On-line scheduling

J. Briat – T. Gautier – J. L. Roch
[Jacques.Briat,Thierry.Gautier,Jean-Louis.Roch]@imag.fr
APACHE Project - CNRS–INPG–UJF–INRIA
LMC-IMAG
Grenoble, France

Abstract

For many applications, parallelism is generated during execution. Dynamic scheduling is then critical to guarantee an efficient execution. We consider the problem of scheduling an application that generates dynamically tasks of unknown duration with precedence relations. We focus on the overhead related to scheduling such a program on a PRAM, considering two aspects: the regularity *of the application which measures the minimal work to schedule it and the* competitive ratio *of the scheduling algorithm which measures the quality of the performed schedule.*

1 Introduction

A parallel dynamic application is characterized by a behaviour related to the input data. This means that neither the precedence graph of the sequential tasks of the application nor the duration of those tasks are known before execution completes.

Such an application needs a dynamic scheduling algorithm to be executed on a parallel machine. To ensure efficient executions, this scheduling algorithm should satisfy the following:

- the overhead due to the scheduling operations has to be neglectible compared with the whole parallel time: this overhead is characterized by the *regularity* of the application [17].
- the performed schedule has to be as close as possible to the optimal one. The *competitive ratio*, introduced in [27] in the context of on-line algorithms, allows a performance measure of the scheduling algorithm [26].

In this paper, we analyze those two criterions on the PRAM machine model. Despite the fact that computing an optimal schedule for a precedence graph with known duration tasks is NP-complete, it is possible to compute on-line a schedule that is within a constant ratio to the optimal one.

Many dynamic scheduling algorithms have been designed that can be classified among different criterions [7]. A difficult point concerning dynamic scheduling or load-balancing is to evaluate performances. Two complementary approaches may be considered. On the one hand, practical experimentations can be used to justify a scheduling algorithm for certain classes of applications on a given machine [30]. On the other hand, complexity analysis are possible on theoretical abstract models either in the static case [5, 11] or the dynamic one [26, 23, 13]. This paper is in this last context.

Organization of the paper. In part 2, we recall the basic definitions introduced in [21] that characterize *efficiency* and *scalability* of an algorithm. Those definitions do not take into account the overheads due to routing data and scheduling tasks that are assumed to be performed optimally. We point out the relationships between those two problems.
This correspondance motivates the definition of the scheduling problem as an extension of the routing problem intoduced in [29]. Paragraph 3 recalls the formalization proposed in [17] for the parallel execution problem.

The amount of work required to schedule an application is measured by its *scheduling complexity*, which can be roughly bounded by the number of tasks generated during the execution. Thus, the ratio of the whole number of operations performed to the scheduling complexity, which defines the *regularity* of the algorithm, is used in paragraph 3.2 to classify algorithms We illustrate this classification on two algorithms that implement a quick-sort in parallel.

In part 4, we recall basics tools to analyze performance of an on-line algorithm. Concerning on-line scheduling, paragraph 5 reviews the results of [26]. An important fact is that a greedy scheduling achieves an optimal competitive ratio among a large class of scheduling algorithms.

On a practical point of view, to obtain better bounds leads to consider restrictive classes of algorithms. Paragraph 5.4 exhibits a scheduling algorithm for a set of independent tasks with unkown durations that achieves a $(1+\epsilon)$ competitive ratio for any efficient scalable PRAM algorithm with polynomial regularity. This result is applied to the parallel quick-sort algorithms previously introduced.

To conclude, we introduce in part 6 the ATHAPASCAN parallel programming model developed by the APACHE project. It is a C++ library where parallelism can be expressed dynamically by building task graphs with precedence relations. Basic class hierarchy is presented.
The quick-sort algorithm studied in section 3 has been implemented on this environment. Its performances, obtained using two different scheduling mechanisms, are then presented.

2 From Parallel Algorithms to portable code

Writing a portable parallel program to solve a given problem needs the building of a parallel algorithm on a general machine model. Several machine models have been proposed in the litterature [12, 2, 29]. Among them, the PRAM model [15] and some of its variants is the most widely used to build parallel algorithms. Two approaches can be distinguished:

- The first consists in trying to emphasize on parallelism. The model is then both simple and very powerful to express complex parallel structures. The PRAM model is in keeping with this general pattern.
- The second approach consists in the definition of a restrictive parallel model, easy to emulate on any parallel practical machine with a bounded loss of efficiency. In this framework, the BSP model plays an important role [28].

Parallel complexity [9, 20, 19] aims at building on general theoretical models such as PRAM (*Parallel Random Access Machine* [15]) parallel algorithms both strongly parallel (thus *scalable* or *with polynomial speed-up*) and efficient (or having *constant inefficiency*) [21].

More precisely, let $\mathcal{P}_n$ be a problem and $T_s(n)$ be the time of the best sequential algorithm solving it. Let A be a parallel algorithm solving $\mathcal{P}_n$ in parallel time $T_{//}(n)$ using $P(n)$ processors.
A is said to have *polynomial speed-up* [20] (or to be *scalable*) if there exists $\epsilon < 1$ such that:

$$T_s(n^\epsilon) = O\left(T_{//}(n)\right).$$

Let $H(n)$ be the number of operations performed by A (without considering nop operations). $H(n)$ is upper bounded by $W(n) = T_{//}(n)P(n)$, the *work* of algorithm A.

To characterize the work overhead, the notion of inefficiency is introduced. *Inefficiency* of A is said:

- *constant* (A is said *efficient*) iff
$$H(n) = O(T_s(n)).$$
- *poly-logarithmic* iff
$$H(n) = O\left(T_s(n)\log^{O(1)}(T_s(n))\right).$$
- *polynomial* iff
$$H(n) = O\left(T_s(n)^{O(1)}\right).$$

Example The class EP is the set of problems that may be solved in parallel by a constant efficiency algorithm with polynomial speed-up. This class captures problems whose parallelization may lead to practical efficient execution. For instance, consider Gaussian elimination. In practice, although this problem belongs to NC, a parallel version of the algorithm that performs sequentially column eliminations leads to a parallel time n^2 using n processors. This algorithm, although not of polylogarithmic depth is in EP and of practical interest [11].

The main problem encountered in coding such a parallel algorithm, optimal on a theoretical point of view, on a given parallel architecture is to bound the overheads due to communication and scheduling. The communication overhead is due to the difficulty of efficiently simulating a PRAM on a general architecture. Scheduling overhead is related to, as we will see further, practical implementation of Brent's principle (see section 2.2).

2.1 Communications: locality

It is possible to characterize the communication overhead inherent to an algorithm using the notion of *locality* or *gross locality*, *i.e.* the ratio of the computational complexity by the communication complexity [24]. Non local algorithms require high performance communication capabilities to be efficiently implemented. As long as communication overheads are significant on the existing parallel computers or in other words, as long as the PRAM cannot be efficiently simulated, the exploitation of locality will be one of the main issues to address in order to achieve high performance.

The notion of locality may also been understood as a notion of *irregularity*. Indeed, problems that have *regular*, oblivious or predictable patterns of memory accesses, give rise more easily to programs organized so that they have low communication costs. It may be argued that the more local an algorithm is, the more regular it is. Consequently, a relevant criterion for the irregularity of an algorithm is the irregularity of the communication patterns that it involves [10]. The locality concept relies on the fact that models of parallel computation emphasize that the tasks of computation and of communication may be distinguished [28].

2.2 Scheduling: irregularity

Task management is the next important issue to be considered to execute efficiently parallel programs. For instance, optimal solutions for key problems such as *list ranking* may rely on task scheduling solutions [8].

More precisely, Brent's lemma [6] says that any synchronous parallel computation performing x operations in time t may be scheduled on p processors in time $\lceil x/p \rceil + t$. However, this principle does not take into account the overhead due to the computation of such a schedule.

The proof of Brent's lemma is based on indexing the operations performed in the parallel algorithm, according to their depth in the related precedence graph. Then, a cyclic – modulo p – allocation of the operations to processors leads to schedule the algorithm with no overhead.

However, in the case where the precedence graph corresponding to some input data can only be determined by a complete execution of the algorithm, indexing tasks cannot be performed without overhead.

In the following, we will consider two criterions to evaluate the scheduling overhead:

- the quality of the schedule: on a given parallel algorithm, this quality may be evaluated either by a comparison to the best possible schedule on the same machine or by the speed-up with respect to the sequential time.

- the cost of the computation of the schedule. This cost is directly related to the number of parallel tasks in an execution of a parallel algorithm, and can be formalized using the notion of *irregularity* [17].

The next section, written from [17], is dedicated to the definition of the scheduling complexity of a given algorithm, that gives a lower bound to the cost of the computation of any schedule for the algorithm.

3 Scheduling and irregularity

We recall definitions introduced in [17]. The model of parallel computer that we use consists of a set $\mathcal{P}$ of p processors. A processor works sequentially with its local memory and communicates with other processors via a global memory. We consider that an execution of a parallel algorithm is a set T of tasks each executing on an in-datum x taken from a set X. Let $\mathcal{O}$ be the subset of $T \times X$ of the couples (t, x) such that t executes on x.

3.1 The parallel execution problem

Abstractions of the communication overhead are usually formalized as *routing problem* or *memory access scheduling problem*. The task management overhead leads to the *scheduling problem*. If we abstract the whole overheads as the *parallel execution problem*, a solution to this problem can be given as a solution to the routing problem and a solution to the scheduling problem. The quality of a solution to the routing problem governs the time needed to simulate a PRAM by other machines with communications like DCM [21], LPRAM [1] or XRAM [29].

Following [29] we propose a unique framework, and we formalize a solution to the *parallel execution problem* as a *parallel execution scheme* (PES). A PES is a couple $(\mathcal{P}, \mathcal{S})$ where $\mathcal{S}$ is a *scheduler* that handles objects in $\mathcal{O}$. An initialized PES is a quadruple $(\mathcal{P}, \mathcal{S}, \mathcal{I}, \mathcal{D})$, where $\mathcal{I}$ is the input specification *i.e.* a mapping $\mathcal{O} \subset T \times X \to \mathcal{P} \times \mathcal{P}$, this mapping indicates where the data and the tasks are initially situated. In the same way, $\mathcal{D}$ is the output specification. It is a mapping $\mathcal{O} \subset T \times X \to \mathcal{P}$ that specifies where the tasks in T will execute and thus where the data in X has to be routed.

We also assume that tasks are indivisible. Moreover, we assume that having started one task, a processor will complete it: this means that we assume that no task migration is allowed (we refer to [4] for a detailed discussion using process migration). This is not a restriction, since a non atomic task may always be splitted in a succession of more elementary tasks, each forked at the end of the previous one.

The two problems of routing and scheduling are often considered separately but have, at least from a theoretical point of view, remarkably similar properties and are handled in similar ways.

If routing problem is addressed alone, we have $\mathcal{O} = X$ a set of packets. The scheduler $\mathcal{S}$ manages the transfers (or equivalently the memory accesses). It consists of a routing algorithm which actually routes the packets that have been scheduled by the queuing discipline [29]. The communication overhead is usu-

ally the cost of the communications themselves (on a real machine, links have a given bandwidth or accesses to a common memory can be quantified). It is unusual to associate an overhead to the computation of the specifications $\mathcal{I}$ and $\mathcal{D}$. When the routing problem is considered, these specifications are known. The problem of routing can be solved before execution (off-line) or during execution (on-line). An execution may consists of alternative phases of computations and of synchronizations [28] or such phases may execute asynchronously [10]. Every synchronization phase may consists in structured communication patterns like permutations or message may be generated dynamically and ask for unstructured patterns.

If scheduling problem is addressed, we take $\mathcal{T} = X$ a set of tasks. The scheduler $\mathcal{S}$ handles the computational tasks generated by the algorithm. It consists of a load estimator that measures the load of the machine and of a decider that assigns a schedule to the tasks [31, 7]. As opposed to the routing problem, the scheduling overhead is usually the cost of measuring the load [14] and of deciding task creation and the schedule. In other words, at task creation, the input and output specifications have to be computed. This duality leads to formalize the irregularity (as it has been done for locality) of an algorithm as being the scheduling cost. The more the scheduler is working, the more the algorithm is irregular. The problem of scheduling can be solved before execution (static scheduling) or during execution (dynamic load-balancing and load-sharing). An execution may consists of alternative phases of computations and of scheduling [8] or phases can be done asynchronously. Associated patterns may be regular (*e.g.* constant unit time tasks to distribute on processors at a given moment) or irregular if tasks are created dynamically with varying time requirements that cannot be determined in advance.

3.2 Scheduling overhead and irregularity

In the following we focus on overheads that take place during execution. Both communication and scheduling overheads have to be considered, the former usually corresponds to carry out the data exchanges while the latter corresponds to compute the specifications. In [24] the locality of a problem is the ratio of the parallel work of the best PRAM algorithm that solves the problem to the communication complexity on two processors. From there it will be easy to express the worst-case irregularity of algorithms provided that we can measure a scheduling complexity. The model of parallel machine that we use will be a p-PRAM (CREW PRAM with p processors). The parallelism is usually expressed by the following statement:

`for all` $x \in X$ `in parallel do` instruction(x)

which assigns to each data element x in X the processor indexed code(x) that is uniquely determined by x in constant time.

Instead of, generalizing the *fork* instruction [15, 2], a program will generate parallelism through statements of the following type:

`for all` $t \in G$ `in parallel do schedule`(t) (1)

The execution of this statement consists of scheduling the tasks of the precedence graph G on p processors so that the execution time is *optimal*. More precisely, `schedule` is an oracle solving the *Graph Scheduling Problem*, GSP. This problem is defined as follows:

Input. p a number of processors and G a DAG with $\#G$ nodes. Each node of G is a task. A task is a program that may be run on one processor but that may contain `schedule` instructions.
The *length* of a task is its sequential time, i.e. the number of unit time operations performed when the graph is scheduled sequentially on one processor. It may not be known until the execution completes but is independent of the way the task is scheduled, especially when it contains `schedule` operations.

Output. $L_p^\star(G)$ the optimal schedule length for the execution of G on p processors.

Notice that a task may contain branching and `schedule` instructions. Thus, GSP allows scheduling of parallel programs for which the precedence graph is unknown.

As a sub-problem, let us consider that all tasks in G are sequential (i.e. do not execute `schedule` instructions) and of known duration. In this case, solving the GSP is NP-complete and deciding whether $L_p^\star(G)$ is a given integer l is co-NP complete [16].

Scheduling complexity. Let A be a parallel algorithm (including `schedule` instructions) that solves a given problem $\mathcal{P}$. On any input x of size n, the execution of A will generate a DAG G_x containing at most $\#G_x$ sequential tasks.
Let $H(G_x)$ be the number of operations performed during the execution and $L^\star(G_x)$ be the length of the critical path of G_x. Thus, $H(G_x)$ is the total length of the tasks and $L^\star(G_x)$ is the minimal parallel time for the execution of G_x on an infinite number of processors. Notice that $H(G_x)$ and $L^\star(G_x)$ may not be known until the execution completes.

Consider now the execution of A with input x on a p-PRAM. We have the following bounds for $L_p^\star(G_x)$:

$$\lceil H(G_x)/p \rceil \leq L_p^\star(G_x) \leq \lceil H(G_x)/p \rceil + L^\star(G_x) \quad (1)$$

A lower bound for solving the GSP problem sequentially is $\#G_x$. Moreover, since no information is known about the optimal schedule computed by the oracle, every `schedule` operations may be performed on the same processor. Thus, the cost of computing the schedule by the oracle is assumed to be $\#G_x$.

Definition 1 *The (worst-case)* scheduling complexity *of A, denoted as $\iota_w(n)$, is the maximum number of tasks that can be executed concurrently for any input x of size n. $\iota_w(n)$ is upper bounded by $\#G_n = Max(\#G_x / \mid x \mid = n)$.*

Let $T_p^\star(G_x)$ be the optimal parallel time for the execution of G_x on p processors, including both execution time $L_p^\star(G_x)$ of the optimal schedule and the scheduling complexity which gives a lower bound on the time needed to compute such a schedule, whatever the lengths of the tasks are.
Corresponding worst-case values for any input x of size n are denoted $H(n)$, $L^\star(n)$, $L_p^\star(n)$ and $T_p^\star(n)$. Then, a lower bound for the whole parallel execution time, including both execution time of the optimal schedule and its computation by the scheduler oracle, is:

$$T_p^\star(G_x) \geq \mathrm{Max}\left(\lceil H(G_x)/p \rceil, L^\star(G_x)\right) + \iota_w(n) \quad (2)$$

Let $T_{seq}(n)$ be the time of the fastest sequential algorithm solving $\mathcal{P}$. In practice, parallel algorithms that provide both scalability and efficient execution are in EP [21], thus verifying:

$$\begin{cases} pT_p^\star(n) = O(T_{seq}(n)), \\ \exists \epsilon < 1 : L^\star(n) = T_{seq}(n^\epsilon) \end{cases} \quad (3)$$

Assuming that $H(n) = O(T_{seq}(n))$, (3) reduces to:

$$p\iota_w(n) = O(H(n)) \quad (4)$$

Having a polynomial speed-up implies that A can theoretically be executed efficiently on a polynomial number of processors, thus it is needed that $H(n)/\iota_w(n) = n^{\Omega(1)}$.

Definition 2 *The worst-case regularity $\rho_w(n)$ of an algorithm written with* `schedule` *instructions is the ratio of the number of operations $H(n)$ required to run the algorithm to the scheduling overhead:*

$$\rho_w(n) = \frac{H(n)}{\iota_w(n)} \quad (5)$$

As it is the case for locality [24], algorithms may be classified with respect to their regularity.

Definition 3 *An algorithm is said to be:*

- irregular *iff* $\rho_w(n) = O(1)$,
- log-regular *iff* $\rho_w(n) = \log^{O(1)} H(n)$,
- polynomially-regular *(or* regular *in short) iff* $\rho_w(n) = H(n)^{O(1)}$.

Such definitions focus much more on practical algorithms than on problems. An efficient dynamic algorithm, that can be easily expressed using dynamic scheduling, will be irregular if the scheduler contributes a lot to the efficiency. Conversely, any algorithm that includes its own scheduling will be regular, since only one `schedule` operation will be performed to start a sequential program on each of the p processors.

For instance, the computation of an n-point FFT graph is implemented with no effort with irregularity $O(1)$ by plugging the scheduling in the program. However, if we assume that the costs of basic operations are of unknown durations, the previous static mapping may lead to an inefficient execution due to the large number of synchronizations involved in the algorithm. Let us now consider the algorithm that builds the precedence graph G_n of the FFT (which is only related to n), and then executes `schedule`(G_n). Since $\#G(n) = n \log n$, this algorithm is *irregular* but its execution will be efficient.

3.3 A didactic example: parallel quick-sort

As a didactic example, we consider a variant of a parallel sorting algorithm due to Reischuk [25]. This algorithm may be seen as a generalization of the sequential quick-sort [19].

Let A be an array containing n elements to be sorted. The algorithm consists in randomly choosing $k-1$ pivot elements ($k = \sqrt{n}$) and in sorting them. In the genuine algorithm this stage is performed in parallel on n processors but for the sake of simplicity we will use a sequential sorting algorithm.

Let $p_0 = -\infty, p_1, \ldots, p_{k-1}, p_k = +\infty$ be the sorted pivot elements: two consecutive (p_i, p_{i+1}) elements thus define a bucket B_i, $0 \leq i < k$.
Computing in parallel, using one processor per element x of A, the bucket that the element belongs to, leads to a rearrangment of A.

Sorting A may then be achieved by computing in parallel either a sequential sorting (Algorithm $\mathcal{A}^{(1)}$) or a recursive sorting (Algorithm $\mathcal{A}^{(2)}$) of each bucket.

For both algorithm, irregularity depends of the number of operations performed in the sequential sorting algorithm. If this algorithm is the sequential quicksort, then, for each algorithm, the number $H^{(1)}(n)$ and $H^{(2)}(n)$ of operations performed is bounded by:

$$n \leq H^{(i)}(n) \leq n^2 \qquad i = 1, 2.$$

The scheduling complexity of algorithm $\mathcal{A}^{(1)}$ is upper bounded by $k+1$ since only one `schedule` operations may be executed that contains $k+1$ tasks. Thus the worst case regularity $\rho_w^{(1)}(n)$ of $\mathcal{A}^{(1)}$, which is related to the input array, is bounded by:

$$n^{1/2} \leq \rho_w^{(1)}(n) \leq n^{3/2}$$

and $\mathcal{A}^{(1)}$ is regular.

In the $\mathcal{A}^{(2)}$ algorithm, recursive splitting is performed until only one element remains in the bucket. Thus its scheduling complexity is $H^{(2)}(n)$. The algorithm is of regularity $\rho_w^{(1)}(n) = 1$ and $\mathcal{A}^{(2)}$ is irregular.

3.4 Authorized scheduling operations

The parallel execution scheme (PES) specifies input and output for a scheduling algorithm (i.e. a *scheduler*), but do not specify what operations a scheduler can do, except the possibility of executing a basic task – an elementary node in the PEP – on a processor. The assignment of a set of tasks by the scheduler to a given processor (referred in the following as allocation operation) is assumed to be a unit-time operation.

Additional scheduling operations are [5]:

Preemption. The scheduler is said *preemptive* if it can suspend a task during its execution on a processor, thus handling the interleaving of executions of different tasks on a processor. A contrario, a *non-preemptive* scheduler has no control of a task once it has assigned it to a processor, just getting informations when the task is finished.

Migration. When migration is authorized, a preemptive scheduler may suspend a task during its execution on a given processor and transfer it to another processor. The task will then continue its execution on the new processor.

Non-preemptive constraint is very strong, and it may be unrealistic to assume that once a task is started it must be run without any form of recourse until its unknown completion time [26]. The *non-preemptive with restarts* model authorizes to cancel a task during its execution and to restart it – from its beginning – on another processor. This operation, introduced in [26], appears in practice in some batch systems. For instance, the "Easy" scheduler available on the IBM-SP 2 [22] may kill a task if it exceeds an amount of cpu time specified in the submission of the task and later restart it on another processor.

4 On-line algorithms and competitive ratio

The previous definition of irregularity focuses more on the contribution required by a given algorithm from a scheduler to provide efficient execution.
For a practical implementation, a scheduling algorithm will be needed that is able to schedule optimally, on a p-PRAM, the precedence tasks graph generated during execution. When this graph is not completely known (especially when task durations are unknown), one needs to compute the schedule dynamically, taking decision both from machine activity and knowledge of the application. The scheduling algorithm is then an on-line algorithm.

In the general case, due to the intrinsic difficulty of scheduling (see previous section) on the one hand and to the fact that we are interested mainly in ensuring linear speed-up on the other hand, we may consider only the building of a nearly optimal schedule, whose length is within a constant factor from the optimal one. In the framework of on-line algorithms, this corresponds to the *competitive ratio*.

In this section, the theory of on-line algorithms, developed in [27], is introduced following the paper [3]. One of the main particularity concerns performances analysis of such an algorithm using the notion of *competitive ratio*. We briefly recall basic definitions, illustrating how they can be applied to evaluate an on-line scheduling algorithm.

4.1 Deterministic and randomized on-line algorithms

The theory of on-line algorithms has been developed in the framework of query-answer games. In such a game, an *on-line* algorithm has to answer a sequence of questions, trying to minimize a given cost function. The algorithm is said on-line if it answers a question knowing neither the whole sequence of questions nor its length. A contrario, an *off-line* algorithm can build its sequence of answers with the complete knowledge of the sequence of questions.

Preliminary Definitions. More precisely, a query-answer game consists of a set of possible questions Q, a finite set of answers R and a set of cost functions $f = (f_n)_{n \in \mathbb{N}}$ defined for any tuple of n elements in $Q \times R$:

$$f_n : (Q \times R)^n \longrightarrow \mathbb{R} \cup \{\infty\}.$$

f will denote the cost function[1] associated to the game.

A *deterministic on-line algorithm* A^d is a sequence of functions $a_n : Q^n \longrightarrow R$, for $n \geq 1$. For any input $\underline{q} = (q_1, \ldots, q_n) \in Q^n$, A^d produces a sequence $\underline{r} = (r_1, \ldots, r_n) \in R^n$ with $r_i = a_i(q_1, \ldots, q_i)$, $i = 1 \ldots n$. The cost of A^d on $\underline{q}$ is $c_{A^d}(\underline{q}) = f(\underline{q}, \underline{r}) = f_n(\underline{q}, \underline{r}) = f_n(\underline{q}, A^d(\underline{q}))$.

A *randomized on-line algorithm* A^p is a distribution of deterministic on-line algorithms A^d_X (X being the random variable relative to the distribution). For any input sequence $\underline{q} \in Q^n$, the output sequence $A^p(\underline{q}) = (r_1, \ldots, r_n)$ and the cost $c_{A^p}(\underline{q})$ are random variables, any r_i being related to the randomized choice of the deterministic algorithm A^d_x at step i.

The optimal off-line algorithm *Opt* is defined as the algorithm that delivers for any sequence of questions $\underline{q}$ a sequence of answers $\underline{r}$ such that $f(\underline{q}, \underline{r})$ is minimal, i.e.:

$$\begin{array}{lccc} Opt: & \bigcup_{n=1}^{\infty} Q^n & \longrightarrow & \bigcup_{n=1}^{\infty} R^n \\ & \underline{q} = (q_1, \ldots, q_n) & \longmapsto & \underline{r} = (r_1, \ldots, r_n) \end{array}$$

with $f(\underline{q}, \underline{r}) = \min\{f_n(\underline{q}, \underline{r'}) \mid \underline{r'} \in R^n\}$. Thus, the cost function $c^\star$ of the optimal off-line algorithm verifies:

$$\forall \underline{q} \in Q^n \quad c^\star(\underline{q}) = \min\{f_n(\underline{q}, \underline{r'}) \mid \underline{r'} \in R^n\}.$$

4.2 Competitive ratio.

The performance of an on-line algorithm is characterized by its competitive-ratio which measures the factor between the cost of the solution that it delivers and the best one delivered by an algorithm in a given class.

Let A^d be a deterministic on-line algorithm and $\mathcal{C}$ a class of algorithms. A^d is said *α-competitive against $\mathcal{C}$* if there exists a constant α such that, for any input sequence of requests $\underline{q}$:

$$c_{A^d}(\underline{q}) \leq \alpha \min\{c_A(\underline{q}) \mid A \in \mathcal{C}\}.$$

This definition is extended in a natural way to a randomized on-line algorithm A^p. A^p is *α-competitive against $\mathcal{C}$* if there exists a constant α such that, for any input sequence of requests $\underline{q}$:

$$E_X\left(c_{A^p}(\underline{q})\right) \leq \alpha \min\{c_A(\underline{q}) \mid A \in \mathcal{C}\}$$

where the expectation E_X is taken over all random choices in A^p.

By extension, an on-line algorithm is said *α-competitive* if it is α-competitive against the optimal off-line algorithm.

[1] i.e. $\forall n \in \mathbb{N} \quad \forall \underline{q} = (q_1, \ldots, q_n) \in Q^n \quad \forall \underline{r} = (r_1, \ldots, r_n) \in R^n : \quad f(\underline{q}, \underline{r}) = f_n(\underline{q}, \underline{r})$.

The competitive ratio gives an evaluation of the efficiency of an algorithm more precise than a worst or average-case analysis.

Two main techniques may be used to bound the competitive ratio. To give an upper bound, it is sufficient to exhibit an on-line algorithm that solves the problem within the desired upper bound. To prove a lower bound, a main technique consists in the conception of an algorithm, called the *adversary*, that builds on-line its sequence of requests $\underline{q}$ in the following way: request q_{n+1} is built from the knowledge of all previous couples (q_i, r_i), $1 \leq i \leq n$, in order to maximize the cost function $f(r_1, \ldots r_n)$.

4.3 Application to on-line scheduling.

In the framework of the scheduling problem, let G be any instance of the parallel execution problem that is to be scheduled on a p-PRAM.

The cost function $f(G)$ to minimize is the makespan or length of the schedule, denoted in the following as $L_p(G)$. With the complete knowledge of the precedence graph, the optimal off-line algorithm will compute a schedule of minimal length $L^\star_p(G)$.

Let A^d be a deterministic on-line algorithm that solves the parallel execution problem, and $L^d(G)$ be the length of the schedule computed by A^d for the instance G.

Algorithm A^d is α-competitive iff, for any instance G, $L^d(G) \leq \alpha L^\star_p(G)$.

5 On-line scheduling

Many on-line scheduling algorithms have been proposed. In [26], D. Shmoys, J. Wein and D. Williamson give different upper and lower bound for the scheduling problem on various models of parallel machines, notably on the *identical machines model* – that correspond to the p-PRAM that we consider in this paper –, the *uniformly related machines* model – speed ratio between two machines is constant – and the *unrelated machines* model – speed ratio between two machines is constant for a given task but may vary from one task to the other –.

In a first part, we consider the basic on-line algorithm given by Graham [18] and its competitive ratio analysis. Then lower bounds for this problem, introduced in [26], are presented.

In a second part, we focus on the problem of scheduling many independent tasks with unknown duration. We recall the fundamental deterministic algorithm from R. Cole and U. Vishkin [8].
We then present a coarse-grain variant of this algorithm. which achieves an asymptotic $1+\epsilon$ competitive ratio for this problem and whose overhead, due to the

scheduling computation, is bounded by the parallel time.

The main consequences, for the considered case where machines are uniformly related, are:

- a non-preemptive deterministic on-line scheduling algorithm with a competitive ratio of $\left(2-\frac{1}{p}\right)$ exists [18].
- a lower bound for the competitive ratio of non-preemptive deterministic on-line (respectively preemptive deterministic on-line and non-preemptive randomized on-line) is at least $\left(2-\frac{1}{p}\right)$ [26].

5.1 Greedy on-line scheduling algorithm

List scheduling algorithms are a classical way of scheduling tasks. Such algorithms consists in managing lists of executable tasks. When a processor terminates the execution of a task, it frees its successors corresponding to precedence constraints. If there are still executable tasks, it gets one of them and begins its execution. If no more tasks are executable, the processor stays inactive until new tasks become executable.

A list algorithm usually specifies an order on the tasks by assigning to each task a priority (for instance depending on the structure of the graph or on the lengths of the tasks when they are known). In the on-line case, since task duration is assumed unknown, the greedy algorithm allocates tasks without taking care of any priority.
Besides, the graph may be determined dynamically, from execution of " `schedule`" instructions.

Let us notice that we do not consider here the cost of computing the schedule, notably to assign distinct tasks to distinct processors. However, a lower bound for this cost is n if a graph with n tasks is to be scheduled (cf paragraph 3.2).

5.2 Tight bounds for non-preemptive scheduling

Proposition 1 *The greedy scheduling algorithm has competitive ratio* $\left(2-\frac{1}{p}\right)$ *on the p-PRAM.*

Proof. Let G be a precedence graph, $L^\star(G)$ the optimal scheduling time using the off-line algorithm on the p-PRAM. The – unknown – duration of a task t in G is denoted $l(t)$. Let $L(G)$ be the length of the schedule given by the greedy algorithm.

Let $I(G)$ be the total idle time in the greedy schedule. We have:

$$L(G) = \frac{I(G) + \sum_{t\in G} l(t)}{p}. \tag{6}$$

Since $L^\star(G) \geq \frac{\sum_{t\in G} l(t)}{p}$, (6) leads to:

$$L^\star(G) \geq L(G) - \frac{I(G)}{p}. \tag{7}$$

Let C be any path in the DAG G ; we have $L^\star(G) \geq \sum_{t\in C} l(t)$.
The greedy algorithm is such that, at any time, at least one processor is executing a task. Moreover, if at a given time a processor is idle then there exists a task on a critical path which is being executed on one processor. In the Gantt diagram of the greedy schedule, consider the instants θ_i, $1 \leq i \leq \Theta$, where at least one processor is idle. Let t_i be a unit operation executed at θ_i and that belongs to a critical task path. Each unit operation t_i belonging to a critical task, we have:

$$\Theta \leq L^\star(G) \tag{8}$$

which gives the following bound on the idle time:

$$I(G) \leq (p-1)\Theta \leq (p-1)L^\star(G) \tag{9}$$

Using this bound for $I(G)$ in (7) leads to:

$$L(G) \leq L^\star(G)\left(1+\frac{p-1}{p}\right) \tag{10}$$

which concludes the proof. □

Thus the greedy scheduling algorithm, which computes a schedule with a competitive ratio lower than 2, gives a satisfying solution (with asymptotic constant inefficiency) to the GSP problem.

Corollary 1 *Let A be a parallel algorithm with constant inefficiency, polynomial speed-up and polynomial regularity and that executes $H(n)$ operations.*
Then, $\exists M \geq 1, \exists \delta > 0$ such that $\forall n$, $\forall p \leq n^\delta$, A may be executed in parallel time

$$T_p(n) \leq M\frac{H(n)}{p}$$

on a p-PRAM.

Proof. It is sufficient to schedule the parallel algorithm using a centralized greedy scheduling algorithm. Each time a processor becomes idle, it asks for a specific processor (the *master*) for a new task. The `schedule` instructions are directly forwarded to the master processor, which allocates at most one task at each step.

On the one hand, since the algorithm has constant inefficiency and polynomial speed up, there exist M and k such that $\forall p \leq H(n)^k$:

$$L_p^\star(n) = M\frac{H(n)}{p}.$$

On the other hand, since the regularity is polynomial, we have $\rho_w(n) = H(n)^{O(1)}$.

Let us consider the execution of A using the greedy scheduling algorithm on $p \leq \text{Min}(H(n)^k, \rho_w(n))$ processors. Since the number of tasks generated by the algorithm is upper bounded by $\frac{H(n)}{\rho_w(n)}$, the parallel overhead due to the computation of the greedy schedule is bounded by the same term.

Besides, from proposition (1), $L_p(n) \leq 2L_p^\star(n)$. We obtain:

$$T_p(n) \leq 2M\frac{H(n)}{p} + O\left(\frac{H(n)}{\rho_w(n)}\right) \qquad (11)$$

which leads to $T_p(n) \leq O\left(\frac{H(n)}{p}\right)$. □

On-line greedy scheduling is then an efficient way to schedule regular efficient parallel algorithms with unknown precedence graphs.

5.3 Lower bounds for the greedy scheduling competitive ratio

A natural question is then to determine if it is possible to have a better competitive ratio than $\left(2 - \frac{1}{p}\right)$, either on the same model or by considering larger classes of scheduling algorithms.

This problem has been studied in [26], in which the following proposition is proved.

Proposition 2 *[26] On the p-PRAM, the competitive ratio is lower bounded by* $\left(2 - \frac{1}{p}\right)$ *for any scheduling algorithm of the following classes:*

1. *Deterministic with no preemption.*
2. *Deterministic with preemption.*
3. *Randomized with no preemption.*

Proof. We only sketch the proof for the first case. The complete proof for this theorem is given in [26].

The adversary builds the following instance G due to Graham [18]. G contains $1 + p(p-1)$ independent tasks. One task α_1 is of length p, while other tasks β_k, $1 \leq k \leq p(p-1)$ are of length 1.

The optimal schedule is of length p. It executes the task α_1 on a given processor, and the $p(p-1)$ unit tasks β_k on the $p-1$ remaining processors.

The length of any schedule of G is equal to $p + t$, where t is the time when the task α_1 starts its execution. Since the tasks durations are unknown for the scheduling algorithm, the adversary strategy will thus consist in making t as large as possible.
The tasks that are processed first are then the $p(p-1)$ unit time tasks β_k, that are executed in $p-1$ time units with no idle time. Then, at time $t = p-1$, the task α_1 starts its execution. The length of the obtained schedule is then $2p-1$, which provides the desired lower bound. □.

When the task precedence graph is unknown, it is then impossible to build an on-line algorithm with better competitive ratio than the greedy algorithm.

To build more efficient on-line scheduling algorithm, one may try to restrict the problem. In this framework, it is possible to consider asymptotic competitive ratio for specific graph structures, introducing little knowledge on the tasks durations.

5.4 Independent tasks scheduling.

In this paragraph, we consider the following problem introduced by R. Cole and U. Vishkin in [8] and referred in the following as the "Independent Task Scheduling Problem" (ITSP for short). ITSP is a restriction of the GSP problem, defined as follows [8]. n tasks are given, each of unknown length bounded between 1 and $c(n)$. The total length of the tasks is bounded by $H(n)$ ($c(n)$ and $H(n)$ are at most t polynomials in n).
The problem $ITSP$ consists in scheduling the tasks on a p-PRAM so that the tasks are completed in time $O(\max\{H(n)/p, c(n)\})$.

In [8], a sophisticated solution is built that needs neither preemption nor migration. This solution leads to the following proposition:

Proposition 3 *[8] If* $H(n) = O(n)$ *and* $c(n) = O(\log n)$*, then, for any* $p \leq \log n$*, it is possible to solve the ITSP problem on the p-PRAM in time:*

$$T_p(n) = O(H(n)/p)$$

including scheduling overheads.

For practical reasons, constant complexity factors being pretty huge, it is interesting to consider coarse grain scheduling algorithms.

In the following, we consider the following variant of the $ITSP$ problem:

input: p an integer and n independent tasks with length bounded between $T_m(n)$ and $T_M(n)$.

output: Execution of the n tasks on the p-PRAM.

This instance may be scheduled using the following parallel scheduling algorithm on the p-PRAM:

1. Initialization: assign $\log p$ tasks to each processor and store the $r = n - \frac{p}{\log p}$ others in an array $R[1..r]$.

2. Execution: in parallel each processor executes $\log p$ computation steps with the tasks assigned to it. Let F_k be the number of terminated tasks by the processor k during this stage.

3. Redistribution: Let $\pi_k = \sum_{i=1}^{k} F_i$ (prefix sum computation). Let $i_0 = r - \pi_p$.
If r is positive, in parallel for any k, each processor k takes the tasks indexed $i_0+\pi_{k-1}, \ldots, i_0+\pi_k$. Let $r = r - \pi_p$.
If r is negative, then empty R by distributing evenly tasks among the processors so that the number of tasks of two different processors may differ at most of one; then let $r = 0$.

Proposition 4 *The coarse grain scheduling algorithm schedules n tasks on $p \leq \frac{n}{\log^2 n}$ processors with a competitive ratio bounded by:*

$$1 + \frac{p}{\log p} \frac{T_M(n)}{T_m(n) H(n)}.$$

Proof. Since $p \log p$ operations are performed during step "Execution" when r is positive, the number of iterations is bounded by: $\frac{H(n)}{p \log p}$. The cost of the redistribution is bounded by $\log p$ (prefix computation).

Finally the parallel time, including both the schedule of the tasks and its computation, is bounded by:

$$T_p(n) \leq \frac{H(n)}{p} + \log p T_M(n) \qquad (12)$$

A lower bound for the length of the schedule given by the optimal off-line algorithm is: $\frac{H(n)}{p} T_m(n)$. We thus obtain the desired upper bound for the competitive ratio. □.

Corollary 2 *Let $\epsilon > 0$ be an arbitrary constant. If $T_M \leq \epsilon (T_m \log n)^2$, the competitive ratio of the coarse grain scheduling algorithm for the ITSP problem is $(1+\epsilon)$ on the p-PRAM for any $p \leq \frac{n}{\log^2 n}$.*

Proof. Direct from previous proposition. □

Application to the parallel quick-sort algorithm. We turn back to the parallel sorting algorithm $\mathcal{A}^{(1)}$. For the sake of simplicity, we consider the average case where $T_M(n) = \Omega(n^{1/2} \log n)$ and $H(n) = \Omega(n \log n)$. From proposition 4, we have:

$$T_p(n) \leq \Omega \left(\frac{n \log n}{p} + \log p n^{1/2} \log n \right).$$

Executed with the coarse-grain scheduling algorithm, the execution will provide optimal execution time for $p \leq \frac{\sqrt{n}}{\log n}$.

5.5 Complementary results.

To conclude this section, a few extensions of previous on-line scheduling algorithms are reviewed.

Tasks with release dates scheduling. For some scheduling problem, that have not been considered in this paper, tasks may be generated dynamically not only due to precedence constraints, as studied previously, but after release dates. In the on-line framework, those release dates are assumed unknown. This problem captures notably interactive multi-users scheduling.

In [26], the following scheduling algorithm is introduced. Let A be an on-line algorithm solving the ITSP problem, with competitive ratio α. A is used to build a scheduling algorithm B dedicated to tasks with unknown release dates.

At time T_0, B applies A to schedule executable tasks. At T_1, once every tasks are terminated, B applies A to schedule all new executable tasks. Such a task may either have been generated due to satisfied precedence constraint or have a release date bounded between T_0 and T_1.
After each application of algorithm A, B updates the executable tasks and keeps on scheduling.

The important fact is that this very simple algorithm B has a competitive ratio upper bounded by 2α [26]. This justify the importance of having efficient scheduling algorithm for independent tasks.

Dynamic trees. Another way of handling tight bound for the competitive ratio, is to focus on some specific graph structures. In this framework, tree are of interest and have been studied in [23].

Many other extensions have been proposed, notably to take in account communications [5, 23] or the topology of the parallel machine [13].

6 Athapascan approach

The formalization of the parallel execution problem presented in section 3.1 leads us to define a parallel programming model, *Athapascan*, that allows to schedule tasks according to their precedence constraints. This programming model separates the expression of a given algorithm from the way it is scheduled on a given machine.

Let us for instance consider the programming of a parallel Gaussian elimination on a dense floating point matrix, each column elimination being performed sequentially.
On a dedicated parallel machine, tasks durations may be assumed to be known and a cyclic scheduling will

be efficient with a neglectible scheduling overhead. However, this in not true anymore if the same algorithm is executed on a network of workstations shared between several users. Here, a dual modelization of the unpredictable activity of other users may be to consider that basic tasks in the Gaussian elimination are of unknown duration. In this case, an online scheduling (for instance the greedy scheduling) is needed to provide an efficient execution under any circumstances[2]. The program then remains the same, only the scheduling algorithm has been customized.

In this section, we focus on the presentation of the ATHAPASCAN-1 programming model. ATHAPASCAN-1 is a parallel library (written in C++) and implemented on the ATHAPASCAN-0 kernel. ATHAPASCAN-0 is a basic kernel that provides remote thread creation, group communications and global memory accesses. This kernel is implemented on the top of MPI and a kernel of POSIX threads.

ATHAPASCAN-1 may be seen as an implementation of the programming model described in section 3, that provides an instruction similar to `schedule(G)` where G is a precedence graph.

6.1 Graph building, interpretation and execution

To build the precedence graph that corresponds to the execution of an algorithm, a user has to write a concrete class that derives from the abstract class `ExecutionGraph`.
This class includes builders and access operators. Especially, a given graph G_2 may be embedded in another G_1, with a specification of the operational semantics of the embedding which may be of four types:

- sequential composition: `G1.After( G2 )`. The graph G_2 will be scheduled only when all tasks of G_1 are completed.
- parallel independent embedding: `G1.IndependentParallel ( G2 )`. Graphs G_1 and G_2 contains independent tasks.
- parallel concurrent embedding: `G1.ConcurrentParallel ( G2 )`. Tasks G_1 and G_2 have to be executed concurrently. A special case is when two tasks (one in G_1, the other in G_2) may communicate each one to each other.
- pipelined composition: `G1.LinkedUp( G2 )` . In this case, it is required that the number of output tasks in G_1 be the number of input tasks in G_2. Each output task in G_1 is then linked up to the corresponding input task in G_2, i.e. the one with the same index.

To ease the building of a specific graph, it is possible to derive a user class from some specific abstract subclass of `ExecutionGraph`. For instance, the `SplitComputeMerge` class (SCM for short) can be used to build precedence graphs with independent tasks.

For each graph object G, a method `G.Execute ( in Scheduler S )` is used to submit the scheduling of tasks in G to the scheduler object S.

6.1.1 Building the precedence graph

In each concrete class which inherits from `ExecutionGraph`, client and server stubs specifying the operational semantics of a task have two be performed.

For instance, a concrete class that inherits from SCM has to specify the following methods which are virtual in SCM:

- `Split (in int i, in int K, out athOStream )` : specifies the client stub – locally executed – corresponding to the i^{th} task among K independent ones. The output stream is used to transmit arguments to the server stub.
- init(**in** int i, **in** int K, **in** athIStream) is the server stub corresponding to the Split client function.
- `main()` specifies the function to be performed on the processor when the server task is scheduled.
- end(**in** int i, **in** int K, **in** athOStream) is the server stub corresponding to the Merge client function.
- `Merge (out int i, out int K, int AthIStream` specifies the merge function and the corresponding client stub.

6.1.2 The `Scheduler` class

The abstract class `Scheduler` provides a standard interface for a scheduler. Each specific implementation of a scheduling algorithm which inherits from `Scheduler` has to implement the following virtual methods:

- `Execute ( in Graph G )` : this is the symmetric method of the `Execute` method in the class `ExecutionGraph`.

[2] let say LaTeX-safe...

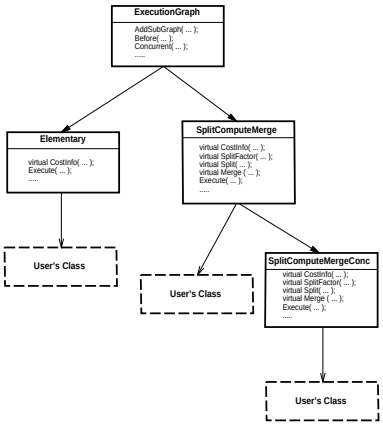

Figure 1: Hierarchy of class `ExecutionGraph`.

- vector<Processor> GetProcessors (**in** int k): delivers in output a vector of $p \geq k$ processors. This interface is used for the implementation of the `Execution` method in the class `ExecutionGraph`.

6.2 Extensions: locality and granularity

Two important features proposed in ATHAPASCAN are locality specification and adaptive granularity.

Locality specification. It is possible to associate a locality attribute to each elementary task of a graph, which may be assigned to the site of a previously scheduled task. This attribute may be used by some scheduling algorithm to provide affinity scheduling.

Adaptive granularity. It is also possible to propose to the scheduler an interval for the splitting factor and let it choose one in the interval. This can used to adapt the granularity of a program to the one of a machine. This may be of particular interest for a regular algorithm: depending on the behavior of the hardware support, the scheduler may choose the factor.

In addition, for many parallel algorithms, it is possible to associate – at least approximative – cost informations about the tasks to be scheduled. The semantics of those costs is dependent on a given scheduler. For instance, some list scheduling algorithm may use cost informations for ordering tasks in a scheduling queue.

6.3 Programming the parallel quick-sort algorithm

We consider the practical implementation of the regular algorithm $\mathcal{A}^{(1)}$ for parallel sorting. We used the fact that its execution is optimal for any $p \leq \frac{\sqrt{n}}{\log n}$ using the coarse-grain scheduling algorithm.

6.3.1 The program

The program inherits from the abstract class SCM. The sequential sorting function is `qsort` of the standard C library. The elements to be sorted are 32-bits integers and the comparison function is: `int IntCompare (int *i, int *j) { return ( *i - *j ) ; }`.

6.3.2 Experimental results

Two different scheduling algorithms have been used: a mapping computed from cost predictions and a centralized greedy algorithm with thresholds. The experimentations have been performed for various size arrays on an IBM - SP2 (32 processors).

Cost-prediction algorithm. If n_i is the number of elements of bucket B_i, the cost of the sequential sorting may be evaluated to $n_i \log n_i$, average cost of the quick-sort algorithm. A mapping that takes into account this information is then computed.

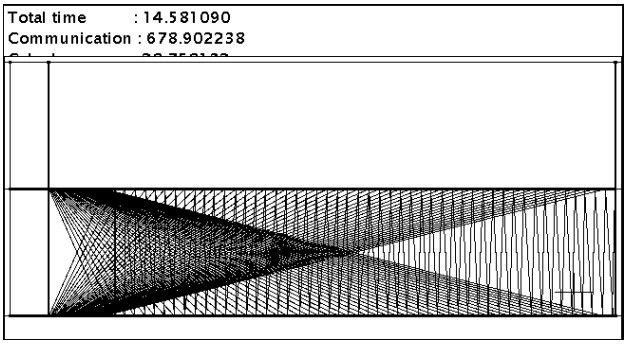

Figure 2: Mapping of the algorithm executed on 2 processors. Bold lines represent threads, and thin lines communications.

Figure 1 presents an execution of the algorithm with the list scheduler. Communication overheads are overlapped by computations. The small variance of the sequential quick-sort cost makes this strategy efficient on a uniform machine.

Centralized greedy algorithm. Tasks to be computed are centralized on a given processor, the *master* and load of slave processors ranges between two thresholds S_m and S_M. When the number r_i of tasks on a given *slave* processor becomes lower than a given threshold S_m, the processor requests for $S_M - r_i$ tasks to the master. Figure 3 gives performance results. Tuning the thresholds S_m and S_M is very important to get efficiency. This on-line approach leads to good

results: 10 millions of 32 bits integers are sorted in 156 s in sequential and in 8 s on 29 processors, corresponding to an efficiency of 67%.

7 Summary and Conclusions

The theoretical framework of on-line algorithms can be used to analyze dynamic scheduling algorithms. Two overheads are to be considered: the cost of the determination of the schedule and the quality of the computed schedule itself. Both costs are related to the parallel algorithm that is to be scheduled.

The first is related to the scheduling complexity which is used to define the regularity of an algorithm. Intuitively, the less regular the algorithm is the more costly to schedule efficiently it is.

The second may be captured by the competitive ratio of the algorithm. Even for an unknown graph, the optimal competitive ratio is $(2 - 1/p)$ on p uniformly related processors, which is realized by the greedy algorithm.

To improve the competitive ratio, it is then needed to restrict the parallel execution problem to specific graph families, such as independent tasks scheduling.

An interesting question remains to analyze if theoretical competitive ratio corresponds to experimental results.

Acknowledgements

We would like to thank Gilles Villard for his contribution in the definition of the scheduling problem. The introduction to on-line and coarse grain algorithms for scheduling of independent tasks was inspired by François Galilée. Mathias Doreille contributed with his work on parallel algorithms, notably quick-sort variants and unknown duration Gaussian elimination. Gerson Cavalheiro helped defining and implementing the class hierarchy of ATHAPASCAN-1.

References

[1] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAM's. *Theoretical Computer Science*, 71:3–28, 1990.

[2] J.L. Balcázar, J. Diaz, and J. Gabarró. *Structural Complexity II.* Springer-Verlag, 1989.

[3] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In New York ACM, editor, *22nd Annual ACM symposium on Theory of Computing*, pages 379–386, May 1990. Baltimore, Maryland.

[4] G. Bernard, D. Steve, and M. Simatic. Placement et migration de processus dans les systèmes reépartis faiblement couplés. *TSI*, 10 (5):375–392, 1991.

[5] J. Blazewicz, K. Exker, G. Schmidt, and Węglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, 1993.

[6] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

[7] T.L. Casavant and J.G. Khul. A taxonomy of scheduling in general-purpose distributed computing sy stems. *IEEE Transactions on Software Engineering*, 14:141–154, 1988.

[8] R. Cole and U. Vishkin. Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM J. Computing*, 17(1), 1988.

[9] S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[10] M. Cosnard. A comparison of parallel machine models from the point of view of scalability. In *proceedings of the 1rst Int. Conf. on Massively Parallel Computing Systems, Ischia, Italy*, pages 258–267. IEEE Computer Society Press, May 1993.

[11] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures.* Thomson Computer Press, 1995.

[12] P. Emde Boas. Machine models and simulations. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.

[13] A. Feldmann, M.-Y. Kao, and J. Sgall. Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 642–651. ACM Press, 1993.

[14] D. Ferrari and S. Zhou. An empirical investigation of load indices for load-balancing applications. In *Proc. Performance'87, 12th IFIP WG7.3 International Symposium on Computer Performance, Brussels Belgium.* Elsevier Science Publishers, 1987.

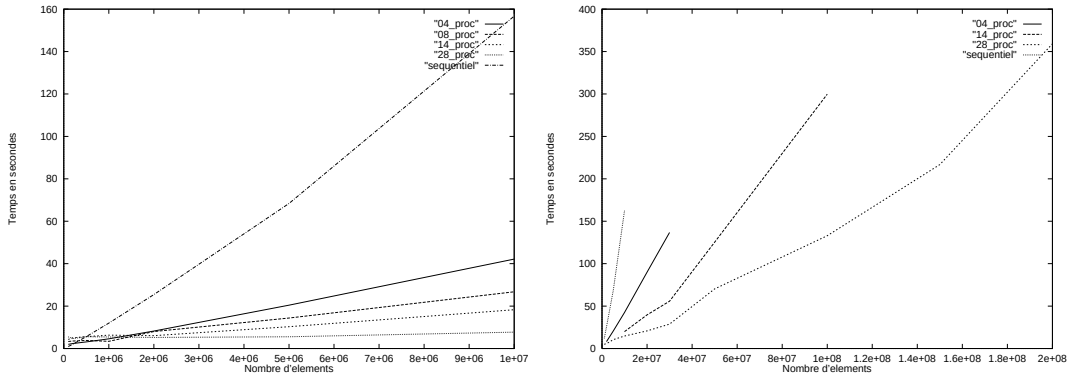

Figure 3: Experimental speed-up for sorting arrays of various sizes.

[15] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118. ACM Press, 1978.

[16] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[17] T. Gautier, J.L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In *Proc. of IRREGULAR'95, Lyon, France*. Springer-Verlag, Sep. 1995.

[18] R.L. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, 45:1563–1581, 1966.

[19] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[20] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.

[21] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[22] D.A. Lifka, M.W. Henderson, and K. Rayl. User's Guide to the Argonne SP Scheduling System. Technical Report ANL/MCS-TM-201, Argonne National Lab, May 1995.

[23] M.A. Palis, J.-C. Liou, S. Rajasekaran, S. Shende, and D. Lei. Online scheduling of dynamic trees. *Parallel Processing Letters*, 5(4):635–646, 1995.

[24] A. Ranade. A framework for analyzing locality and portability issues in parallel computing. In *Parallel Architectures and their Efficient Use*, pages 185–194. Springer-Verlag L.N.C.S. 678, 1993.

[25] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Computing*, 14(2):396–409, 1985.

[26] D. B. Shmoys, J. Wein, and P. Williamson. Scheduling parallel machines on-line. *SIAM, J. Comput.*, 24(6):1313–1331, 1995.

[27] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rule. *Communication of the ACM*, 28(2):202–208, February 1985.

[28] L. Valiant. A bridging model for parallel computation. *Communication ACM*, 33:103–111, 1990.

[29] L. Valiant. General purpose parallel architectures. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 944–971. Elsevier, 1990.

[30] M.H. Willebeek-Le-Mair and P. Reeves. Strategies for dynamic load-balancing on higly parallel computers. *IEEE Parallel and Distributed Systems*, 4(9):979–993, 1993.

[31] S. Zhou. A trace-driven simulation study of dynamic load-balancing. *IEEE Trans. on Software Engineering*, 14(9), 1988.