HAL open science

PaSTeL : Parallel Runtime and Algorithms for Small Datasets

Brice Videau, Erik Saule, Jean-François Méhaut

To cite this version:

Brice Videau, Erik Saule, Jean-François Méhaut. PaSTeL : Parallel Runtime and Algorithms for Small Datasets. [Research Report] RR-6650, INRIA. 2008, pp.20. inria-00322158v2

HAL Id: inria-00322158

https://inria.hal.science/inria-00322158v2

Submitted on 17 Sep 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PaSTeL : Parallel Runtime and Algorithms for Small Datasets

Brice Videau — Erik Saule — Jean-François Méhaut

N° 6650

Septembre 2008

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6650--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

PaSTeL : Parallel Runtime and Algorithms for Small Datasets

Brice Videau, Erik Saule, Jean-François Méhaut

Thème NUM — Systèmes numériques
Équipe-Projet MESCAL

Rapport de recherche n° 6650 — Septembre 2008 — 20 pages

Abstract: In this document, we put forward PaSTeL, an engine dedicated to parallel algorithms. PaSTeL offers both a programming model, to build parallel algorithms and an execution model based on work-stealing. Special care has been taken on using optimized thread activation and synchronization mechanisms. In order to illustrate the use of PaSTeL a subset of the STL's algorithms was implemented, which were also used on performance experiments. PaSTeL's performance is evaluated on a laptop computer using two cores, but also on a 16 cores platform. PaSTeL shows better performance than other implementations of the STL, especially on small datasets.

Key-words: multi-core architecture, small datasets, STL, performance study, programming

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

PaSTeL : environement de programmation et algorithmes parallèles pour petits jeux de données

Résumé : Ce document décrit PaSTeL, un moteur d'exécution dédié aux algorithmes parallèles. PaSTeL offre à la fois un modèle de programmation pour écrire des algorithmes parallèles et un modèle d'exécution basé sur le vol de travail. Une attention particulière a été portée aux mécanismes d'activation et de synchronisation. Dans le but d'illustrer l'utilisation de PaSTeL un sous ensemble des algorithmes de la STL a été implanté, ces algorithmes ont également servi à des études de performances. Les performances de PaSTeL sont évaluées sur un ordinateur portable possédant 2 cœurs, mais aussi sur une plate-forme à 16 cœurs. PaSTeL montre de meilleures performances que les autres implantations de la STL, particulièrement sur les petits jeux de données.

Mots-clés : architectures multi-cœurs, petits jeux de données, STL, étude de performances, programmation

1 Introduction

When increasing frequency of processors, thermal output raises much more than performance. In order to solve this problem, processor manufacturers begin to increase the number of cores while keeping the frequency at a reasonable level. Thus, desktop and laptop computers rapidly become multi-processor platforms. There is a competition between major manufacturers (AMD, Intel) where each one tries to be the first to release quad-core processors. Intel even went as far as demonstrating the Teraflops Research Chip using 80 cores, with a peak performance of a teraflop consuming 62 Watts. Nevertheless, three problems may temper the enthusiasm for multi-core processing. First, the end user spends most of his/her time reading or writing emails or surfing the Internet. Thus, he/she feels relatively unconcerned because he/she spends most of the time using only one core. Second, application programmers, beginners and even experienced, seldom have the knowledge and time to design parallel algorithms to take advantage of the multiple cores of a processor. Last, even programs designed with parallelism in mind can be challenged by scaling problems. Enabling an efficient and easy to use parallel programming platform is a major challenge.

Since a decade, object oriented programming and design have encountered great success. Several programming languages, like C++ and Java, are widely used in the industry and the academic world. Moreover, these languages are taught to most of the computer science students. These languages depend on standard libraries that aim at easing and reducing the development phase. A well known example is the C++ Standard Template Library or STL which defines a set of classical data structures in a generic manner. It also proposes several algorithms using these structures. Programmers can therefore focus on the application, taking advantage of well-known structures and algorithms, instead of reinventing the wheel. However, these algorithms can be time consuming, for example when a large set of complex elements needs to be sorted. In such cases, programmers and users would like to have algorithms that can be executed efficiently on several cores. These algorithms should be efficient on large datasets. However, frequently, algorithms are used on small datasets. Thus, the implementation should be efficient also on small datasets. This is especially important as the number of cores continue to increase, the ratio between the work available and the number of core will decrease.

When using multi-core platforms, two objectives can be targeted, that at first seem antagonistic: the time to complete a task and the electrical power consumed. Nevertheless, if one knows how to use efficiently multiple cores these two objectives are not so different. Let us suppose a core running at frequency F can produce W instructions per second using P Watts. The same core running at frequency $F/2$ can produce $W/2$ instructions per seconds using hypothetically $\sqrt{P}$ Watts. If one can aggregate efficiently the work of two such cores, one can execute W instructions per second using only $2\sqrt{P}$ Watts. In the case of a desktop computer, which is mainly doing small tasks from times to times, the smaller the task that can be parallelized, the better the power saved while maintaining good performances.

We thus designed PaSTeL in order to study the size of the tasks that could be parallelized on modern multi-core architectures. PaSTeL is a parallel library that offers an API to develop parallel algorithms and a runtime to execute these

algorithms. The execution model is based on work stealing in order to guarantee a good load balancing between processors and cores. We kept PaSTeL simple and easily tunable, this way many parameters can be modified, both at runtime and on the compilation. In order to evaluate the performances of PaSTeL and the impact of the parameters on the performance, a subset of the STL is implemented. By using efficient synchronizations and activations of threads, we show that PaSTeL presents very good performance even on small datasets. The performance of PaSTeL has been compared to the one obtained with the Multi-Core STL or MCSTL [1] library developed at the Karlsruhe University and with algorithms programmed with Intel Threading Building Blocks or TBB [2].

The report is organized as follows: section 2 summarizes previous works on STL parallelization. We also describe the work-stealing approaches found in several environments. Main design and implementation choices of PaSTeL are presented in Section 3. Section 4 depicts the main experiments we conducted, first on a laptop computer equiped with a dual-core processor, and after on an octo-processor platform using 16 cores. The last section presents concluding remarks.

2 STL Parallelization Approaches

Parallelization of STL algorithms is a topic addressed by several authors. Section 2.1 portrays the libraries which add a parallel semantic to the STL. Work-stealing approaches are not new either, many previous works study this problem. Section 2.2 presents available work-stealing engines. Lastly, section 2.3 concludes on the different works presented.

2.1 Parallelized STL Libraries

We describe two parallel implementations of the STL.

STAPL [3] is a library that redefines STL concepts (containers, iterators and algorithms) for parallel architectures, with either shared or distributed memory. Despite its adaptive algorithmic, STAPL's performances are poor on small datasets [3].

MCSTL [1] is a parallel implementation of the STL dedicated to multi-core architectures. It relies on OpenMP to parallelize STL's algorithms. Embarrassingly parallel algorithms use a work-stealing engine while others are implemented with *ad hoc* parallel algorithms. MCSTL is the only STL we know to be specifically aimed at shared memory multi-core platforms. That is why we chose MCSTL as a reference in our experimental study. OpenMP reliance is both an asset because the code is highly portable and a drawback as the implementation of OpenMP might not be tuned to meet MCSTL's needs.

Moreover, some works aim at obtaining speedup on small datasets. For instance, [4] studies the use of vectorial instructions in modern processors in order to sort more efficiently datasets of about a hundred elements.

2.2 Work-Stealing Engines

Cilk [5] is an extension of the C language allowing parallel programming of shared memory platforms. Efficiency of parallel computing is guaranteed by

work-stealing techniques that rely on the work-first principle. This principle claims that the overhead due to the parallelization must only appear when parallelization is effective. Cilk aims at parallelizing a whole application, its engine is thus very generic. In fact, each Cilk process updates a double ended queue of the work it has to do. The process adds and consumes tasks on one end of the queue while other processes (known as thieves) may consume tasks by the other end. This way the application parallelism is explicitly showed to other running processes. Cilk programming model is strongly tied to recursion. Indeed, in Cilk, functions can be executed asynchronously.

KAAPI/ATHAPASCAN [6] is an execution engine for adaptive programming using work-stealing. It shows many common traits with Cilk. The main differences are the programming language chosen and the targeted platforms. KAAPI is written in C++ and this way offers high level interfaces. KAAPI is efficient on distributed memory architectures like computing grids and clusters.

Recently, Intel offered a set of tools called Threading Building Blocks or TBB [2] in order to help exploit multi-core platforms. Its architecture is based on Cilk. TBB is claimed to be very efficient. That is why we implemented some algorithms using this engine to compare against PaSTeL. TBB is very easy to program. Indeed, it is really well designed.

2.3 Conclusion on Related Works

PaSTeL is the only library that aims at obtaining high performance at all scales. No other parallel implementation of the STL aims at obtaining interesting performance on small datasets.

Explicitly subdividing work can not be considered negligible when dealing with small volumes of data. This partition is necessary in order to parallelize an application as a whole, making it a relevant choice for work-stealing engines like Cilk, KAAPI or TBB. Nevertheless, when writing a toolbox of highly responsive parallel algorithms like PaSTeL, this partition implies an important overhead.

3 PaSTeL

In this section we present PaSTeL in details. It can be downloaded at [7]. We first present the programming model and the execution model of algorithms. Then, we explain how this model is implemented in PaSTeL. Finally, the tuning of PaSTeL parameters is presented.

3.1 Programming Model

When describing the model, the computational unit is called core and the function call to be parallelized is called algorithm.

Implementing an algorithm in PaSTeL requires to write some data structures and functions, synthetically presented in Figure 1.

First, one needs to establish a global data structure that can be seen by all cores and a local data structure that only concerns a core. The global data structure is instantiated once per algorithm and should be used to synchronize cores. While the local data structure is owned by each core in the algorithm and should be used to manage the execution of the related core. For instance,

```
struct GlobalData {...}
/*Contains algorithm's data*/
struct LocalData {...}
/*Contains per-core data*/
bool GlobalComputations (GlobalData g){...}
/*Returns false if the algorithm is finished*/
bool LocalComputations (LocalData l){...}
/*Returns false if core computations is finished*/
void ComputeChunk (LocalData l){...}
/*Executes one local chunk*/
void CommitLocalToGlobal
  (LocalData l, GlobalData g){...}
/*Declares l state to g*/
void Steal (LocalData l, LocalData v){...}
/*Transfers work from v to l*/
```

Figure 1: PaSTeL API

the local data of a core could be used to store which part of the computation should be done by this core.

Secondly, some functions must be specified to match PaSTeL's API. Computations are implicitly partitioned into chunks where the chunk's size is user-defined. The ComputeChunk function takes the core's local data as an input and computes one chunk. A core knows that it owns some work by using LocalComputations which takes the core's local data in parameter. Eventually, the local data of a core and the global data need to be synchronized which is done by CommitLocalToGlobal. The end of the algorithm is detected by GlobalComputations which takes the global data as an input. The Steal function gets some work from another core and takes the thief core's local data and the victim core's local data in parameter. Finally, some functions to allocate and deallocate data structures previously presented must also be given.

To illustrate how to write an algorithm in PaSTeL, we present the implementation of the *two way merge* algorithm in the PaSTeL programming model. This algorithm merges two sorted arrays into a larger sorted array.

Each core has an interval of both arrays to merge referenced in its local data. The function ComputeChunk merges a fixed number of elements. It keeps a track of how many elements have been merged and updates intervals that need to be merged. The LocalComputations function decides that there is still some work to execute by checking that both intervals are not empty. When the core has no more work, it reports how much work it did: CommitLocalToGlobal decreases the number of elements that still need to be merged by the number of elements it has merged since the last synchronization. GlobalComputations returns that the algorithm is finished when all elements of both arrays have been merged.

The steal operation, which is done by Steal, splits both arrays of the stolen into two. The thief will merge the last part of both arrays whereas the stolen will merge the first part. For the resulting array to be sorted, one needs to select

```
Function workstealing(GlobalData g, LocalData l)
while GlobalComputations (g) do
  if LocalComputations (l) then
    while LocalComputations (l) do
      ComputeChunk (l)
      if Steal target then
        CommitLocalToGlobal (g,l)
        Wait until thieves are gone
      end if
    end while
    CommitLocalToGlobal (g,l)
  else
    t = select_a_victim()
    Steal (l,t)
  end if
end while
```

Figure 2: Execution Model

the splitting points so that all elements of the first parts of both arrays are lesser than all elements of the last parts. This is done by cutting the largest array in the middle and looking forward the appropriate cutting point in the other array using a bisection search. It is easy to compute where both cores should merge their intervals since the number of elements to merge by each core and their ordering is known. Remark that the steal operation is done in logarithmic time.

Using this programming model, three kind of algorithms are currently implemented in PaSTeL: *min_element*, *merge* and *stable_sort*. The implementation works with any data container that implements random access (in the STL terminology, data container should provide *RandomAccessIterator*. The semantic is equivalent to the array semantic).

3.2 Execution Model

This section presents how the functions of the PaSTeL's API are called by the engine.

Before executing the main procedure of PaSTeL, the calling function creates and initializes global data, algorithm management data, core management data and local data. Global and local data are user defined. PaSTeL also uses some runtime data: Algorithm management data includes the state of the algorithm and some locks, and core management data contains the thread state and some locks. Other working threads initialize their local data and their personal management data before executing the main procedure of PaSTeL.

Figure 2 presents the main procedure of PaSTeL, the workstealing function. It is executed by all the threads contributing to the computation. The algorithm goes to the termination phase when the GlobalComputations function returns that there is no more work. Otherwise, the algorithm has two streams.

A thread executes the first stream when it owns some work. It begins by changing its state to declare that it is a working thread. It executes its work chunk by chunk; each one is processed by ComputeChunk. After each chunk, the thread knows if another thread is trying to steal some work from itself by checking its own state. If the thread is victim of a steal operation, it updates the global state with CommitLocalToGlobal and waits until no more thread tries to steal it. Eventually, the thread finishes the work it owns and then it updates the global state before re-executing the whole algorithm. About thread synchronization, updating the global state is protected using a lock and modifying the thread states is protected by a lock per thread.

Eventually, a thread has no longer local work and executes the second stream. In this case, the thread updates its state to notify that it is a thief. Then, it chooses randomly a working thread and notifies the victim thread that it is going to be stolen by changing the state of the victim thread. The thief thread uses the Steal function to retrieve some computations and then, it changes its state to notify that it is a working thread. All updates of thread's state need to be protected using a lock.

The termination phase consists only in counting the number of threads that finished the algorithm.

Remark that in several algorithms the GlobalComputations function is atomic. If not, remark that the global state is only modified in CommitLocalToGlobal. Two calls to GlobalComputations will return the same result unless a thread called CommitLocalToGlobal. Thus, one can add some code in CommitLocalToGlobal to pre-compute the result of GlobalComputations, making GlobalComputations an atomic operation that does not require to be protected by a lock.

Moreover, while choosing a thread to steal, one does not need to lock threads' data since all reading of a thread state are atomic. When the stolen will be chosen, it is useful to confirm that its state has not changed. This will be done after the lock has been taken.

The work-stealing mechanism of PaSTeL is synchronized, the victim thread stays in a waiting state until no more thread tries to steal it. Other work-stealing mechanisms use an asynchronized mechanism to ensure a fast execution of computations in the case where no steal operation appears, but requires to manage two sets of work explicitly. However, the overhead induced by checking a thread state is very small. The only significant overhead is due to idle times induced by synchronization when a stealing operation occurs. Moreover, any algorithm based on work stealing should minimize the number of steal operations, and thus, reduces the overhead induced by using synchronized steal operations.

3.3 Implementation Details

The execution model of PaSTeL is currently implemented using POSIX threads. To be able to get some speedup, one needs a kernel that maps POSIX threads to kernel threads. To reduce thread spawning overhead, number of cores used minus 1 computation threads are created during the initialization of PaSTeL. Each of these threads are bound to a single core.

Before executing the main procedure of PaSTeL, the calling function registers the data into a queue. Computation threads pop an algorithm from this

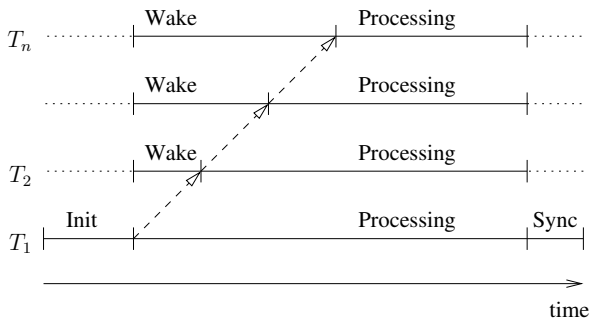


Figure 3: PaSTeL threading model.

queue and check in the management data if the algorithm is finished or not. If the algorithm is finished, it is discarded. If the algorithm is not finished, the computing thread keeps a pointer to global data, pushes the algorithm back to the queue, then it executes the work-stealing procedure. The access to the queue is protected using a lock and the access to management data is protected by a lock per algorithm.

Locks in PaSTeL are implemented using spinlocks since mutexes were found to be less efficient. However, using spinlocks results in a waste of CPU resources. Thus, a mutex based alternative is available.

3.4 Performance Prediction and Thread Allocation

Let us recall that the goal of PaSTeL is not to provide another highly efficient, high performance parallel engine but to provide tools to understand how modern parallel systems behave in order to tune engines at runtime. This section provides a first model of PaSTeL execution to tune PaSTeL's parameters.

The current implementation of PaSTeL contains two major parameters. The first one is the size of the chunks to be computed. The second one is the number of threads that PaSTeL should allocate to an algorithm. Computing efficient values of those parameters requires a model predicting PaSTeL's performances.

The behavior of threads, such as their launch and termination times, is the first point to model. The execution of a single task in PaSTeL can be modeled rather simply. Figure 3 represents a fairly accurate model of an execution of a PaSTeL algorithm with n threads. First, the main thread T_1 initializes PaSTeL. This step takes I time units. Then, T_1 wakes all other threads. The wake-up mechanism follows a cascading model with equal latency W between each wake-up. Thread i takes $W_i = (i-1)W$ time units to wake-up. When it is waken-up, thread T_i works during p_i time units. Finally, the main thread will perform a synchronization operation for S time units.

Let us denote by c the total time to complete the task and by P the time the task would have taken on a single core (*i.e.* the sequential time). Let us assume that during the working phase, the parallelism is perfectly efficient: $P = \sum_{i=1}^{n} p_i$. This assumption neglects the cost of steals and the cost of the

stealing mechanism. Let us also assume that I, W and S are constant and do not depend on the number of threads.

The final synchronization phase can not start before all the threads are waken: the total time to complete is $c = I + \max_i(W_i + p_i) + S$. Since the parallelism is perfect, the nth thread is useful as long as it has enough time to do some work, *i.e.*, as long as $p_n > 0$. Thus, c is minimized by maximizing n such that $p_n > 0$.

First, determining p_1 in function of the number of threads is a key point. Summing the time spent by each thread while T_1 works leads to:

$$\begin{aligned} np_1 &= \sum_i p_i + \sum_{i=1}^{n} W_i \\ &= P + \frac{n(n-1)}{2} W \\ p_1 &= \frac{P}{n} + \frac{n-1}{2} W \end{aligned}$$

The time each core spends on the work is given by:

$$\begin{aligned} p_i &= p_1 - (i-1)W \\ p_i &= \frac{P}{n} + \frac{n-2i+1}{2} W \end{aligned}$$

Recall that p_n is positive. Thus,

$$\begin{aligned} p_n = \frac{P}{n} - \frac{n-1}{2} W &> 0 \\ n^2 - n - \frac{2P}{W} &< 0 \end{aligned}$$

Solving this equation in the real set leads to the following bound:

$$n < \frac{1}{2} + \frac{1}{2}\sqrt{1 + 8\frac{P}{W}}$$

It is now possible to compute the optimal number of threads that PaSTeL should use and the total execution time of a call to PaSTeL. Both values are now given:

$$n = \left\lceil \frac{1}{2}\sqrt{1 + 8\frac{P}{W}} - \frac{1}{2} \right\rceil \tag{1}$$

$$c = I + \frac{P}{n} + \frac{n-1}{2} W + S \tag{2}$$

This model is currently implemented in PaSTeL in order to predict the performances at runtime. Parameters such as I, W, S are determined when PaSTeL starts, and are constant if the core's speed does not vary. The sequential time P depends can only be known when a PaSTeL call occurs. P is estimated by

measuring the processing time of a given number of unitary operations of the algorithm using the cycle counter of the cores. If the first measure is too small to give an accurate result, it is reproduced on a larger number of elements. Then knowing the complexity of each algorithm, we can predict the sequential running time P.

Since all parameters are known, it is possible to determine if using the parallel algorithm (*i.e.* PaSTeL) leads to interesting speed-up or not. c can be easily lower bounded by: $c > I + W + S$. So if $P \leq I + W + S$, then the sequential algorithm is used. Otherwise PaSTeL computes the optimal number of threads using Equation 1 and launch the parallel algorithm using n threads. Equation 2 is not directly needed so it is not used at this time.

For now this mechanism is only implemented in the *minimum* algorithm, but it can be easily extended to others. The overhead of this mechanism is low (a few hundred cycles), and it is showing good results as will be seen in the next section.

Nevertheless, it will have to be extended if the processing time is irregular. It should be possible to use a similar technique if the distribution is known to the user and communicated to PaSTeL or if the distribution can be obtain at runtime.

Finally, the last parameter to tune is the chunks' size. Using large chunks will reduce the overhead of the PaSTeL engine while using short chunks will increase the reactivity of the stealing mechanism and provides a better load balancing. This parameter is not tuned automatically at the moment. However, it should be easy to compute a reasonable value by fixing the overhead of the PaSTeL engine to a given percentage.

Parameter evaluation has already been used successfully. For instance, Atlas [8] is a high performance linear algebra library that benchmarks the computing platform during the installation of the library.

4 Performance Study

In this section we study the performance of PaSTeL. After presenting the experimental platforms used in section 4.1, we will compare the performances of PaSTeL, MCSTL and TBB on a laptop computer in section 4.2 and on a computing server in section 4.3. The efficiency of the performance prediction mechanism is investigated in Section 4.4.

4.1 Experimental Platforms and Methodology

The first platform used in this study is *laptop*. It is a laptop computer using an Intel Core2 Duo T7100 dual core processor clocked at 1.8 GHz with 2GB of RAM. Memory bus is clocked at 667 MHz. The second platform is named *octo*. It is a server powered by 8 AMD Opteron 875 dual core processors clocked at 2.2 GHz. Each processor is associated with 4 GB of RAM. Thus this machine is composed of 16 cores and 32 GB of RAM. Nonetheless, only 4 processors have been used, as tests proved none of the tested library scaled well above 8 cores, because of the lack of memory bandwidth.

On *laptop* the kernel is a 64bit Linux 2.6.22 and the compiler is g++ 4.2.1. On *octo* the kernel is a 64bit Linux 2.6.23 and the compiler is g++ 4.2.3. On both platforms, codes were compiled with *-O2* and *-DNDEBUG* to disable assertions.

In order to compare PaSTeL against TBB, we implemented the same algorithms in TBB and PaSTeL. These algorithms were not readily available in TBB and were implemented with TBB20_20080319. They were previously presented in section 3.1, but in TBB we used the *auto partitioner*. In MCSTL 0.8, the algorithms were of course available and so were used directly.

4.2 Global Performances on Core2 Duo

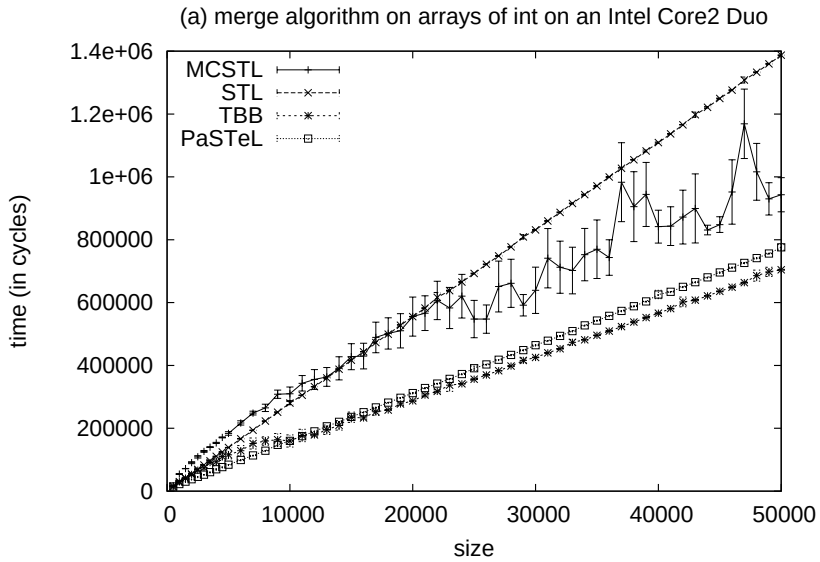
The first experiment aims at comparing the performances of TBB, MCSTL and PaSTeL on a dual core environment. The result of the gcc-provided STL implementation is given as a reference. The three algorithms previously presented have been executed on *laptop*, using random instances of size varying between 50 and 50000. Instances are arrays of *int*. Experiments using other data types have been conducted. However, results are similar and thus are not presented here. PaSTeL algorithms also have a chunk's size parameter that is set manually to the best possible value (50 iterations on *min_element*, 100 for *merge* and 400 for *stable_sort*). Each measurement is repeated 20 times, and each measurement is the mean time of 20 runs on the same instance in order to be placed in continuous running.

Figure 4 and 5 presents mean results and 95% confidence intervals of the execution time (in cycles) of each algorithm for different data sizes (in number of elements). This figure also presents results for very small data sizes, comprised between 0 and 3000 elements, for the *min_element* algorithm.

The first thing to notice is that on *min_element* and *merge* algorithms MCSTL is at a disadvantage compared to PaSTeL and TBB (especially on relatively small arrays like those presented here). It can also be seen, especially on the *min_element* result that MCSTL fixed overhead is much more important than the one of its counterparts. One could argue that the OpenMP implementation in gcc is based on mutexes. However, experiments using the mutex alternative of PaSTeL can show that mutexes are not responsible for the bad performance of MCSTL. For the sake of equity, it must be mentioned that MCSTL stable sort algorithm is more effective than the one we implemented in PaSTeL and TBB, and gives the best results on larger arrays.

TBB is showing better performances than MCSTL, and is comparable with PaSTeL. Indeed, it is slightly better than PaSTeL on the *merge* algorithm, and slightly less effective on the *stable_sort* algorithm. The *min_element* case is special because its completion time is very short (less than 200000 cycles) on the largest array we considered. TBB *auto partitioner* is not working very efficiently at those sizes, whereas on larger arrays TBB and pastel *min_element* performance are very much alike.

Our objective is to demonstrate that, using multi-core architectures, it is possible to parallelize algorithms and still be efficient on *small* datasets. On *laptop*, PaSTeL is showing speedups on arrays of less than 3000 *int* (or 15000 cycles) on the *min_element* algorithm, of less than two time 800 *int* (or 20000 cycles) on *merge* and less than 900 *int* (or 120000 cycles) on *stable_sort*. Defining *small* is difficult, but nonetheless, PaSTeL is showing speedups on smaller arrays than MCSTL or TBB and can be considered reactive.



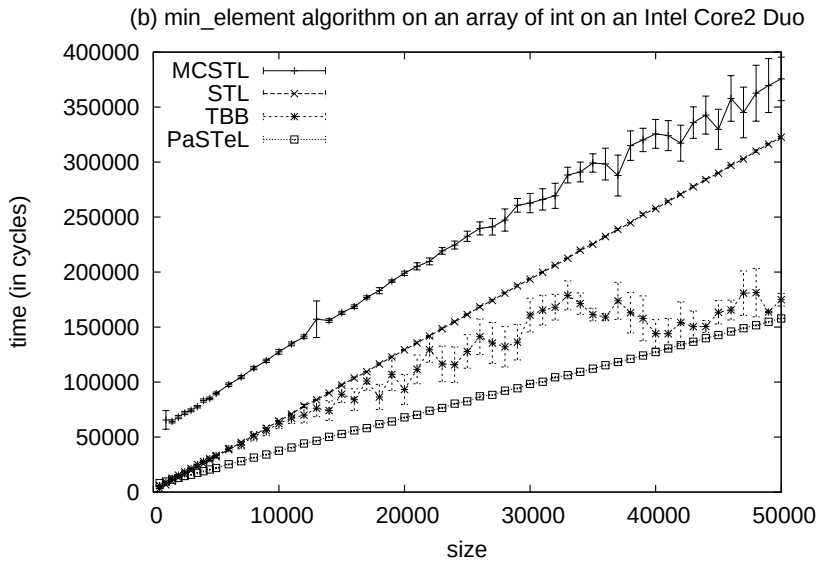
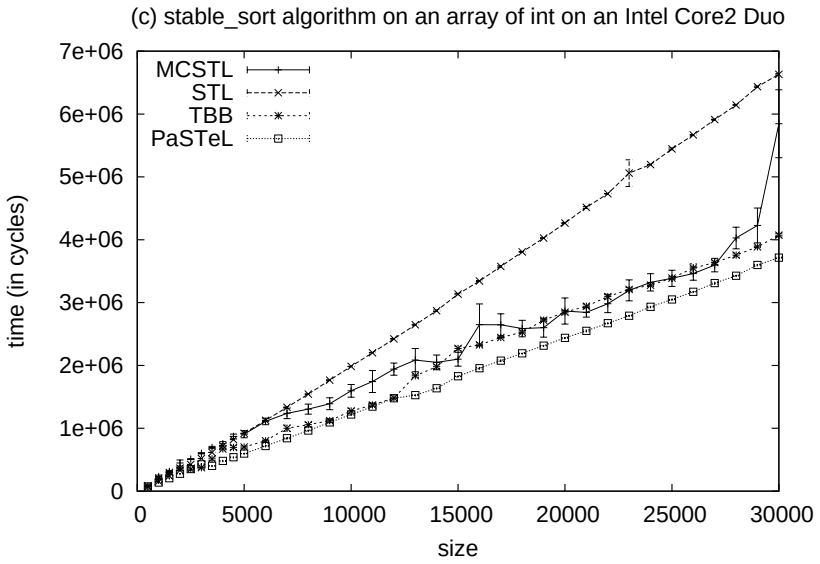


Figure 4: Compared results of PaSTeL, MCSTL, TBB and STL on Core2 Duo (2 cores)



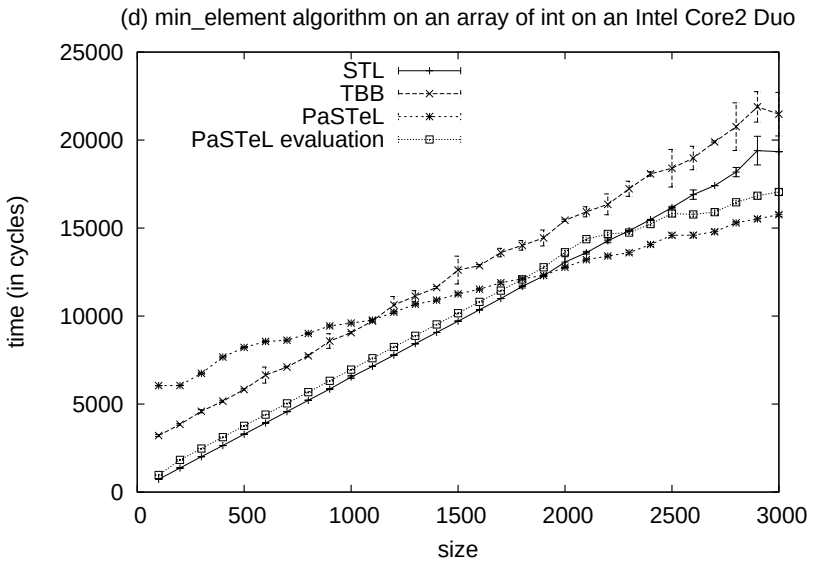


Figure 5: Compared results of PaSTeL, MCSTL, TBB and STL on Core2 Duo (2 cores)

4.3 Global Performances on Opteron 875

Figure 6 and 7 presents the same experiments on the *octo* computer, using 8 cores and thus 8 threads for parallel algorithms. The objective is to check on the behavior of pastel on a hierarchical architecture using several processors. For all algorithms, the chunks' size is fixed to 400 elements.

Results using TBB are presented without confidence intervals (they were too large) because TBB on this platform is showing performances' problems on some executions, and deadlocks. Different releases of TBB were tried, but problems still appeared. A bug report has been filed. Nonetheless when no problem appears, what was said in the previous subsection can be transposed in this one.

Results are similar to those obtained on Core2: PaSTeL keeps obtaining lower execution time than MCSTL or TBB. PaSTeL's engine seems to scale correctly with the number of processors for all tested algorithms. However, the *stable_sort* algorithm on arrays of more than one million *int* elements (not shown in figures) presents some differences. The presence of synchronization barrier in the PaSTeL implementation of the stable sort algorithm seems to diminish significantly performance when the number of cores becomes large. At this point, MCSTL becomes a better choice.

PaSTeL shows performances better than the sequential code for arrays of more than 18000 *int* on the *min_element* algorithm on Opteron 875. Recall that on the Core2 architecture, the parallel implementation was better from 2500 *int*. The Core2 architecture is thus around 7 times more reactive than the Opteron architecture. However, there are more processors in the *octo* machine leading to better computation time when the total processing times increases. For instance, merging arrays of 50000 *int* on *octo* took around 500000 cycles whereas *laptop* merged them in around 800000 cycles.

4.4 Performance prediction

In this section, we investigate the behavior of the mechanism that predicts the number of thread presented in Section 3.4. Figures **??**(d) and **??**(d) show a curve 'PaSTeL evaluation' which is obtained with the prediction mechanism on the *min_element* algorithm.

Without the prediction mechanism, PaSTeL shows an overhead on smaller arrays because too many workers are assigned a task too small. On *laptop* (Figure **??**(d)), the overhead induced by PaSTeL (without prediction) is around 5000 cycles which is prohibitive since the sequential algorithm runs in 3000 cycles. The PaSTeL engine with prediction is able to determine that the sequential code should be used for arrays of size less than 2000 *int* whereas it detects that the parallel implementation should be used after. In the current implementation, the prediction overhead is around 100 cycles if the sequential code is used and around 1200 is the parallel code is used. Indeed, when the parallel code is used, the mechanism has to compute the number of threads to use which implies more computations. This phase could have been removed on *laptop* since there are only 2 cores: a parallel code must use 2 threads.

On *octo* (Figure **??**(d)), running the PaSTeL without prediction with 8 threads leads to an important overhead for arrays of less than 35000 *int* (around 50000 cycles for arrays of 10000 *int*). However, the prediction mechanism almost

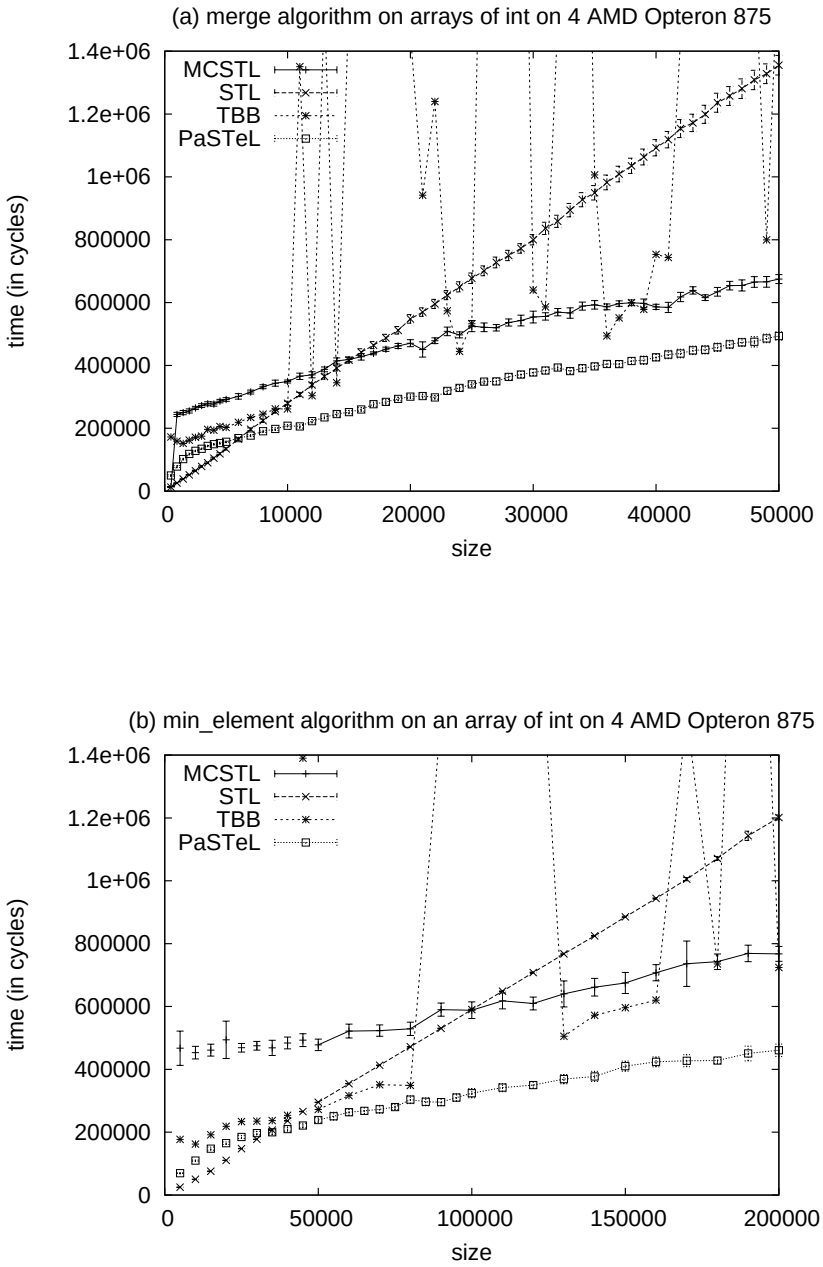
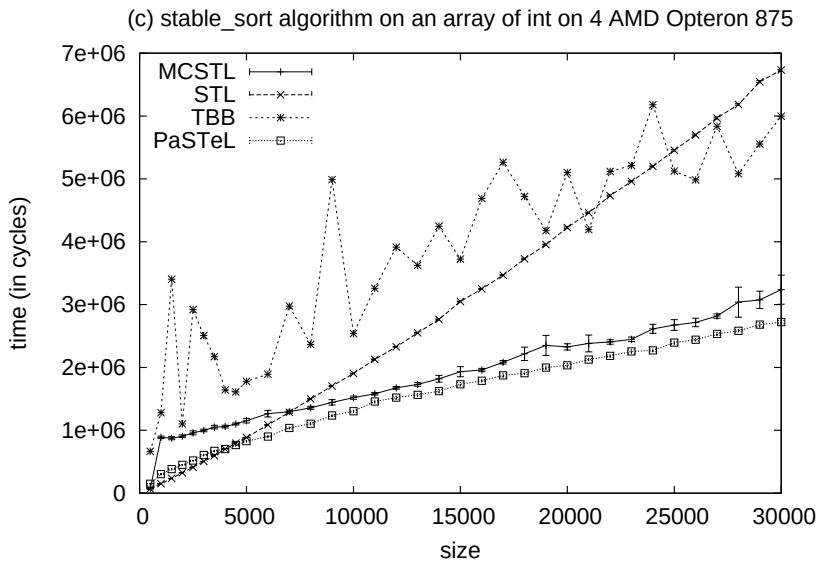


Figure 6: Compared results of PaSTeL, MCSTL, TBB and STL on Opteron 875 (8 cores)



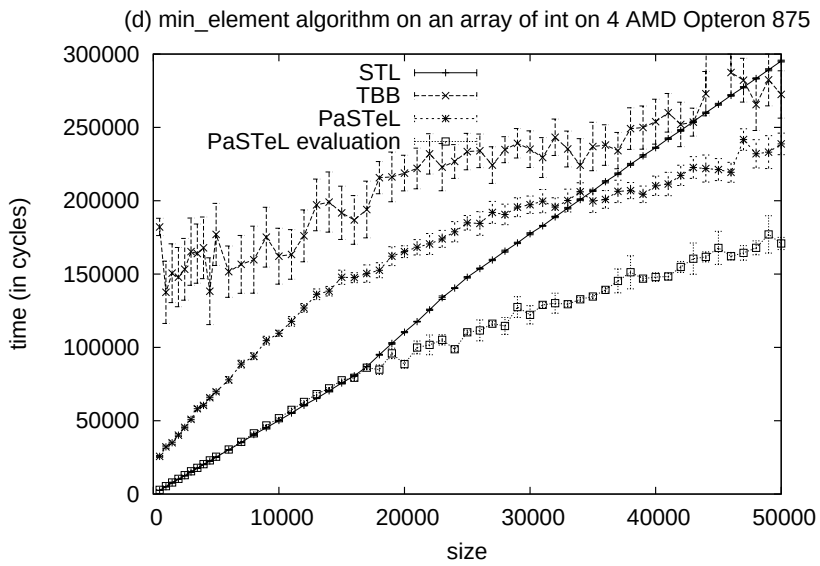


Figure 7: Compared results of PaSTeL, MCSTL, TBB and STL on Opteron 875 (8 cores)

completely suppresses the overhead for arrays of less than 20000 *int* (around 300 cycles of overhead). For more than 20000 *int*, the prediction mechanism is able to select a number of threads that leads to better performances than the sequential algorithm and PaSTeL with 8 cores. Thus, the mechanism is effectively able to select a number of threads that leads to interesting computation time.

It is now clear that the prediction mechanism is able to select an interesting number of threads. However, one can wonder the accuracy of the mechanism. Three kind of executions are considered: *Prediction* is the computation time of the PaSTeL algorithm using the prediction mechanism to select a number of threads. *Best* is the computation time of PaSTeL without prediction leading to the lowest computation time (including sequential) that is to say, the lowest computation time achieved by a run of pastel using a fixed number of thread (at compile time) have been selected manually from all possible number of threads. *Worst* is the worst run time achieved by a fixed number of threads with PaSTeL.

Figure 8 presents the performances of the prediction mechanism relative to the best achievable performances on *octo* for different sizes of the array. Results are shown as a degradation of not being the best run. Three quantities are presented in Figure 8 which are the ratio between *Best* and *Best*, *Prediction* and *Worst*. Obviously, *Best* achieves a ratio of 1 to itself. For arrays of 50000 elements, the figure shows that *Worst* achieved 55% of the performance of the optimal number of threads. In other words, *Worst* is around 2 times longer than *Best*.

The values of *Best/Prediction* are always greater than 0.8 meaning that the prediction mechanism waste less than 20% of its computation time. Most of the time, *Best/Prediction* is above 0.9: less than 10% of the computation time was wasted. One could argue that losing 20% of the computation time is a lot, but one should recall that the comparison between *Best* and *Worst* is unfair since *Best* has been manually tuned. Moreover, the prediction is quite accurate since a wrong number of threads could have lead to performances close to *Worst*, and *Prediction* is far better than *Worst*.

Let us now consider the variations of the curve with the array's size. The ratio of *Prediction* increases from arrays of 1 elements to approximatively 15000. In fact, the prediction engine does not start the parallel implementation which is the best choice. Since the prediction phase has a constant cost, the efficiency increases with the array's size. After 15000, the non-monotony of the ratio states that the prediction engine makes some mistakes. About the ratio of *Worst*, the number of threads of the worst execution decreases from 8 to sequential (after 35000).

5 Conclusion

In this report we presented PaSTeL, a tool to study parallel computing at all scales in multi-core architecture. Some algorithms of the STL have been implemented with it. It allows to take advantage of the parallelism offered by multi-core processors at almost no cost in development. PaSTeL is along the same line as other works undertaken to parallelize algorithms of the STL. It is built upon a simple model of programming and an execution engine based on work-stealing. PaSTeL distinguishes itself from its counterparts by using

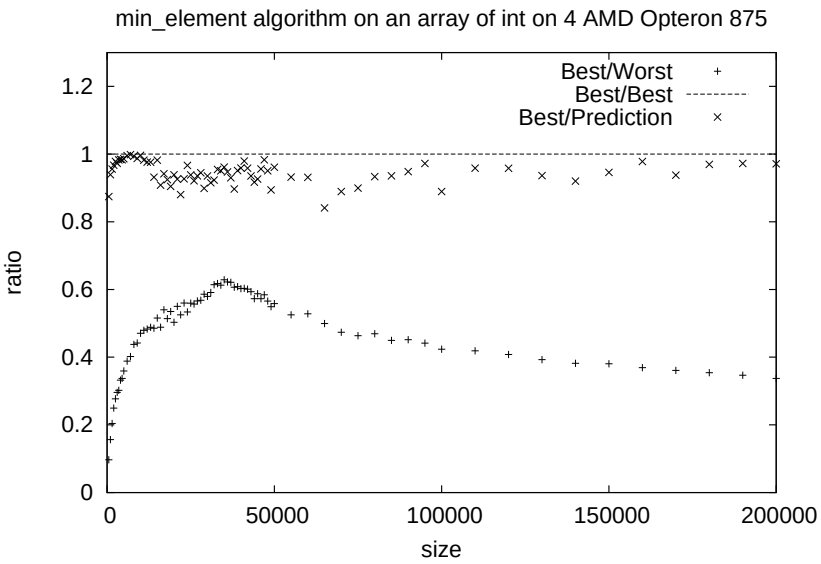


Figure 8: Relative performance of the prediction mechanism

threads reactive to synchronization events. Experiments that have been conducted shows that PaSTeL is efficient especially on short parallel executions.

This first work shows that predicting threads' behavior in a parallel computing engine is possible and can be done quite accurately. Some tracks still have to be followed. The prediction mechanism should be extended to more complex algorithms such as *merge* and *stable_sort* and automatically tuning the chunks' size should be studied. PaSTeL is a tool to study parallel engine at runtime: understanding the difference between the estimated number of threads and the optimal number of thread is a key issue. In particular, does the error come from bias in time measurement? Or does the model need to be refined?

References

[1] J. Singler, P. Sanders, and F. Putze, "MCSTL: The Multi-Core Standard Template Library," in *Euro-Par*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641. Springer, 2007, pp. 682–694, http://algo2.iti.uni-karlsruhe.de/singler/mcstl/.

[2] A. Kukanov and M. Voss, "The foundations for scalable multi-core software in intel threading building blocks," *Intel Technology Journal*, vol. 11, no. 4, Nov. 2007.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: An Adaptive, Generic Parallel

C++ Library," in *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, Aug. 2001, pp. 193–208.

[4] T. Furtak, J. N. Amaral, and R. Niewiadomski, "Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms," in *Proceedings of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, 2007, pp. 348–357.

[5] J. Singler, P. Sanders, and F. Putze, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 212–223, vol. 33, No. 5.

[6] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors," in *Parallel Symbolic Computation'07 (PASCO'07)*, no. 15–23. London, Ontario, Canada: ACM, 2007.

[7] "pastel website," http://gforge.inria.fr/projects/pastel/.

[8] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, cD-ROM Proceedings.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399