giroux, and Hartmut Kaiser. An interface for abstracting execution. Technical Report P0058R1, February 2016.

[73] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, and Michael Wong. Executors design document. Technical Report P0761R2, February 2018.

[74] Richard D. Hornung and Jeffrey A. Keasler. The raja portability layer: Overview and status.

[75] Intel. *Intel® Xeon® Processor Scalable Family. Specification Update*, February 2018.

[76] Intel Corp. Intel math kernel library (MKL) 11.0. http://software.intel.com/en-us/intel-mkl.

[77] ISO. Programming languages – technical specification for c++ extensions for parallelism. Technical Report N4507, ISO/IEC DTS 19570, International Organization for Standarization, May 2015.

[78] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth edition, December 2017.

[79] Ralph E. Johnson and Brian Foote. Designing reusable classes. 2001.

[80] S.P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming. Cambridge University Press, 2003.

[81] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

[82] Project Jupyter. JupyterLab: Building Blocks for Interactive Computing. https://jupyter.org/, 2019. [Online; accessed 19-April-2019].

[83] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[84] Hartmut Kaiser, Bryce Adelstein Lelbach aka wash, Thomas Heller, Agustín Bergé, Mikael Simberg, John Biddiscombe, Anton Bikineev, Grant Mercer, Andreas Schäfer, Adrian Serio, Taeguk Kwon, Kevin Huck, Jeroen Habraken, Matthew Anderson, Marcin Copik, Steven R. Brandt, Martin Stumpf, Daniel Bourgeois, Denis Blank, Shoshana Jakobovits, Vinay Amatya, Lars Viklund, Zahra Khatami, Devang Bacharwar, Shuangyang Yang, Erik Schnetter, Patrick Diehl, Nikunj Gupta, Bibek Wagle, and Christopher. STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency, February 2019.

[85] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[86] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Champaign, IL, USA, 1993.

[87] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Lambda calculus as a workflow model. *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, pages 15–22, 2008.

[88] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos SYCL for OpenCL: A tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, pages 24:1–24:1, New York, NY, USA, 2015. ACM.

[89] Khronos®Group. *The Industry Open Standard Intermediate Language for Parallel Compute and Graphics*, 1.3 edition, March 2018.

[90] Khronos®Group. SYCL™specification. SYCL™integrates OpenCL™devices with modern C++. version 1.2.1, document revision 5, Apr 2019.

[91] William Killian, Tom Scogland, Adam Kunen, and John Cavazos. The design and implementation of openmp 4.5 and openacc backends for the raja c++ performance portability layer. In Sunita Chandrasekaran and Guido Juckeland, editors, *Accelerator Programming Using Directives*, pages 63–82, Cham, 2018. Springer International Publishing.

[92] D. Kist, B. Pinto, R. Bazo, A. R. D. Bois, and G. G. H. Cavalheiro. Kanga: A skeleton-based generic interface for parallel programming. In *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pages 68–72, Oct 2015.

[93] Edward Kmett. Lens: Lenses, folds, and traversals. https://github.com/ekmett/lens/, 2019.

[94] Chris Kohlhoff. Executors and asynchronous operations, revision 2. Technical Report P0113R0, September 2015.

[95] Chris Kohlhoff. Using customization points to unify executors. Technical Report P0285R0, February 2016.

[96] S.M. Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 2013.

[97] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[98] G. Lawson, M. Sosonkina, and Y. Shen. Energy evaluation for applications with different thread affinities on the intel xeon phi. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pages 54–59, Oct 2014.

[99] J.Y.T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2004.

[100] Jian Li and J. F. Martinez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 124–134, March 2005.

[101] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 54–61, Oct 2013.

[102] Codeplay Software Ltd. Codeplay computecpp, 2019. [Online; accessed April-2019].

[103] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, Dec 2009.

[104] EDWARD A. LUKE and THOMAS GEORGE. Loci: a rule-based framework for parallel multi-disciplinary simulation synthesis. *Journal of Functional Programming*, 15(3):477–502, 2005.

[105] Lukasz Masko, Pierre-Francois Dutot, Gregory Mounie, Denis Trystram, and Marek Tudruj. Scheduling moldable tasks for dynamic smp clusters in soc technology. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 879–887, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[106] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[107] Michael Mattsson. Object-oriented frameworks : A survey of methodological issues. 1996.

[108] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[109] B. Milewski and I. Tabacnik. *Category Theory for Programmers*. Blurb, Incorporated, 2018.

[110] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised*. The MIT Press. MIT Press, 1997.

[111] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[112] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[113] NVIDIA. CUDA basic linear algebra subprograms. https://developer.nvidia.com/cuBLAS, 2019.

[114] NVIDIA. CUSOLVER Library, howpublished = https://developer.nvidia.com/cusolver, 2019.

[115] NVIDIA Corporation. *NVIDIA CUDA Toolkit Documentation*, 10.1.105 edition, March 2019.

[116] NVIDIA Corporation. *NVVM IR Specification*, 1.5 edition, March 2019.

[117] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.

[118] Dominic A. Orchard. Programming contextual computations. 2014.

[119] Matthieu Ospici, Dimitri Komatitsch, Jean-François Méhaut, and Thierry Deutsch. Sgpu 2: a runtime system for using large applications on clusters of hybrid nodes. 01 2011.

[120] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. 2001.

[121] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of International Conference on Functional Programming*, ICFP 2014.

[122] J. Planas, R.M. Badia, E. Ayguade, and J. Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 138–149, May 2013.

[123] Fethi Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. 03 2003.

[124] Krishna K. et al. Rangan. Thread motion: Fine-grained power management for multi-core systems. *SIGARCH Comput. Archit. News*, 37(3):302–313, June 2009.

[125] Gabriel Dos Reis, Microsoft, Jose Daniel Garcia, Manuel Fähndrich, and Shuvendu Lahiri. Simple contracts for c ++. 2015.

[126] Antón Rey, Francisco Igual, Manuel Prieto Matias, and Jan Prins. Performance and Scalability Study of FMM Kernels on Novel Multi- and Many-core Architectures. *Procedia Computer Science*, 108:2313–2317, 12 2017.

[127] Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. HeSP: A Simulation Framework for Solving the Task Scheduling-Partitioning Problem on Heterogeneous Architectures. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, pages 183–195, Cham, 2016. Springer International Publishing.

[128] Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. STEEL-RT: combining single task–single executor model and expanded scheduling to ease heterogeneity exploitation. *The Journal of Supercomputing*, Aug 2019.

[129] Antón Rey, Francisco D. Igual, and Manuel Prieto-Matías. Variable intra-task threading for power-constrained performance and energy optimization in DAG scheduling. *The Journal of Supercomputing*, 75(3):1717–1731, Mar 2019.

[130] Antón Rey, Francisco Igual, Manuel Prieto Matias, and Jan Prins. Performance and scalability study of fmm kernels on novel multi- and many-core architectures. In *Procedia Computer Science*, volume 108, pages 2313–2317, 12 2017.

[131] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15(2):66–71, 87, 2013.

[132] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[133] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[134] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[135] D. Sorin, M. Hill, and D. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool, 2011.

[136] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, Mar 2017.

[137] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[138] B. Stroustrup. *A Tour of C++*. C++ In-Depth Series. Pearson Education, 2018.

[139] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[140] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36(1):277–286, March 2008.

[141] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, Dec 1998.

[142] I. Takouna, W. Dawoud, and C. Meinel. Accurate mutlicore processor power models for power-aware resource management. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 419–426, Dec 2011.

[143] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 666–677, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[144] Simon Thompson. *Type Theory and Functional Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[145] H. Topcuoglu and S. Hariri and. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[146] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

[147] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 323–332, New York, NY, USA, 1992. ACM.

[148] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. pages 135–167, 11 2005.

[149] Bharadwaj Veeravalli, Debasish Ghose, and Thomas Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6:7–17, 01 2003.

[150] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

[151] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[152] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, Dec 2010.

[153] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 368–377, Washington, DC, USA, 2010. IEEE Computer Society.

[154] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[155] T. White. *Hadoop: The Definitive Guide*. Oreilly and Associate Series. O'Reilly, 2012.

[156] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc — first experiences with real-world applications. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 859–870, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[157] Wei Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165, May 2015.

[158] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[159] C. . Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 366–373, June 1988.

[160] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. Upc++: A pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.

Acronyms

API Application Programming Interface. 59, 147, 148

CPS Continuation-Passing Style. 95, 145

DAG Directed Acyclic Graph. xxiii, 13, 15, 20, 21, 23, 24, 26–29, 32, 34, 36, 40, 44, 46, 100, 101, 109, 137, 145

EFT-P Earliest Finish Time Processor. 22, 23

EIT-P Earliest Idle Time Processor. 22

F-P Fastest Processor. 22

FCFS First-come First-served. 22

FMM Fast Multipole Method. 37

FPGA Field-Programmable Gate Array. 43, 62

GPU General-Purpose Graphics Processing Unit. xxiii, 16, 20, 21, 36, 38, 40, 41, 43, 45, 52, 56, 62, 73, 86, 88, 96, 99, 110, 112, 113, 115, 118–121, 130, 148, 163, 164

HeSP Heterogeneous Scheduler-Partitioner. 22–26, 36, 38

HPC High-performance Parallel Computing. 1

IoC Inversion of Control. 93

MIC Many Integrated Core. 43

MILP Mixed-Integer Linear Program. 20, 28, 29, 31, 37, 38

MTME Multiple Task-Multiple Executor. 41–44, 51

MTSE Multiple Task-Single Executor. 41–44, 51

OOCC Out-of-core Computation. xx, 104, 105

PL Priority-List. 22, 23

R-P Random Processor. 22

RAII Resource Acquisition Is Initialization. xxvii, 61, 80, 81, 140, 143, 148, 161

SIMD Single Instruction - Multiple Data. 9, 15, 16, 38, 46

SIMT Single Instruction - Multiple Thread. 9, 15, 16, 38, 46

SMP Symmetric Multiprocessor. 20, 40, 41, 43, 51–53, 56, 73, 86, 105, 109, 111, 112, 118, 120

SPMD Single Program - Multiple Data. 9

STEEL Single Task / Expanded-Execution-Leveraged. xx, xxvii, 59, 63–66, 68, 71, 74, 79, 80, 82, 84, 85, 88, 96, 97, 99, 115, 124, 130, 138, 140, 144, 153, 160

STEEL-API Single Task / Expanded-Execution-Leveraged Interface. 59, 60, 63, 65, 71, 84, 85, 93, 99, 119, 121, 123, 140, 142, 143, 149, 154, 159, 161

STEEL-PM Single Task / Expanded-Execution-Leveraged Programming Model. xx, xxi, xxiii, xxiv, 59–62, 91, 93–97, 99, 123, 136, 140, 142, 144–146, 150, 152, 154, 155, 160, 161

STEEL-RT Single Task / Expanded-Execution-Leveraged Runtime. 59–66, 68, 69, 71, 74, 75, 77, 79–81, 84–86, 88, 91, 93–95, 99, 105, 106, 110–112, 120, 121, 123, 135, 136, 140, 141, 143, 144, 147–149, 153–155, 159–161

STME Single Task-Multiple Executor. 41, 43

STSE Single Task-Single Executor. xxvii, xxviii, 41, 43–45, 50, 57, 59, 64, 82, 83, 95, 109, 113, 114, 121, 123, 130, 147, 148, 153–155

Glossary

application A computational problem that the user needs to resolve. 44

application instance A computational problem together with a set of requirements such as the amount of data to compute, the precision requirements, together with any characteristic regarding the input data. 151

bottom allocator A software abstraction of a physical memory space / storage context (e.g., RAM, disk) able to satisfy memory allocation requests. 47, 62

bottom executor A particular class of executor that abstracts a hardware processing device and eventually perform architecture- or task-related generalized scheduling actions. xxvii, 43, 68–70, 72, 84, 93, 94, 130, 131, 133, 135, 141, 142

callee With respect to executors, the executor that is requested to execute a task. 41

caller With respect to executors, the actor (user or another executor) that demands an executor to execute a task. 41

datum A *unit* / *atom* of data that represents a input dependency and/or the result of a task. 44

execution context An environment in which a workload is processed. It can represent a *physical* hardware or a *logical* / *software-abstracted* set of other execution contexts. 39

execution expansion A runtime behavior that characterizes a generalized scheduler. A particular execution expansion is defined by a set of possible actions that can be taken at run-time. 39, 48–50, 55, 147

execution relaxation A user-defined action consisting of expressing a set of options regarding execution possibilities that can be taken by the runtime system (see *relaxed execution*). xxi, 12–17, 39, 51, 55, 59, 66, 106, 123, 124, 147–149

executor A parametrized and composable software abstraction of an execution context. 39, 40, 67, 93, 123, 124, 130, 138, 144

expanded execution A non-deterministic execution composed by a set of user-defined execution relaxations from which a runtime can choose from. 12, 39, 59, 145, 150, 154

generalized scheduler A software component associated to an executor, designed to perform generalized scheduling actions. 49, 50, 55, 153

generalized scheduling The orchestration at run-time of a program execution by means of actions defined by the user. xx, 49, 51, 55, 150, 153

kernel instantiation The implementation of a task as a particular computational kernel –typically associated to a processor architecture– that can be run in a context of a bottom executor. It is also a particular class of task featurization. 45

mapper executor A particular class of a top executor that encompass more than one executor of any kind to which the incoming tasks are scheduled. 43, 64, 71, 130, 133, 135, 141

relaxed execution A paradigm for application development in which a set of execution possibilities are declared to be delegated to the runtime system. 19, 127, 149, 150, 153

scheduling The assignment of workload to computing resources. It can be regarded as a specific case of *generalized scheduling*. 49

storage context An environment in which data is stored. It can represent a *physical* memory / disk device, or a *logical* / *software-abstracted* set of other storage contexts. 46

task An abstraction of a workload or *unit* / *atom* of computation that processes input data and returns output data. A task can represent a whole application instance or a specific subroutine belonging to the application. 40, 44

task featurization The act of enriching and providing a task with different execution possibilities in terms of possible reimplementations, partitions or kernel instantiations. xx, 44

task partition In the context of an unfolder executor, the replacement of a task by a set of tasks forming (in general) a DAG, whose computation is equivalent to the source task. It is also a particular class of task featurization. Also known as *task decomposition*. 44

task reimplementation In the context of an unfolder executor, the replacement of a task by other, which is then delegated to the unfolder callee. It is also a particular class of task featurization. 44, 45

task-executor taxonomy A classification that relates scenarios in which a single or multiple tasks can be assigned to execution contexts represented by a single or multiple executors. 40

top allocator A software abstraction of several physical memory spaces / storage contexts that provides a memory consistency and coherence layer. A top allocator always manages at least two bottom allocators. 47, 62

top executor A particular class of executor that encompass one (Unfolder) or several (Mapper) executors. 43, 72

unfolder executor A particular class of a top executor that encompass a single executor of any kind, generates tasks via *unfolding* operations applied to the incoming tasks and delegates them to the executor it is associated. xxvii, 43, 69, 71, 93, 94, 130, 133–135, 138–140, 142, 143