Master in Computers Engineering Final Project

List Ranking on Multicore Systems

Author:
Hugo María Vegas Carrasco

Professors in charge:
Thierry Gautier
Manuel Prieto Matías

Master in Computer Science Research
Faculty of Computer Science
Complutense University of Madrid
Year 2009-2010

I don't know half of you as well as I should like;
and I like less than half of you half as well as you deserve

Bilbo Baggings

Autorización

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de M'áster: "List Ranking on Multicore Systems", realizado durante el curso académico 2009-2010 bajo la dirección de Manuel Prieto Matías [y con la colaboración externa de dirección de Thierry Gautier] en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Hugo María Vegas Carrasco

Abstract

In this project we have revisited the implementation of parallel linked-list ranking algorithms on modern Multicore processors. This computation exhibits highly irregular memory referencing patterns, which do not typically benefit from the aggressive mechanisms that integrate current architectures to hide the large latency of main-memory accesses (cache hierarchies, data pre-fetching, ...). Because of this intrinsic characteristic, the performance of any List Ranking algorithm on modern cache-based processors is seriously limited by non-contiguous memory accesses. On a parallel setting, performance is further aggravated since concurrent irregular memory access patterns usually cause more contention for shared memory resources and as such, List Ranking represents a challenging problem for parallel computing.

The development of parallel algorithms for List Ranking has received significant attention in previous literature dating back to the early 80's but most recently, the emerge of many Internet applications that involve extremely large amount of data with linked structures has renewed the interest in List Ranking. Some recent papers have discussed the implementation of these algorithms on modern GPUs but multicore systems still dominate the server market for many applications.

We have focused on the Helman and Jájá's algorithm, since any attempt to achieve satisfactory results with other algorithms such as the famous Wyllie's pointer jumping technique have proven to be ineffective. As main contribution we have shown how the standard Helman and Jájá's implementation can be optimized to reduce the number of non-contiguous memory access. We have also suggested a dynamic parallel version based on the work-stealing paradigm, although preliminary results are still unsatisfactory.

Keywords: *Parallel List Ranking, Prefix Computation, Irregular Algorithms, Pointer Jumping, Wyllie's algorithm, Helman and Jájá's algorithm, Work-stealing*

Resumen

En este proyecto hemos revisado la implementacin de algoritmos paralelos para el *ranking* de listas enlazadas en procesadores multicore. Este tipo de algoritmos exhibe patrones de acceso a memoria fuertemente irregulares que no se benefician de los mecanismos agresivos que integran las arquitecturas actuales para ocultar los costosos accesos a memoria (caches, mecanismos de prebsqueda, ...). Debido a esta caracterstica intrnseca, el rendimiento de cualquier algoritmo para el ranking de listas esta limitado por los accesos a memoria no consecutivos. En los algoritmos paralelos los problemas de rendimiento se agravan ya que los patrones de acceso irregular suelen provocar mayor contencin por recursos compartidos y por lo tanto, continua siendo un importante desafo disear algoritmos eficientes para esta aplicacin.

Desde comienzos de los 80 se han propuesto un buen numero de alternativas para obtener algoritmos paralelos eficientes, pero recientemente ha aumentado el inters debido al auge de muchas aplicaciones, muchas de ellas relacionados con Internet, donde se manejan grandes cantidades de datos almacenadas en estructuras de datos tipo listas enlazadas. Algunos artculos recientes han analizado la implementacin de estos algoritmos en GPUs, pero los sistemas basados en procesadores multicore todava dominan ampliamente el mercado de servidores para muchas de estas aplicaciones.

Nos hemos centrado en el algoritmo de Helman y Jájá's, ya que los intentos por obtener resultados satisfactorios con otros algoritmos, como ha sido el caso de la conocida tcnica de *pointer jumping* propuesta por Wyllie no ha dado resultados satisfactorios. Como principal aportacin mostramos como es posible optimizar el algoritmo standard de Helman y Jájá's para reducir el numero de accesos a memoria no consecutivos. Tambin sugerimos una implementacin dinmica basada en el paradigma de *work-stealing* paradigm, aunque todava, los resultados preliminares no son satisfactorios.

Keywords: *Algoritmos paralelos para el ranking de listas enlazadas, Prefix, Algoritmos irregulares, Pointer Jumping, algoritmo de Wyllie, algoritmo de Helman y Jájá', Work-stealing*

Contents

List of Figures

Outline

In this study we have revisited the implementation of the irregular list ranking algorithm on modern Multicore processors, specifically, we used an Intel Westmere machine. Multicore systems still dominate the high-end server market despite much progress made by accelerators such as GPUs and FPGAs and are currently the most common building block for large scale multiprocessor systems.

In our case we have chosen to develop two versions of the algorithm with different optimizations between them. At the same time we have develop both versions with the help of two current parallel technologies: OpenMP and KAAPI, the first one has better facilities for programming shared memory systems and the second one exploits the dynamic task parallelism thanks to the Athapascan API. For this study we have selected various performance measures for the parallel computing like the Speedup and the way in how the variation of the size of the parallel parts can influence in the final execution results and the importance of choose the correct one. All our results have been contrasted with sequential and parallel versions of the standard List Ranking algorithm confirming the different graphs obtained by experiments.

The organization of this manuscript is as follows. In **Chapter 1** we give some background about the parallel list ranking problem, focusing on describing some of the most popular algorithms found on previous literature and discussing the relevance of this problem. In **Chapter 2** we describe briefly the different parallel methodologies that we have used to address the implementation of the parallel list ranking algorithm on Multicore systems. In **Chapter 3** we present the different implementations we have developed and in **Chapter 4** we speak about the Multicore system that we have used to run our test, we give a short explanation of these

systems and discuss the performance of our implementations against the sequential and standard parallel versions. Finally, in **Chapter 5** we summarize the main conclusions we have found and give some hints about our future research.

Chapter 1

List Ranking

The List Ranking problem is a well known example of an irregular application that may exhibit poor data locality. Such kind of problems tend to be the most challenging to implement efficiently on today's parallel architectures for several reasons, including:

- The parallelization of irregular algorithms is usually limited by irregular memory access patterns to dynamic (pointer-based) data structures whose data-dependence set can only be uncovered at run-time.
- Parallel algorithms for these problems tend to be quite different than the serial algorithms and are often more complicated requiring larger overheads.

This algorithm has been studied on multiple architectures and for different purposes ([1, 2, 3, 4, 5, 6, 7]). In this Chapter we give some background about the List Ranking problem and describe some of the most popular parallel algorithms found on previous literature.

1.1 The List Ranking Problem

Figure 1.1 graphically illustrates the List Ranking problem. Given an arbitrary linked list that is stored in a contiguous area of memory, the List Ranking problem determines the distance of each node to the head of the list [8]. Given that the successor of each node of a linked list can appear anywhere in the memory, this computation exhibits highly irregular memory referencing patterns, which do not typically benefit from the aggressive mechanisms that integrate current architectures to hide the large latency of main-memory accesses (cache hierarchies, data pre-fetching, ...). Because of this intrinsic characteristic, the performance of any sequential List Ranking algorithm on modern cache-based processors is seriously limited by non-contiguous memory accesses. On a parallel setting, performance is further aggravated since concurrent irregular memory access patterns usually cause more contention for shared memory resources and as such, List Ranking represents a challenging problem for parallel computing.

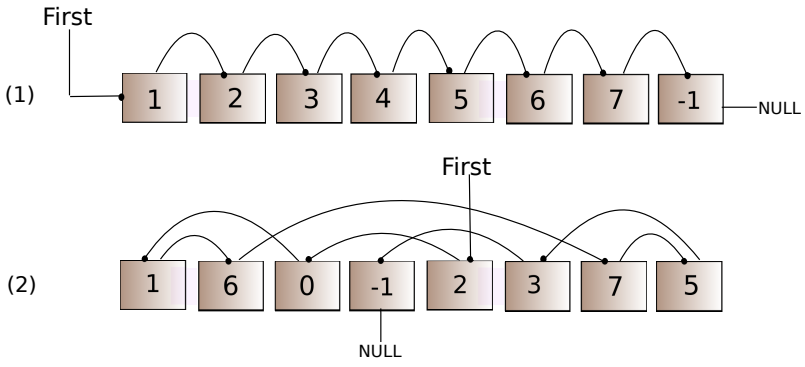

Figure 1.1: List ranking algorithm lists: ordered(1) and random(2)

The development of parallel algorithms for List Ranking has received significant attention in the literature dating back to the work of J.C. Wyllie [9], in which he introduced the **pointer jumping** technique [10]. Some of the reasons that explain such interest are:

- From an application perspective, List Ranking is a representative problem from the class of combinatorial and graph-theoretic applications and for many

algorithms, List Ranking a basic step.

- As mentioned above, memory locality issues limit the performance of this problem, even on sequential implementations and this has motived some attempts to seek parallel solutions for this problem [11].

- From a theoretical perspective, this problem has been a rich source for ideas about the design and implementation of parallel algorithms in general. For example certain arbitration techniques that have been developed for List Ranking have turned out to have much wider application and many ideas about methodologies for parallel algorithms and scheduling have come out of this work [12].

- Most recently, the emerge of many Internet applications that involve extremely large amount of data with linked structures has renewed the interest in List Ranking [13].

1.1.1 The Sequential Prefix Computation Algorithm

List Ranking is an instance of the more general problem of performing a **Prefix Computation** on a linked list [14]. Consider a linked list of n elements stored in arbitrary order in an array X. For each element $X[i]$, we are given $X[i]$.succ, the array index of its successor, and $X[i].data$, its input value for the prefix computation. Then, for any binary associate operator $\otimes$, the prefix computation is defined as [10]:

$$X[i].\text{prefix} = \begin{cases} X[i].\text{data} & \text{if } X[i] \text{ is the head of the list} \\ X[i].\text{data} \otimes X[pre].\text{prefix} & \text{otherwise} \end{cases} \tag{1.1}$$

where pre is the index of the predecessor of $X[i]$. The last element in the list is distinguished by a negative array index in its successor field, and nothing is known about the location of the first element. If all the elements $X[i]$ are 1 and the operator is addition, then prefix reduces to list ranking.

A prefix computation can be performed by a single processor with two passes through the list, a first traversal down the list to identify its head and then a second one to compute the prefix values. The pseudocode for this simple sequential algorithm is as follows [10]:

1. Visit each list element $X[i]$ in order of ascending array index. If $X[i]$ is not the terminal element, then label its successor with the index $X[i]$.succ as having a predecessor.

2. Find the one element not labeled as having a predecessor by visiting each list element X[i] in order of ascending array index — this unlabeled element is the head of the list.

3. Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor pre, set $X[i]$.prefix = $X[i]$.data $\otimes X[pre]$.prefix.

The performance of this algorithm is limited by $O(2n)$ noncontiguous memory access: Step 1 requires at most n noncontiguous memory accesses to label the successors and Step 3 requires at most n noncontiguous accesses to update the successor of each element [10].

The noncontiguous memory accesses of Step 1 can be replaced by a single contiguous memory access by observing that the index of the successor of each element is a unique value between 0 and $n-1$ (with the exception of the tail, which by convection has been set to a negative value). Since only the head of the list does not have a predecessor, it follows that the successor indices comprise the set $\{\ 0,\ 1,\ \ldots\ ,\ h-1,\ h,\ h+1,\ \ldots\ ,\ n-1\}$, where h is the index of the head. Since the sum of the complete set $\{\ 0,\ 1,\ \ldots\ ,\ n-1\}$ is given by $\frac{1,2}{n}(n-1)$, it is possible to identify the head by simply subtracting the sum of the successor indices by $\frac{1,2}{n}(n-1)$. Since the sum of the successor indices can be found by visiting the list elements in order of ascending array index, i.e. with contiguous memory accesses, this alternative Step 1 is able to achieve higher performance. The pseudocode for this improved sequential algorithm is as follows [10]:

1. Compute the sum Z of the successor indices by visiting each list element $X[i]$ in order of ascending array index. The index of the head of the list is $h = (\frac{1,2}{n}(n-1)) - Z$.

2. Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor pre, set $X[i]$.prefix $= X[i]$.data $\otimes X[pre]$.prefix.

The performance of this improved algorithm is limited by $O(n)$ noncontiguous memory accesses and hence it is optimal up to an additive constant if performance is dominated by non-contiguous memory accesses [10].

1.1.2 Pointer Jumping. The Wyllie Algorithm

While it is relatively easy to solve the List Ranking problem in the sequential setting, as described above, algorithms in the parallel setting are quite non-trivial and may differ significantly from the sequential counterparts. Unlike the prefix sum (the scan operation) on an array of elements [15], there is no obvious way to divide a random linked list into even, disjoint, continuous sublists without first computing the rank of each node. Concurrent tasks may also visit the same node by different paths, requiring synchronization to ensure correctness [16]. The range of techniques deployed to arrive at efficient parallel List Ranking algorithms include, among others, independent sets, ruling sets, and deterministic symmetry breaking [17].

In this subsection we describe the parallel List Ranking algorithm proposed by J. C. Wyllie [9] based on the pointer jumping technique, which was probably the first attempt to design a fast parallel algorithm for List Ranking. Algorithm 1 presents a pseudocode of Wyllie's algorithm for a canonical parallel computer with p processors. Initially, it sets the rank $X[i].data = 1$ for all nodes i, except for the last node, whose rank is set to 0. The pointer jumping technique consists of concurrently updating both (1) the rank of each node by adding to it the successor's rank and (2) the successor of each node by that successor's successor. As the technique is applied repeatedly, the successor index of each element in the list is repeatedly updated so that it jumps over its successor until we reach the end of the list. It is also important

to note that if a thread updates either the rank or successor value of an element in between the update operation of another thread that uses the information of that element, the algorithm will fail and hence, the update of the rank and the successor of a given element has to be performed using an atomic operation. By the end of the algorithm, all ranks would have been updated correctly.

Algorithm 1 Wyllie's Algorithm

Input: An array **X** containing the elements of the list. Each node of the list **X[i]** includes an index to its successor (**X[i].Succ**) and has its rank ($X[i].R$) initialized to 1.

Output: Every node of the list holds in **X[i].R** the rank of the element with respect to the head of the list.

```
1: for each element in X do in parallel
2:   while X[i].Succ and X[X[i].Succ] are not the end of the list do
3:     X[i].R = X[i].R + X[X[i].Succ].R
4:     X[i].Succ = X[X[i].Succ].Succ
5:   end while
6: end for
```

On a canonical parallel computer where the number of processor is equal to the size of the list, each thread performs $O(logn)$ steps, and it performs at most a noncontiguous memory access per step. Therefore, the corresponding total number of noncontiguous memory accesses is at most $O(nlog(n))$. This algorithm is non-optimal in view of the existence of a $O(n)$ sequential algorithm.

Other parallel algorithms that improves the complexity of Wyllie's algorithm include those of Vishkin ($5n$ non-contiguous accesses), Anderson and Miller ($4n$ non-contiguous accesses), and Reid-Miller and Blelloch ($2n$ non-contiguous accesses). We have focused our research on the Helman and Jájá's algorithm, which in its worst case requires $\mathrm{O}(\log n + \frac{n}{p})$ non-contiguous accesses (where p are the number of processors) [18, 17], which is described in the next Section.

1.2 Helman and Jájá's Algorithm

The pointer jumping algorithm can be made optimal if we can somehow reduce the size of the list to $\mathrm{O}(\frac{n}{\log n})$ nodes using a linear number of operations. The stan-

dard strategy to achieve optimality would be (1) to partition the input list into approximately $\frac{n}{\log n}$ blocks {Bi}, each containing O($\log n$) nodes; (2) to rank each node within its block (called the preliminary rank) by using an optimal sequential algorithm; and (3) to combine the preliminary ranks using an O($\log n$) time paralleling algorithm. Unfortunately, each block can have $\Omega(\log n)$ sublists, in which case the size of the input of the list to the O($\log n$)time parallel algorithm would not necessarily have been reduced to O($\frac{n}{\log n}$) nodes. Therefore, we need an alternative method.

There are different ways to implement this. The overall strategy for solving the list-ranking problem optimally is outlined next:

1. Shrink the linked list L until only O($\frac{n}{\log n}$) nodes remain.
2. Apply the pointer jumping technique on the short list of the remaining nodes.
3. Restore the original list and rank all the nodes removed in step 1.

Figure 1.2 graphically illustrates the Figure 1.1 for this strategy.

A first implementation works as it is explained below:

1. First we have the list of nodes with two values: the successor of the node and its rank that are initialized to 0
2. For each node, while we do not have the successor of the node or the successor of its successor pointing to the end of the list, we update the rank as the sum of the current rank of the node and the rank of its successor, and we have to update the value of the successor to the successor of its successor too.

This implementation has a problem: we have to update these values in parallel and for that reason it requires some control of this region (for example, using locks). This is a complicated issue because if we do not perform the correct control we can serialize the execution and get a sequential result without any improvement.

Another more faithful option which also follows the main idea explained above is the following:

1. Partition the list into $\frac{n}{p}$ sublists by choosing **splitters**.
2. Processor p_i traverses each sublist, computing the local (with respect to the start of the sublist) ranks of each element and store in R[i].
3. Rank the array of sublists S sequentially on processor p_0
4. Use each processor to add $\frac{n}{p}$ elements with their corresponding splitter prefix in S and store the final rank in R[i]

This is the baseline algorithm used in our implementations.

Figure 1.2: Helman and JáJá's list ranking algorithm

Chapter 2

Parallel Methodologies

This chapter explains which are the two parallel technologies that we have used to develop our version of the list ranking algorithm. The first one is OpenMP that is a good technology choice due to the facility of programming shared memory systems with parallel codes. The second one is KAAPI and we have select this other for its facilities to exploit parallelism in dynamic tasks.

2.1 OpenMP

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Microsoft Windows platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different

processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an "id" attached to it which can be obtained using a function (called omp_get_thread_num() in C/C++ and OMP_GET_THREAD_NUM() in Fortran). The thread id is an integer, and the master thread has an id of "0". After the execution of the parallelized code, the threads "join" back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. "Work-sharing constructs" can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both Task parallelism and Data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labelled "omp.h" in C/C++.

A compiler directive in C/C++ is called a pragma (pragmatic information). Compiler directives specific to OpenMP in C/C++ are written in codes as follows:

#pragma omp rest_of_pragma

2.1.1 Thread creation

With *omp parallel*. It is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original process will be denoted as master thread with thread ID 0.

2.1.2 Work-sharing constructs

used to specify how to assign independent work to one or all of the threads.

- omp for or omp do: used to split up loop iterations among the threads, also called loop constructs.
- sections: assigning consecutive but independent code blocks to different threads
- single: specifying a code block that is executed by only one thread, a barrier is implied in the end
- master: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

2.1.3 OpenMP clauses

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as data sharing attribute clauses by appending them to the OpenMP directive. The different types of clauses are

2.1.3.1 Data sharing attribute clauses

- shared: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- private: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
- default: allows the programmer to state that the default data scoping within a parallel region will be either shared, or none for C/C++, or shared, firstprivate,

private, or none for Fortran. The none option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.

- firstprivate:like private except initialized to original value.
- lastprivate: like private except original value is updated after construct.
- reduction: a safe way of joining work from all threads after construct.

2.1.3.2 Synchronization clauses

- critical section: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- atomic: similar to critical section, but advise the compiler to use special hardware instructions for better performance. Compilers may choose to ignore this suggestion from users and use critical section instead.
- ordered: the structured block is executed in the order in which iterations would be executed in a sequential loop
- barrier: each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- nowait: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

2.1.3.3 Scheduling clauses

- schedule(type, chunk): This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:

1. static: Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter "chunk" will allocate "chunk" number of contiguous iterations to a particular thread.

2. dynamic: Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter "chunk" defines the number of contiguous iterations that are allocated to a thread at a time.

3. guided: A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter "chunk"

2.1.3.4 IF control

- if: This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

2.1.3.5 Initialization

- firstprivate: the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.

- lastprivate: the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both firstprivate and lastprivate.

- threadprivate: The data is a global data, but it is private in each parallel region during the runtime. The difference between threadprivate and private is the global scope associated with threadprivate and the preserved value across parallel regions.

2.1.3.6 Data copying

- copyin: similar to firstprivate for private variables, threadprivate variables are not initialized, unless using copyin to pass the value from the corresponding global variables. No copyout is needed because the value of a threadprivate variable is maintained throughout the execution of the whole program.
- copyprivate: used with single to support the copying of data values from private objects on one thread (the single thread) to the corresponding objects on other threads in the team.

2.1.3.7 Reduction

- reduction(operator — intrinsic : list): the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in "operator" for this particular clause) on a datatype that runs iteratively so that its value at a particular iteration depends on its value at a previous iteration. Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the datatype, then increments the datatype in order so as to avoid racing condition. This would be required in parallelizing Numerical Integration of functions and Differential Equations, as a common example.

2.1.3.8 Others

- flush: The value of this variable is restored from the register to the memory for using this value outside of a parallel part
- master: Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.

2.1.4 User-level runtime routines

Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc.

2.1.5 Environment variables

A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc. For example OMP_NUM_THREADS is used to specify number of threads for an application.

2.2 KAAPI

KAAPI means Kernel for Adaptative, Asynchronous Parallel and Interactive programming. It is a C++ library that allows to execute multithreaded computation with data flow synchronization between threads. The library is able to schedule fine/medium size grain program on distributed machine. The data flow graph is dynamic (unfold at runtime). Target architectures are clusters of SMP machines.

Main features are:

- It is based on work-stealing algorithms
- It can run on various processors
- It can run on various architectures (clusters or grids)
- It contains non-blocking and scalable algorithms

2.2.1 Programming Model

- task description: What is a task ?
- shared memory: Distributed memory model
- task samples: some kind of tasks

KAAPI is a middleware working on Dynamic Acyclic Data flow graphs. Once given this graph, it can dynamically schedule it using a work-stealing algorithm.

But most distributed computing users are familiar with message passing paradigm, and KAAPI uses an other paradigm:

1. Describe the task of your graph / program
2. Describe the dependencies between your task

Once this is done, KAAPI will schedule the tasks in an efficient way, making sure that

1. All dependencies are respected
2. Parallelism between independent tasks is used as much as possible

Of course, this will not work as expected on all kind of graph, but it has been proven to be an asymptotically optimal way to schedule tasks from a divided and conquer algorithm, based on fork and join calls.

2.2.1.1 Task description

KAAPI library is based on the Athapascan's API. Athapascan is a macro data-flow application programming interface (API) for asynchronous parallel programming. The API permits to define the concurrency between computational tasks that synchronize on the access through a global distributed memory. Parallelism is explicit and functional but detection of synchronizations is implicit. The semantic of Athapascan is sequential; then, an Athapascan's program is independent from the target parallel architecture (cluster or grid). The execution relies on an interpretation algorithm that computes a macro data-flow graph. The graph is direct and acyclic (DAG) and it encodes the computation and the data dependencies (read and write). It is used by the runtime support to schedule the tasks and map the data onto the target architecture. Implantation is based on the use of lightweight process (threads) and one-sided communications (actives messages).

A task in Athapascan is more or less a function object with no side effect. A task execution is somewhat similar to a standard procedure call (Tasks are dynamically created at run time). The only difference is that the created task's execution is fully asynchronous, meaning the creator is not waiting for the execution of the created task to finish to continue with its own execution. So an Athapascan program can be seen as a set of tasks scheduled by the library and distributed among nodes for its execution.

A task corresponds to the execution of a function object, i.e. an object from a class (or structure) having the *void operator()(...)* defined:

```
struct user_task
        void operator() ( /* formal parameters */ )
        {
                /* ... */
        }
};
```

A sequential (hence not asynchronous !) call to this function class is written in C++:

```
user_task() ( /* effective parameters */ ) ;
```

And an asynchronous call to this task is written in Athapascan:

```
a1::Fork <user_task> () ( /* effective parameters */ ) ;
```

Chapter 3

Scientific Development

In this section we present two optimizations of Helman's and Jájá's parallel List Ranking Algorithm. Here we offer the explanation about how we have done our structures and why, and we explain all the features that we have take into account to develop both versions too.

3.1 Minimizing Non-Contiguous Memory Access

We assume a multiprocessor with p cores. The instance of the problem is given by a list stored as an array L of nodes. The size of the list is n. Each node data structure contains two fields:

```
struct node
{
        index nS;
        index R;
};
```

Where nS is the index of the successor in the array L. R is the rank of the node in the list. After initialization, R = -1 for all nodes. The last node has nS = -n as marker of the end of the list. For all other nodes, the value of the successor

nS index is 0. In [2], the author presents a memory model to try to explain the cost of accessing the elements of the list. In the abstract the discussion is: the cost of k accesses to non contiguous memory location are greater than k accesses to k contiguous memory locations. This remark drives some algorithmic decision.

The algorithm is "efficient": it means that the total number of operations is, up to a constant, the same as the number of operations in the sequential algorithm and that the expected parallel time is bounded, with high probability, by $O(\frac{T_{seq}}{p})$, $Tseq$ is the sequential time.

Nevertheless, there are two possible improvements based on the reduction of the number of memory accesses or by transforming non-contiguous accesses by contiguous accesses to memory location.

1. **SLIndex**: If, for each element of the list L and during the local list ranking computation, each node stores its index (in 0, .., $s1$) of the sublist in sL, then the last step may be done using contiguous accesses. The last step will becomes: the core Pi will update the rank for elements with index in the range $[ib, (i+1)b)$ (and $[ib, n)$ for the last core)

```
for (index i=0; i < n; ++i)
{
        L[i].R += sL[sLindex[i]].pR;
};
```

   If the data structure for the sublist can be stored in the cache (that will be the case even for large list, because the number of sublists is small) and the write instruction uses a non-temporal write instruction(to not store *L[i].R* into the cache in order to not pollute the cache), it could improve a lot the time for the last step.

   But this solution add an extra structure (**sLindex**) with the *ids* of the sublists.

2. **Accumulated rank**: In the same way, the local rank update of the step 2 has n non contiguous write to the main memory to store *L[i].R*. And then at step 4, this value is updated. The non contiguous write could be eliminated:

(a) step 2: each core only reads the nodes into a sublist and returns the last **computed rank accumulated** into the *lr_head_list* function, but without writing, to the main memory, the value into each node data structure.

(b) step 3: it remains unchanged: the prex ranks are computed in the same way.

(c) step 4: the algorithm is like the list ranking for a sublist with the initial rank of the head set to the prex rank. Each time the core traverses a node of the list, it compute the global rank as:

$$\text{L[i].R} = \text{sL[j].pR} + \text{L[pred(i)].R} + 1$$

In this way, we avoid the need for the n non contiguous writes at step 2 and the n non contiguous read at step 4, where the accumulation is replaced by a write.

These two idea of improving the complexity (in term of memory accesses) are not compatible together.

3.2 Other Problems and optimizations

In this section we describe other problems we have found and how we have faced them.

3.2.1 The hyperthreading affinity

The architecture that have used allows two threads per core and this implies a reduction in the overall speed of the cores if we have more than one thread per core. For this reason we have come to the conclusion that we use only one thread per core to avoid this problem. The solution we have applied is use the function "sched_getaffinity" that lets us to choose in which core we want to schedule each thread, so that we can ensure that in each core we have only one thread into the the parallel regions.

3.2.2 The false sharing problem

another problem we have to solve is the false sharing problem. This problem has relationship with the cache line size of the architecture and the cache organization.

The new cache coherence protocol of the Westmere architecture is known as the MESIF (Modified, Exclusive, Shared, Invalid, Forward) protocol, which is a modification of the popular MESI protocol. Each cache line is in one of the five states:

- Modified - the cache line is only present in the current cache and does not match main memory (dirty). This line must be written back to main memory before any other reads of that address take place.
- Exclusive - The cache line is only present in the current cache and matches main memory (clean).
- Shared - The cache line is clean similar to the exclusive state, but the data has been read and may exist in another cache. This other cache should be updated somehow if the line changes.
- Invalid - The cache line is invalid
- Forward - This cache line is designated as the responder to update all caches who are sharing this line.

With the extra "Forward" state, the excessive responding among shared cache line is eliminated.

Moreover, the Westmere architecture has a cache line size of 64 Bytes. If we have structures which their sizes are different to that size or its multiples we found that we will get a lot of invalidations in our computation because the data of the structures are not distributed in a regular way into the memory. For this reason we have had to adapt our structures to this size or its multiples.

Chapter 4

Experimental Results

In this section we present all the features that we have taken into account our study. First, in Section 4.1 we describe briefly the experimental platform we have used to run our tests. In Section 4.2 we analyze the relationship between the number of splitters that we can choose for our sublists and the execution times that we get depending on this selection. Section 4.3, and 4.5 show the results of the sequential List Ranking, the Wyllie's algorithm and the standard Helman's and Jájá's algorithm respectively. In Section 4.6, we discuss the results of our optimized OpenMP versions and we compare them with the hyperthreading version. Finally in Section 4.7 we give some preliminary results of the KAAPI implementation.

4.1 Intel Westmere Machine

Westmere (formerly Nehalem-C) is the name given to the 32 nm die shrink of Nehalem. The first Westmere-based processors were launched on January 7, 2010 and branded as members of the Core i3, Core i5, and dual-core mobile Core i7 families.

The next table shows the main features of the Intel Westmere machine that we have used to run our test for this study.

Processor	Xeon X5670 2 chips x 6 cores (2,93 GHz)	
	L1 Cache (per Core)	32KB
	L2 Unified Cache	256KB
	L3 Unified Cache	12MB
Memory	48 GB, 32 GB/s, 3xDDR3-1333	
Operative System	GNU/Linux 2.6.32-5-amd64	
G++ Compiler 4.4.5	-O3 -fopenmp -lm	

Table 4.1: Intel Westmere Information System

4.2 Relationship between the number of splitters and the execution time

In this section we want discuss about one specific feature of the parallel implementations of the List Ranking algorithm. As we know, the standard version of Helman's and Jájá's List Ranking divides the main list into different sublist with potentially different sizes. In [2] Helman and Jájá explained that an optimal value could be $p \log n$ where p is the number of threads and n the size of the list. In [16] they say that the number of splitters has to vary between $\frac{n}{\log n}$ and $\frac{n}{2 \log n}$, but this measure is not so good in our case because it is only useful when we have a big amount of threads like on GPUs. In our case we have run some tests using different number of splitters with the hole rank of threads and using four representative list sizes (32K, 512K, 4M and 64M). The following Figures analyze the effect of this parameter for the random lists in the standard version of the algorithm:

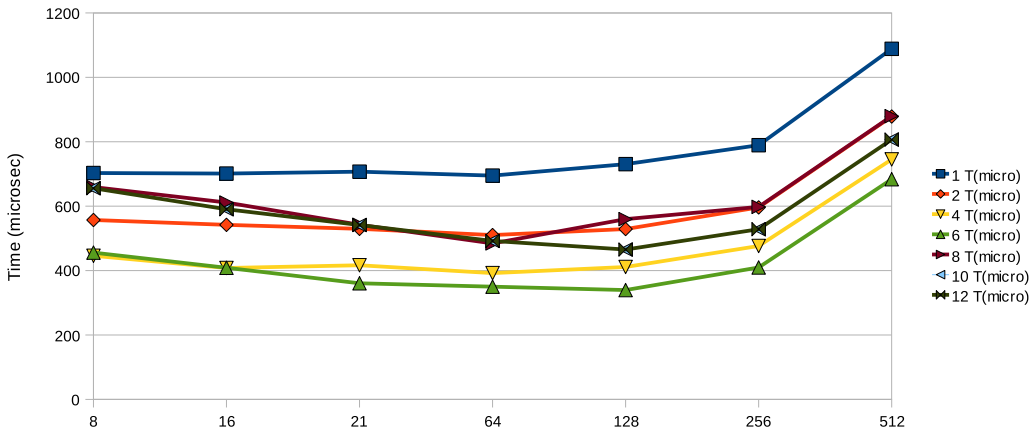

Figure 4.1: Execution Time of Helman's and Jájá's algorithm for a 32K random lists using different number of splitters

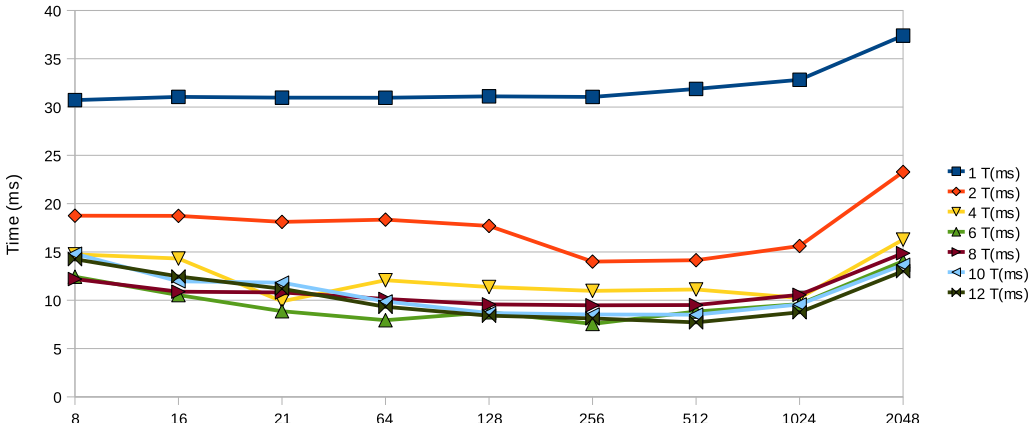

Figure 4.2: Execution Time of Helman's and Jájá's algorithm for a 512K random lists using different number of splitters

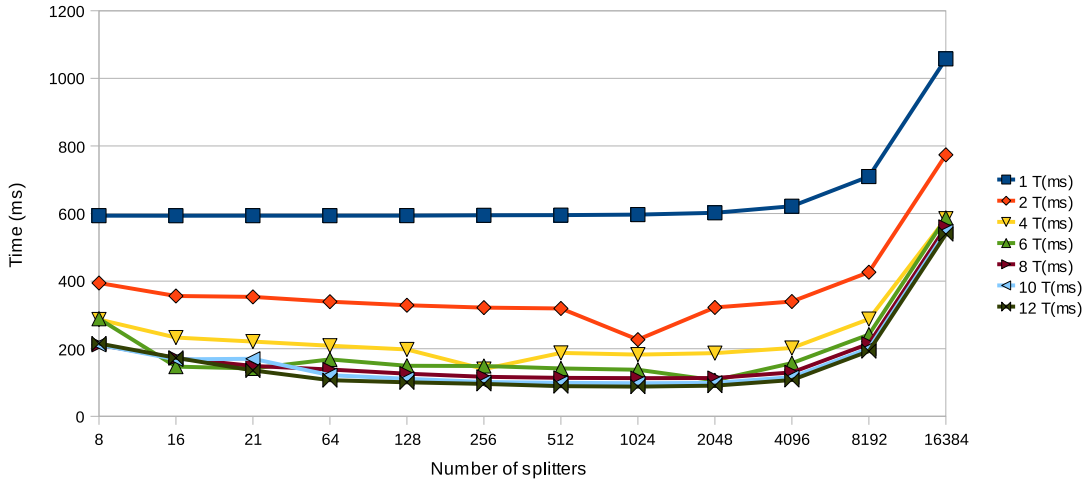

Figure 4.3: Execution Time of Helman's and Jájá's algorithm for a 4M random lists using different number of splitters

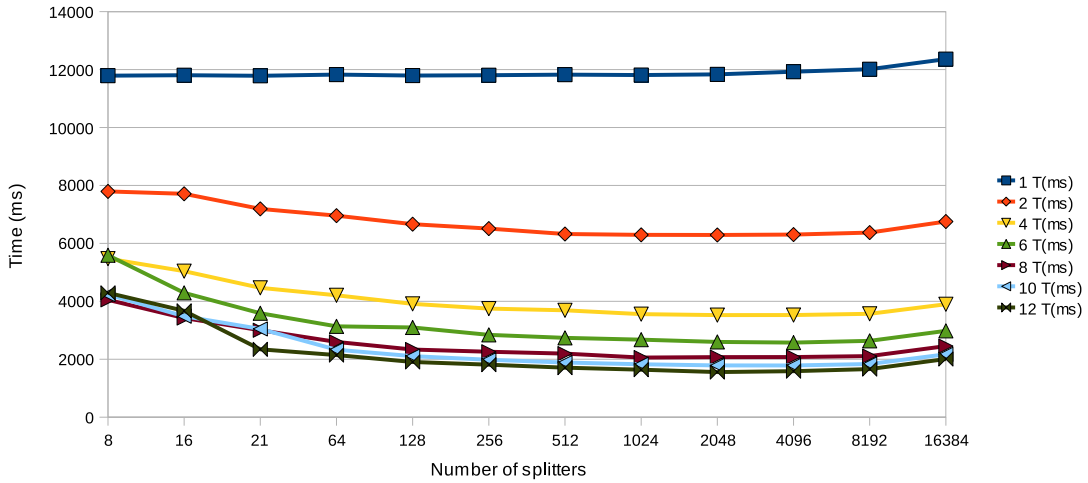

Figure 4.4: Execution Time of Helman's and Jájá's algorithm for a 16M random lists using different number of splitters

Figures 4.1 to 4.4 show the results for the random lists. For the smallest list, the optimal value varies between 64 splitters (for up to 8 threads) and 128 splitters (for 10 and 12 threads) which validates the $p \log n$ model predicted by Helman and Jájá. Nevertheless performance remains similar up to 256 splitters. Beyond this limit, the execution time grows exponentially.

As the list size increases the range of optimal number of splitters also increases and we do not observe large degradations in the execution time the number of splitters grow beyond the theorical optimal values.

The following Figures analyze the effect of this parameter for the ordered lists in the standard version of the algorithm:

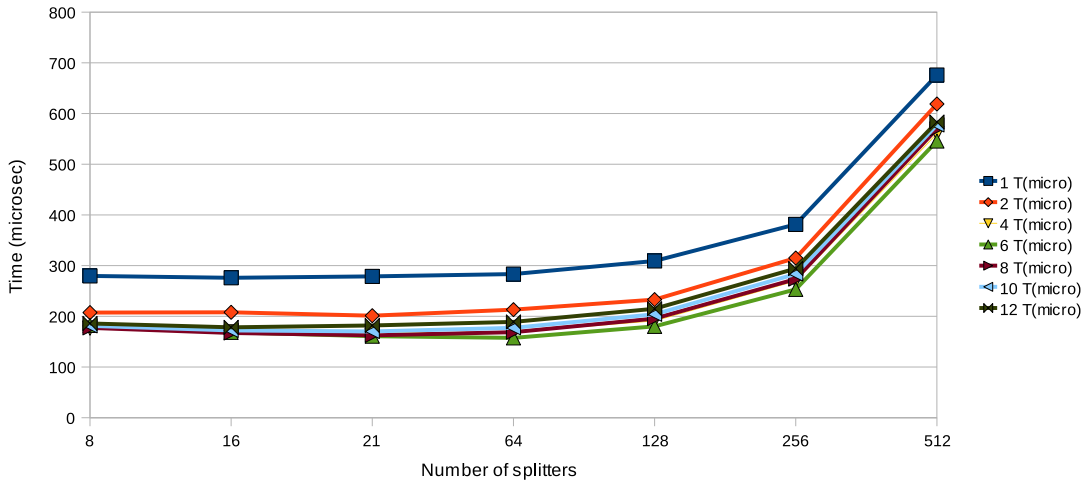

Figure 4.5: Execution Time of Helman's and Jájá's algorithm for a 32K ordered lists using different number of splitters

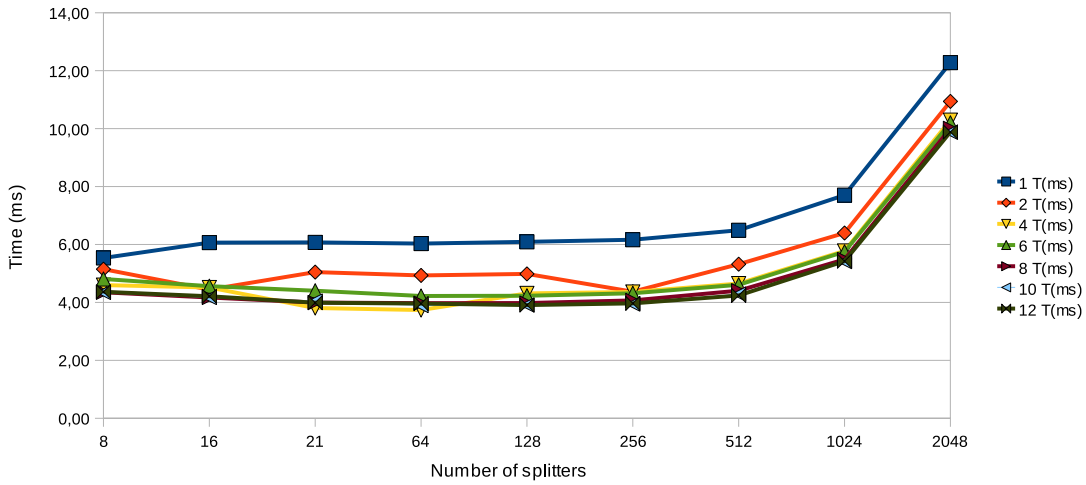

Figure 4.6: Execution Time of Helman's and Jájá's algorithm for a 512K ordered lists using different number of splitters

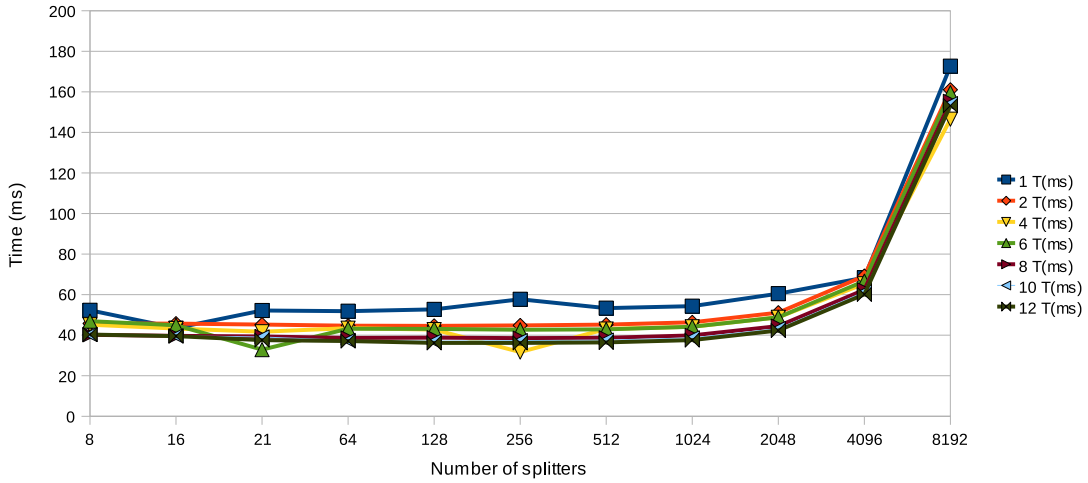

Figure 4.7: Execution Time of Helman's and Jájá's algorithm for a 4M ordered lists using different number of splitters

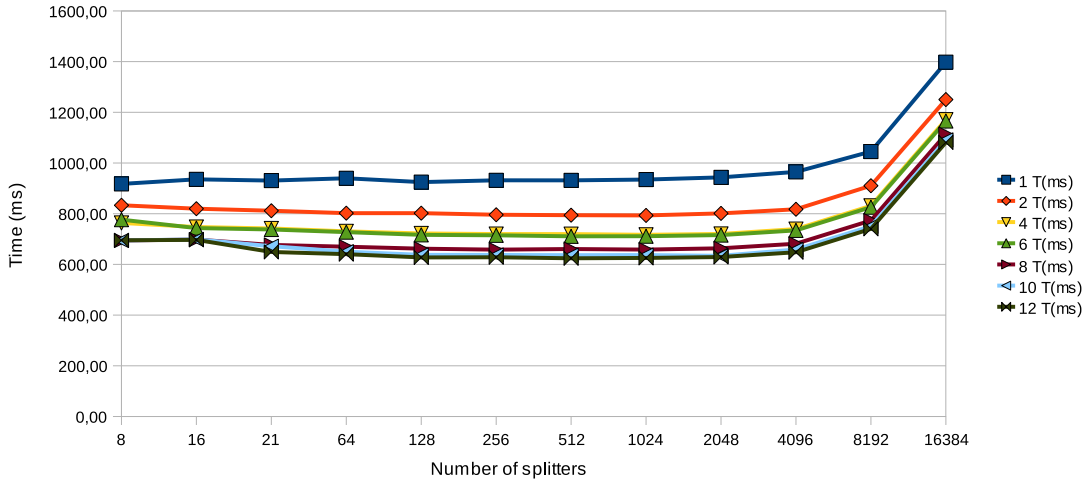

Figure 4.8: Execution Time of Helman's and Jájá's algorithm for a 64M ordered lists using different number of splitters

Figures 4.5 to 4.8 show that up to a certain limit which is really much larger than the optimal value predicted by Helman and Jájá)the performance does not vary at all. Beyond this limit, performance deteriorate significantly.

This Figures exhibit a similar pattern. We do not really mind what number of

splitters we choose at the beginning because they seem to have similar results for a big rank of them. Nevertheless, like in the random versions if we have too much splitters we get worst results than with an intermediate number.

With all these tests we prove that the choice of the number of splitters at the beginning is a critical factor and we have to take it into account if we want to obtain better results.

4.3 Sequential List Ranking

In this Section of the study we analyze the sequential List Ranking. These results are important because we confront our results to those obtained here. First we will show the results for ordered lists and afterwards the random lists counterparts. It is important to take into account both results because the ordered list takes advantage of spatial locality and gives better execution times.

The following Figures analyze the execution times for the random lists and the ordered lists:

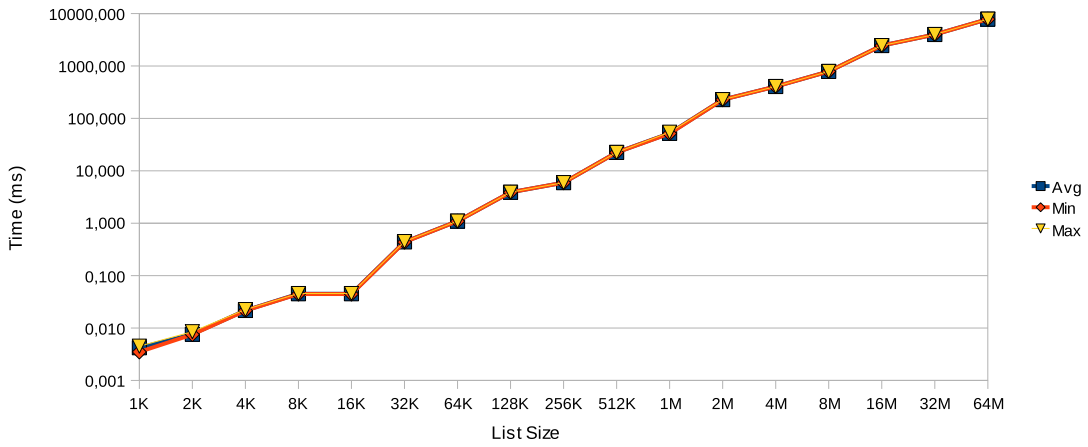

Figure 4.9: Times of the sequential version of List Ranking using random lists

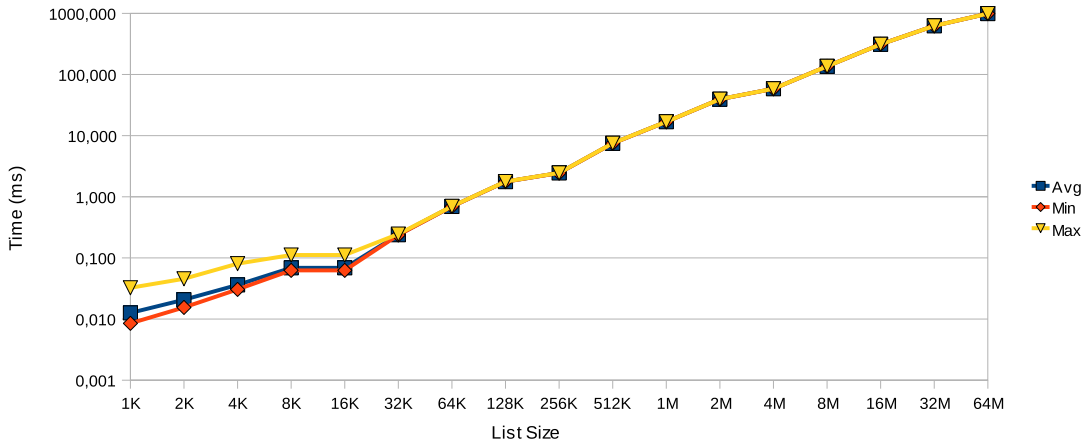

Figure 4.10: Times of the sequential version of List Ranking using ordered lists

As shown in Figures 4.9 and 4.10 ordered lists are about an order of magnitude lower than the random lists counterparts. In both cases, the execution times grow linearly with the list sizes.

4.4 Wyllie's Algorithm Results

This section shows the Wyllie's Algorithm results. As we said on subsection 1.1.2, it is based on the pointer jumping technique, and was probably the first attempt to design a fast parallel algorithm for List Ranking. This is one of our challenge sections. We have tried to implement the standard Wyllie's algorithm applying two optimizations. As we explained before, the steps 3 and 4 of this algorithm must be done in an atomic way to assure that two threads do not update the same rank or successor at the same time.

Both implementations are described below:

1. **Barrier-Based Implementation:** In this implementation we duplicate the elements of each list node with an old an new fields. In each one of the $\log(n)$ iterations of the algorithm we use one of this fields to store the the information,

and the other one in the next iteration. Thanks to this we can avoid the use of critical sections, barriers or other control flow elements.

2. **Hash-Based Implementation:** Multiple locks to control the accesses to common parts of the list. We do a hash control based on the module of each position *id*. With this strategy we let some threads work concurrently as long as they work on different elements of the list.

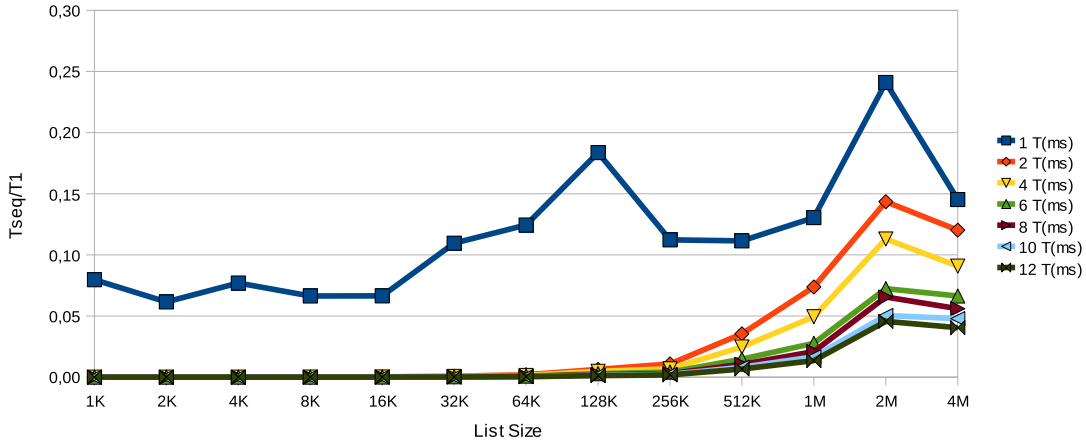

Figure 4.11: Speedup of the Wyllie's algorithm using the Barrier-Based Implementation

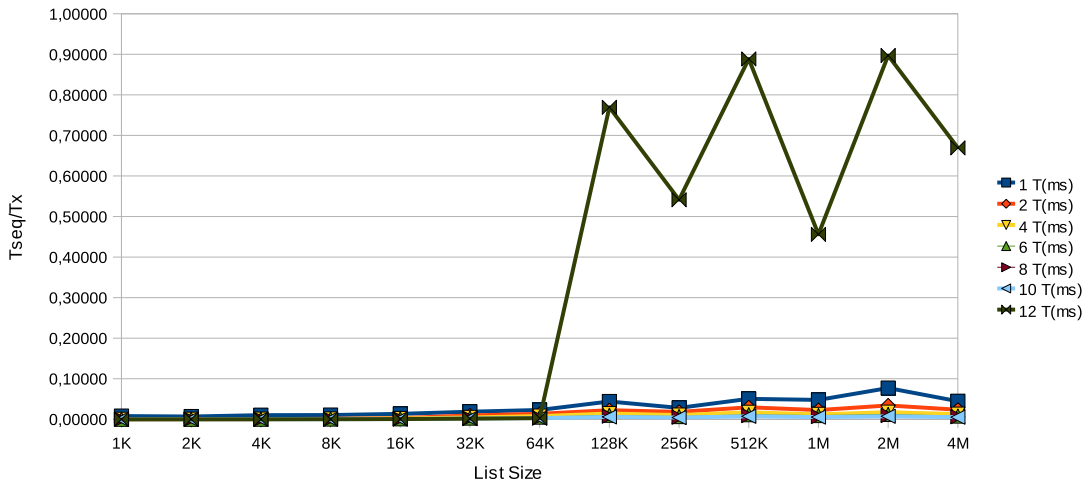

Figure 4.12: Speedup of the Wyllie's algorithm using the Hash-Based Implementation

Unfortunately both implementations give unsatisfactory results. Wyllie's algorithm seems to not be a good approach to this problem in our architecture. If we take a look to the nature of the problem we see that the parallel part works like if one thread had only one element instead a group of them, for that reason this type of implementation has better results in a parallel environment with more threads like in GPUs than in a Multicore system with a few threads.

4.5 Standard Helman's and Jájá's

In this section we illustrate the results of the standard List Ranking algorithm that Helman and Jájá described on [2].

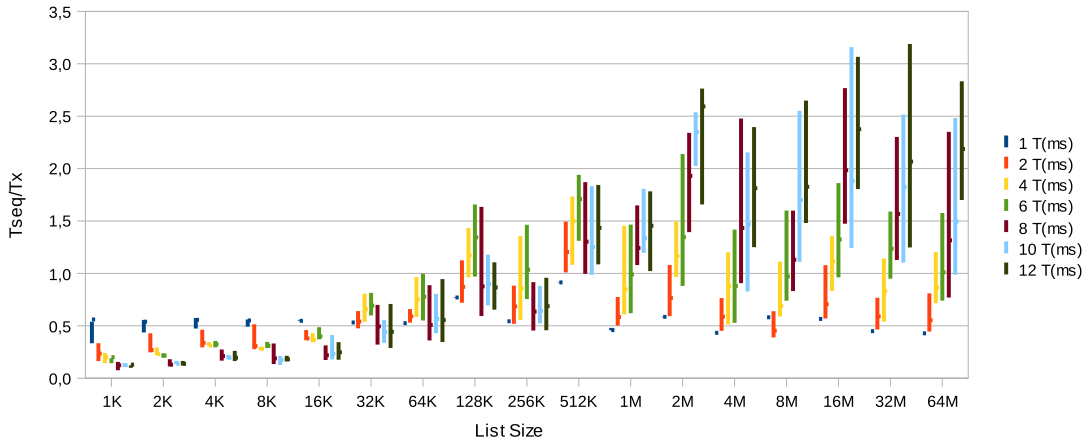

Figure 4.13: Speedup of the Standard Helman's and Jájá's algorithm using random lists

For each problem size we have shown the average, the maximum and the minimum speedups over the sequential version using optimal number of splitters. The speedups are much lower than could be expected (really much lower than the speedup figures reported for older shared-memory architectures).

In any case, we achieve certain improvements for list sizes larger than 512K elements.

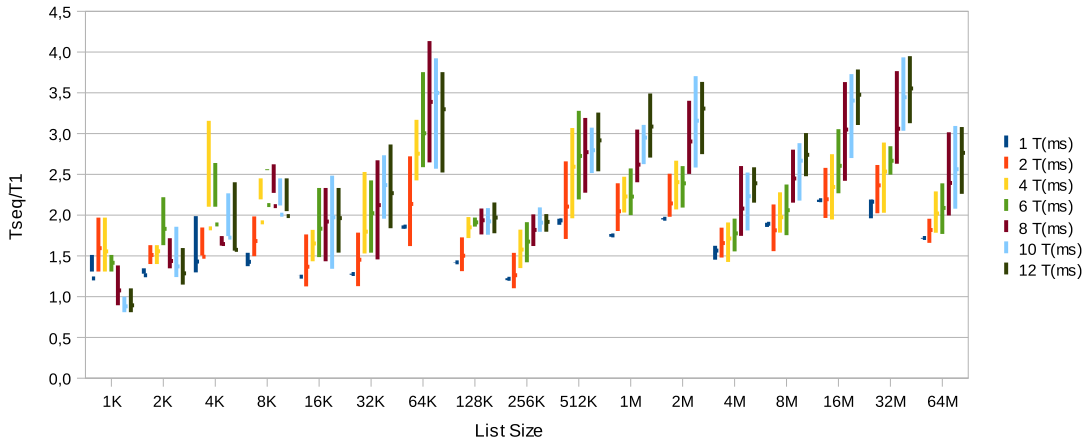

Figure 4.14: Speedup of the Standard Helman's and Jájá's algorithm using ordered lists

For random data, the speed up figures tend to be lower than the ordered counterparts, but we have different speedup patterns. For random lists speedups tend to grow with the list size, while for ordered lists there are some sizes for which the speedup go down significantly.

4.6 Optimized Helman's and Jájá's

In this section we show the results of our optimized Hellman's and Jájá's algorithm:

Next figures show the random list results to our two optimizations:

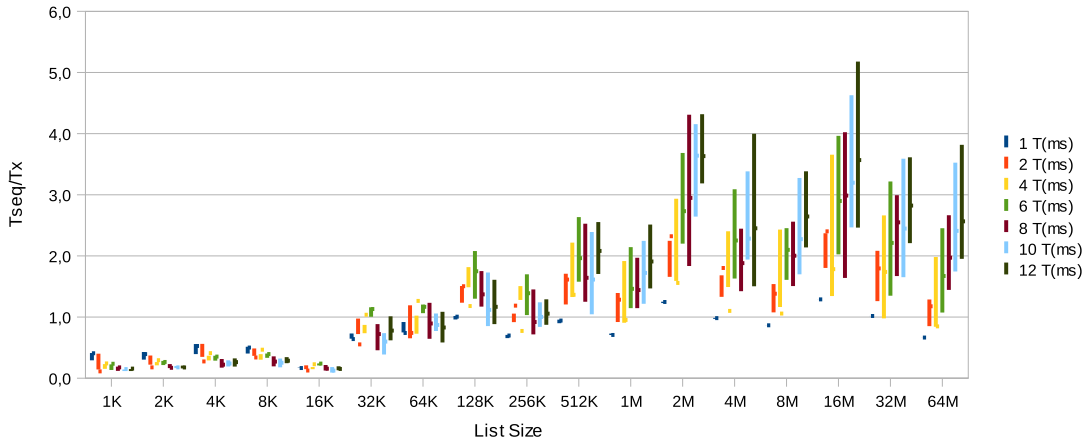

Figure 4.15: Speedups of the SLIndex Helman's and Jájá's using random lists

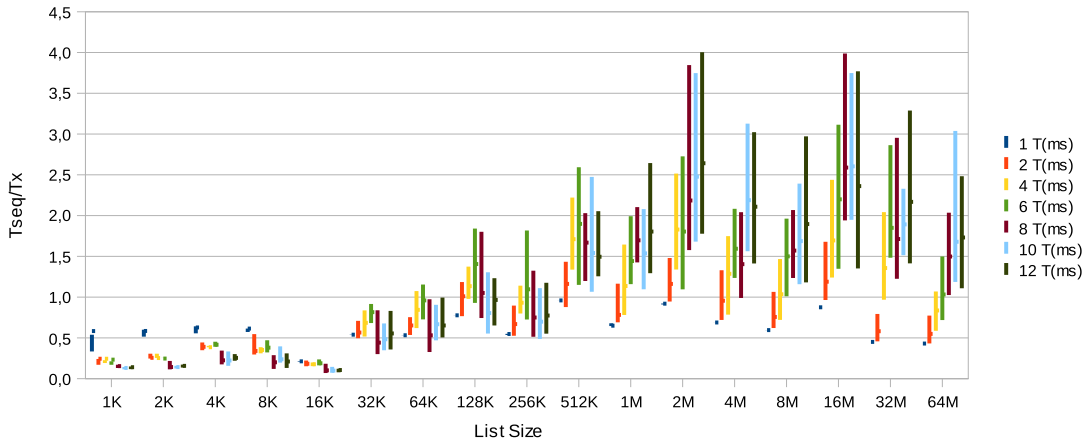

Figure 4.16: Speedups of the Accumulated Rank Helman's and Jájá's using random lists

Figures 4.15 and 4.16 illustrate the results of our optimizations using random lists. The SLIndex is able to outperforms the standard algorithm and the Accumulated Rank versions. With the SLIndex, some improvements are achieved for list sizes larger than 128K and beyond this size results are closer to the standard Helman's and Jájá's with ordered lists.

Next figure show the ordered list results to our Accumulated Rank optimization:

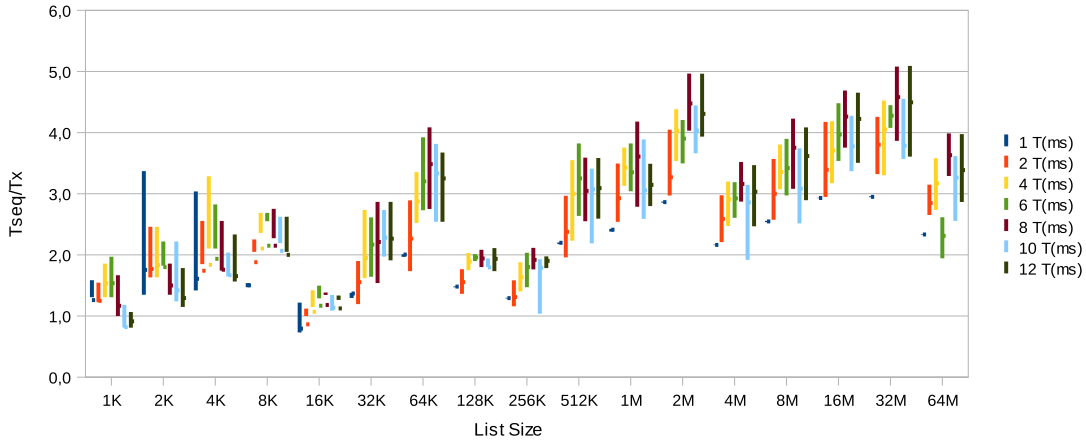

Figure 4.17: Speedups of the Accumulated Rank Helman's and Jájá's using ordered lists

For ordered lists, the SLindex behaves similarly to the standard Helman's and Jájá's, while the Accumulated Rank version provides some additional improvements, although it exhibits a similar speedup pattern.

Finally, we illustrate in Figures 4.18 and 4.19 the impact of enabling hyper-threading.

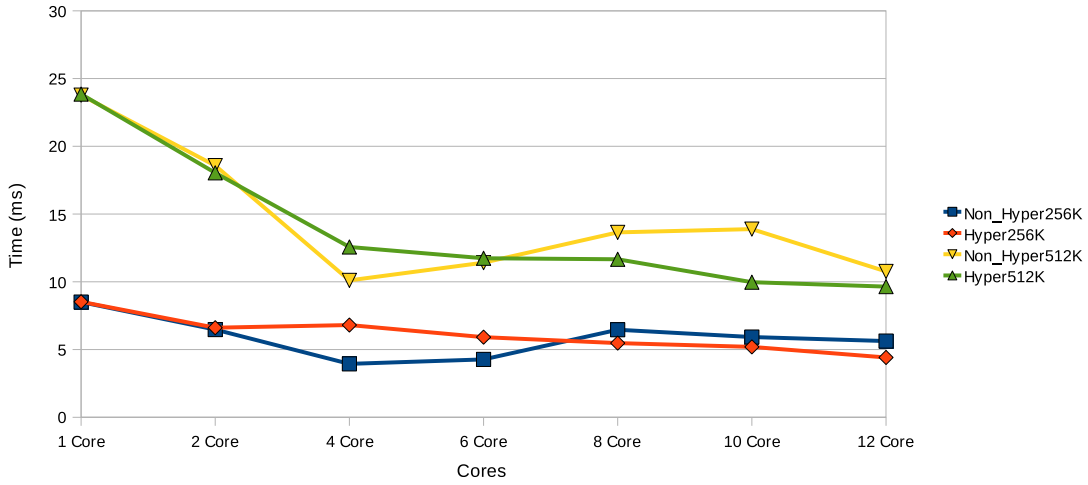

Figure 4.18: Execution Times of the SLIndex Version with and with out hyper-threading for 256K and 512K list sizes

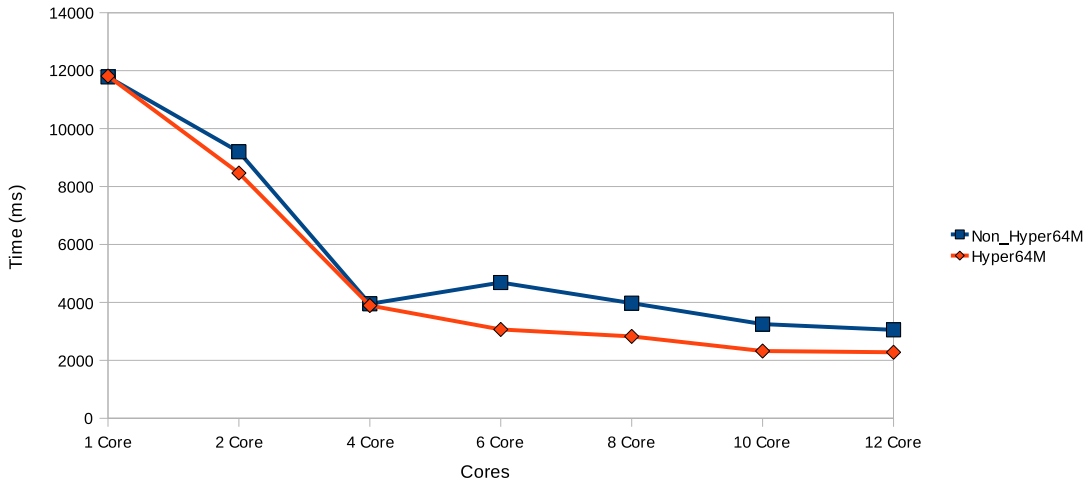

Figure 4.19: Execution Times of the SLIndex Version with and with out hyper-threading for 64M list size

For the SLIndex version, using small lists the maximum improvement achieved with hyperthreading is about 7%, while for the large sizes it grows to around 25%.

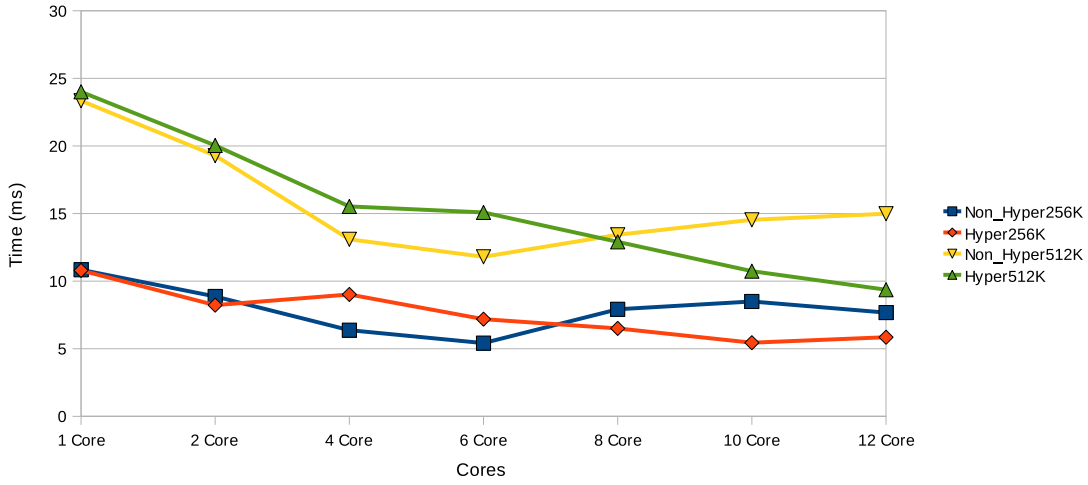

Figure 4.20: Execution Times of the Accumulated Rank Version with and with out hyperthreading for 256K and 512K list sizes

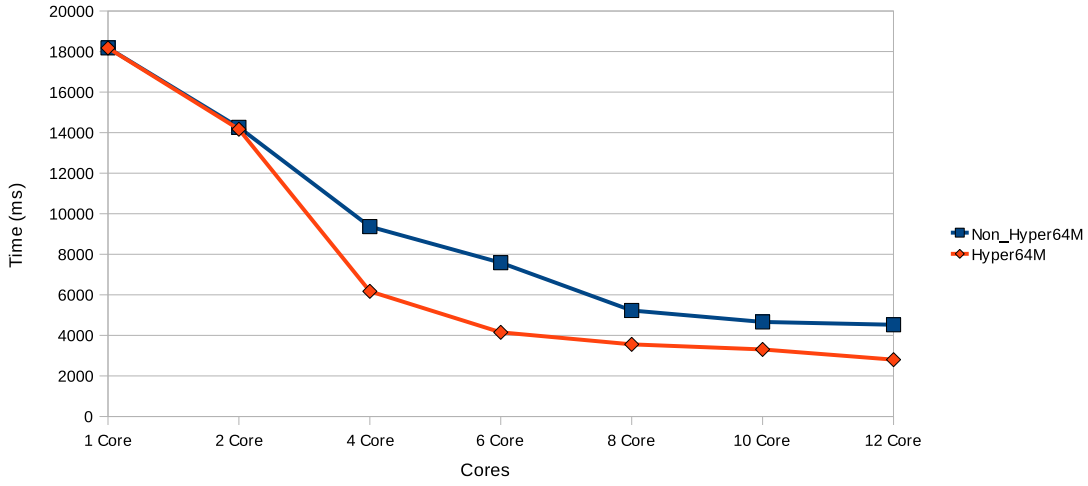

Figure 4.21: Execution Times of the Accumulated Rank Version with and with out hyperthreading for 64M list size

For the Accumulated Rank version, using small lists the maximum improvement achieved with hyperthreading is about 9%, while for the large sizes it grows to around 40%.

Overall, these results indicate that hyperthreading is beneficial for large lists,

since it is able to mitigate memory access penalties.

4.7 KAAPI

To end up this chapter, we present our KAAPI version features and its results.

The KAAPI implementation has several differences with the C standard version. The first important difference are the **structures**. Structures in KAAPI have an special notation. For our structures is like this:

```
/** NODES OF THE LIST */
struct node
{
  long int nS; // Successor
  long int R; // Rank

  //empty constructor
  node()
  :nS(-1),R(1){}

  //copy constructor
  node( const node& nod)
  :nS(nod.nS),R(nod.R){}
};

//packing operator
a1::OStream& operator<< (a1::OStream& out, const node* nod)
{
  return out;
}

//unpacking operator
a1::IStream& operator>> (a1::IStream& in, node* nod)
```

```
{
  return in;
}
```

The *a1::OStream* is the serializing operator which puts into the output_stream the information needed to reconstruct an image of x using the operator >>. The *a1::IStream* is the deserializing operator which takes from the input_stream the information needed to construct the object x; it initializes x with the value related to the information from input_stream.

Other difference is the structure of the parallel functions. If we want to parallelize a function in KAAPI we have to declare them like *structs*. One example of this is our **LocalRankingPhase**:

```
struct LocalRankingPhase
{
  LocalRankingPhase(){}

  void initList(std::vector<node> *L)
  {
    pL = L;
  }

  node operator() (struct node_subl &nod)
  {
    long int pos = nod.nS_cp;
    long int j=1;
    long int n_sub=0;

    while(n_sub == 0)
    {
      if((*pL)[pos].nS >= 0)
      {
        pos = (*pL)[pos].nS;
        j++;
```

```
      }
      else
      {
        if((*pL)[pos].nS == -(num_elements))
          j++;
        n_sub = 1;
      }
    }
    if((*pL)[pos].nS == -(num_elements))
      nod.next = -1;
    else
      nod.next = -((*pL)[pos].nS)-1;
    nod.size = j;
  }
private:
  static std::vector<node> * pL;
};
std::vector<node> * LocalRankingPhase::pL = NULL;
```

Our main structure where we call to all the functions is called **do_main** and look like this:

```
struct do_main
{
  void operator()(int argc, char * argv[])
  {
    ...
  }
};
```

The do_main task is the only one which has specific parameters.

The last part is the **main**. The main function calls to our main structure and has always the same look in KAAPI:

```
int main(int argc, char * argv[])
```

```
{
  long int i;

  a1::Community com = a1::System::join_community( argc, argv );
  try
  {
    /* Start computation by forking the main task*/
    a1::ForkMain<do_main>()(argc, argv);

    com.leave();

    a1::System::terminate();
  }
  catch (Error& E) {
    ...
  }

  return 0;
}
```

Usually the community is defined in the main method of the program. Athapscan reads its parameters from the program arguments. They are used to initialize the *Community*. The starter is hit once we ask to leave the community. A community can only be left if it contains no task. The *com.leave()* is important, it computes the termination of the program, checking whether the local task list is empty or not.

In Figure 4.22 we show the results of this preliminary KAAPI version:

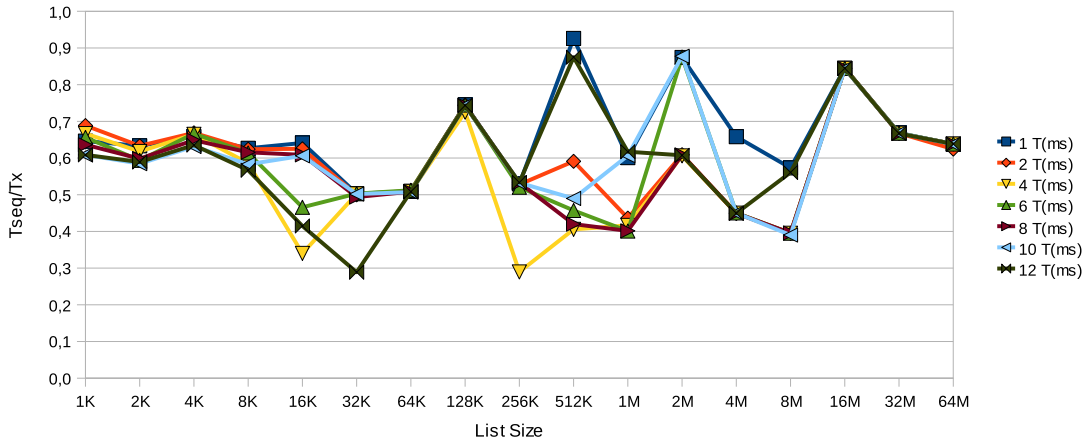

Figure 4.22: Speedups for the List Ranking algorithm with KAAPI

Unfortunately, performance is still very poor and we are currently working on this implementation.

Chapter 5

Conclusions and Future Work

In this project we have shown how to optimize list-ranking algorithms on modern Multicore system. For this kind of memory-intensive applications, performance is dominated by the memory access pattern and hence for random lists, it is quite difficult to achieve satisfactory results. A remarkable conclusion of this project is that the attainable speedups of well-known algorithm proposals on current multicore systems are really much lower than the reported speedups on some previous systems such as older SMP multiprocessors. In fact, algorithms such as the well-known pointer jumping technique proposed in the late 70's by Wyllie, do not provide any improvement at all.

We have focused on the Helman and Jájá's algorithm and as a main contribution we have shown how its standard implementation can be transformed to reduce the number of non-contiguous memory accesses, which dominate performance. We have also proposed another transformation that reduces the number of memory accesses that achieve higher speedups for ordered lists, but overall, since the random case is the most important one, reducing non-contiguous access has proven to be the most effective optimization.

We has also studied the impact of hyperthreading on this algorithm. Despite increasing the pressure for the shared resources, hyperthreading is able to further reduce the execution times by around 30% for large list sizes. For small sizes the benefits are much lower.

Finally, we have also suggested a dynamic parallel version of the Helman and Jájá's algorithm based on the work-stealing paradigm. This paradigm could improve the load balance between threads but unfortunately our preliminary results are still unsatisfactory. Nevertheless, as future research we plan to continue the study of this kind of implementations since we believe it fits well with the intrinsic characteristics of this highly irregular problem.

Bibliography

[1] Margaret Reid-Miller. List ranking and list scan on the cray c-90. In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 104–113, New York, NY, USA, 1994. ACM.

[2] David R. Helman and Joseph JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, pages 37–56, London, UK, 1999. Springer-Verlag.

[3] David A. Bader, Sukanya Sreshta, and Nina R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (smps) (extended abstract). In *Proceedings of the 9th International Conference on High Performance Computing - HiPC 2002*, pages 63–78, 2002.

[4] Isabelle Guérin Lassous and Jens Gustedt. Portable list ranking: an experimental study. 2002.

[5] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[6] David A. Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 547–556, Washington, DC, USA, 2005. IEEE Computer Society.

[7] David A. Bader, Virat Agarwal, and Kamesh Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–10, 2007.

[8] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Inf. Process. Lett.*, 33(5):269–273, 1990.

[9] J. C. Wyllie. The complexity of parallel computation. Technical Report TR 79-387, Dep. Comput. Sci., Cornell Univ., Ithaca, NY., 1979.

[10] David R. Helman and Joseph JJ. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265 – 278, 2001.

[11] Uzi Vishkin. Randomized speed-ups in parallel computation. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 230–239, New York, NY, USA, 1984. ACM.

[12] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. pages 81–90, 1988.

[13] Zheng Wei and Joseph JáJá. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *IPPS 2010: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 7–13, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989.

[15] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[16] M. Suhail Rehman, Kishore Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the gpu. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 235–243, New York, NY, USA, 2009. ACM.

[17] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[18] David R. Helman and Joseph JáJá. Prefix computations on symmetric multiprocessors. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 7–13, Washington, DC, USA, 1999. IEEE Computer Society.