Proc. Irregular'95, SPRINGER-VERLAG L.N.C.S., vol. 980, pp 1-26

Regular versus irregular problems and algorithms.

T. Gautier, J.L. Roch and G. Villard *

LMC-IMAG, 46 av. F. Viallet, F38031 Grenoble Cedex

Abstract. Viewing a parallel execution as a set of tasks that execute on a set of processors, a main problem is to find a schedule of the tasks that provides an efficient execution. This usually leads to divide algorithms into two classes: *static* and *dynamic* algorithms, depending on whether the schedule depends on the indata or not. To improve this rough classification we study, on some key applications of the STRATAGÈME project [21, 22], the different ways schedules can be obtained and the associated overheads. This leads us to propose a classification based on regularity criteria *i.e.* measures of how much an algorithm is regular (or irregular). For a given algorithm, this expresses more the quality of the schedules that can be found (irregular versus regular) as opposed to the way the schedules are obtained (dynamic versus static).

These studies reveal some paradigms of parallel programming for irregular algorithms. Thus, in a second part we study a parallel programming model that takes into account these paradigms to free the user from task scheduling. An implementation, PAC++, is presented.

1 Introduction

There are two main differences between sequential and parallel computers: in the latter, there are potentially very large overheads in communication (or memory accesses) and in task management. Independently of these overheads and beyond the NC class [12], the notion "amenable to a good parallel solution" may be captured by classifying problems and parallel algorithms with respect to their *nonoptimality* or *inefficiency*, *i.e* the extra amount of work done by the parallel algorithm as compared to a sequential algorithm [43, 45]. This can be also expressed using the concept of *scalability*, *i.e* the ability to provide a linear scaling of performance as function of the cost of the machine [65].

Communications: locality. The communication overhead may be modeled using various models of the parallel random access machine, PRAM, family [26, 43]. Much work has been done in this field. This has led to classify problems and parallel algorithms with respect to their *gross locality*, *locality* for short, *i.e.* the

* The work presented here was in part supported by the STRATAGÈME project of the french *Ministère de l'Enseignement Supérieur et de la Recherche.*

ratio of the computational complexity by the communication complexity [56]. 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, to exploit locality will be a main issue to acheive high performance.

The notion of locality may also be 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 an algorithm is local the more it is regular. But this is not sufficient [13]: another relevant criterion for the irregularity of an algorithm is the irregularity of the communication patterns it involves: "a key factor success is the use of regular global communication patterns". The locality concept relies on the fact that models of parallel computation emphasizes that the tasks of computation and of communication can be distinguished [67].

Scheduling: worst-case irregularity. Obviously, task management is also a main issue toward efficiency of parallel programs. Optimal solutions for key problems such as *list ranking* may rely on task scheduling solutions [11]. There is also a context for parallelism where tasks have costs that cannot be determined in advance (*e.g.* indata dependent costs). From that angle, concepts are not so well established than concerning communication overheads. For instance, Brent's theorem does not handle the problem of assigning processors to their jobs. One can divide algorithms into two classes: *static* and *dynamic*. The former are characterized by that the structure of their executions is known in advance, the computational steps remain the same regardless of indata; on the other hand, the latter may execute differently depending on the indata. Dynamic algorithms often have static parts which may be referred as *basic blocks* [2]. But this division into static and dynamic algorithms is very poor and focus more on the way a schedule can be computed than on the quality of the executions that it provides.

Thus, since the scheduling is a key point for efficiency of executions, it is much interesting to try to classify algorithms with respect to the difficulty to schedule them finely *i.e.* in some sense, with respect to the cost-overhead due to scheduling. In this paper we are going to capture this by defining the *irregularity* of problems and algorithms. This will be done in a natural way, one may have an approach dual to the one used for communication overheads. Following the definitions of locality and informally, the more an algorithm is *irregular*, the more it must be difficult or costly to schedule the tasks it generates so that the resulting execution is efficient. Even if resources needed for executions may heavily depend on the indata, only worst-case executions are usually considered from a theoretical point of view: this first notion of irregularity will be called *worst-case irregularity* at section 2.

Local efficiency and irregularity. When measuring invariants of an algorithm, the yardstick that is used is either a sequential algorithm to capture the inherent parallelism, or a parallel algorithm to capture the communication complexity. Quite often, only worst-case execution times are considered. We think this can be far from reality when execution time varies a lot with the indata, and we will revisit the definition of efficiency at section 3. Indeed, from a practical point of view, one usually compare the parallel execution to the sequential one with the same indata. Limitations of worst-case definitions are crucial for NP-complete problems or algorithms that require exponential serial time. These latter may well be classified as "efficient algorithm" in the EP class (Efficient/Polynomially fast [45]), even if it is inherently difficult to dynamically load-balance their executions, so that, for any fixed indata, the implementation is efficient (we will speak of *local efficiency*). This is well known and has been also somehow illustrated by now anectotal superlinear speed-ups in various domains [46, 70, 57, 48]. This will lead to a second measure of irregularity in the definitions at section 3, where usual efficiency will be preferably computed for executions (local efficiency) than for algorithms (worst-case efficiency).

Irregular algorithms versus irregular data-structures. Irregularity is often considered in the literature. Two aspects of the notion appear: *irregular algorithms* and *irregular data structures*. As it can be underlined, we have chosen to measure irregularity on algorithms rather than on data structures. However we will consider both approaches to be equivalent. This is true in many cases, especially when a "satisfying distribution of the data structure" on which the algorithm operates is a "satisfying distribution of the work-load" and thus corresponds to a task scheduling with efficient execution. Indeed, a data structure is irregular when the cost to operate on parts of it is not exactly known or is unknown by advance. Parallelization approaches that are based on the splitting of such structures may consequently lead to a bad distribution of the load *a priori* and require dynamic load-balancing. Examples among various others can be found in [21, 22]. Especially for image processing [49], for a dictionary machine [15, 18] and for branch and bound algorithms [23], the irregularity of the data structure is an imbalance of the load.

Organization of the paper. The paper is organized as follows. We focus at section 2 on worst-case studies and try to define irregularity with respect to scheduling complexity. Regular and irregular strategies (or patterns) of scheduling will be associated to different costs. As noticed earlier by several authors we try to emphasize a duality between routing and scheduling. We believe that this leads to a better understanding of irregularity (at least in worst-case studies). This is formalized by the definition of *the parallel execution problem* [68]. Then at section 3 we recall well known facts on efficiency: from many practical points of view, when executions times are indata-dependent, a local efficiency (for fixed indata) better suits to measurements. Load-balancing will be necessary to have globally efficient algorithms (locally efficient everywhere): this will be another aspect of irregularity.

From these two first sections, some basic concepts will appear to be useful for parallel programming of irregular algorithms. We identify them at section 4 and show how they can be implemented using generic C++ classes at section 5.

2 Worst-case irregularity

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

2.1 The parallel execution problem

Abstractions of the communication overhead is 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 the DCM [44, 45], the LPRAM [1] or the XRAM [68].

Following [68] 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 $\mathcal{O} \subset \mathcal{T} \times \mathcal{X} \rightarrow \mathcal{P} \times \mathcal{P}$, 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 $\mathcal{O} \subset \mathcal{T} \times \mathcal{X} \rightarrow \mathcal{P}$ that specifies where the tasks in $\mathcal{T}$ will execute and thus where the data in $\mathcal{X}$ has to be routed. We assume that having started one task, a processor will complete it (we refer to [6] for a detailed discussion using process migration). We also assume that tasks are indivisible.

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. For instance, when the model of machine incorporates barrier synchronization [39, 67], it is noticed in [67] that in a "general dynamic load-balancing situation there also exist phenomena that are compatible with barrier synchronization" (this is not clear from a practical point of view [8, 3, 29]). Similarities between the two problems may also be pointed out when load-balancing are characterized by the communication patterns they involve [25]; a scheduling overhead can be viewed as a communication overhead. Further, as surprisingly noticed in [66], "general many to many routing can be reduced to sorting plus load-balancing".

The routing problem. If routing problem is addressed alone, we have $\mathcal{O} = \mathcal{X}$ a set of packets. The scheduler $\mathcal{S}$ manages the transfers (or equivalentely the memory accesses). It consists of a routing algorithm which actually routes the packets that have been scheduled by the queing discipline [68]. The communication overhead is usually 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 [67] or such phases may execute asynchronously [13]. Every synchronization phase may consists in structured communication patterns like permutations or message may be generated dynamically and ask for unstructured patterns.

The scheduling problem. If scheduling problem is addressed, we take $\mathcal{O} = \mathcal{T}$ a set of tasks. The scheduler $\mathcal{S}$ handles the computational tasks generated by the algorithm. It consists of a load estimater that measures the load of the machine and of a decider that assigns a schedules to the tasks [72, 9]. As opposed to the routing problem, the scheduling overhead is usually the cost of measuring the load [24] and of deciding task creation and the schedule. In other words, at task creation, the input and output specifications has to be computed. This duality leads to formalize the irregularity (as it has been done by locality) of an algorithm as being related to a scheduling complexity. 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 [11] 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.

Routing and scheduling may be handled simultaneously. In static scheduling studies such as in [63, 20, 37, 71], the problem is to schedule tasks given that a communication between two tasks that are mapped onto different processors has a nonzero cost. There, the problem is to compute an output specification $\mathcal{D} : \mathcal{T} \times \mathcal{X} \rightarrow \mathcal{P}$ to minimize a given cost [52].

2.2 Worst-case 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 whilst the latter corresponds to compute the specifications. In [56] the locality of a problem is the ratio of the parallel work of the best PRAM algorithm that solves the problem by the communication complexity on two processors. From there it will be easy to express the worst-case irregularity of algorithms and problems provided we can measure a scheduling complexity. The model of parallel machine we use for that will be a p-PRAM (CREW P-RAM with p processors). The parallelism is usually achieved by the following statement:

`for all` $j \in J$ `in parallel do` instruction(j)

this statement assignes to each element j in J the processor indexed code(j) that is uniquely determined by j in constant time.

Instead, generalizing the *fork* instruction [26, 4], a program will generate parallelism through statements of the following type:

`for all` $t \in \mathcal{T}$ `in parallel do schedule`(t) (1)

Here the execution of the statement consists of scheduling the tasks in T so that they are completed in optimal time. A task is a program to be run by one processor. The *length* of a task is its sequential time: it may not be known until the execution completes. Processors are chosen by a scheduling mechanism that solves the *Task Scheduling Problem*, TSP. This problems is defined as follows [11]. Polynomially many tasks are given, each of length between 1 and $c(n)$ and the total length of the tasks is bounded by $w(n)$ ($c(n)$ and $w(n)$ are at most polynomials in n). The problem is to schedule the tasks on a p-PRAM so that the tasks are completed in time $O(\max\{w(n)/p, c(n)\})$. If p is upper bounded by $w(n)/c(n)$ the time is optimal.

We assume the TSP is solved in time the ratio of the number tasks $\#\mathcal{T}$ divided by the number of processors. We will say that the *scheduling complexity* of one call to the scheduler is $\max(1, \#\mathcal{T}/p)$; the *scheduling complexity* of an algorithm is the sum of the scheduling complexity of all the calls to the scheduler.

Definition 1. The worst-case regularity $\rho_w(n)$ of an algorithm on p processors written with `schedule` instructions is the ratio: the parallel work $w(n)$ required to run the algorithm divided by the scheduling complexity. The worst-case irregularity $\iota_w(n)$ is the scheduling complexity.

This definition focus much more on practical algorithms than on problems. A dynamic algorithm, that can be easily expressed using dynamic scheduling and attain good efficiency, will be irregular if the scheduler contributes a lot to the efficiency. Solving the TSP it has been established in [11] that list ranking can be computed in optimal logarithmic time on a PRAM. We can easily write the corresponding algorithm using the scheduler, since only one call is sufficient to schedule n tasks on $p = n/\log n$ processors, the irregularity is $\iota_w(n) = \log n$. Conversely, a static algorithm that is easily expressed using a static mapping will be regular. The computation of an n-point FFT graph is implemented with no effort with irregularity $O(1)$ on n processors. To implement the time optimal list ranking with irregularity $O(1)$ (without calling the scheduler) using the rather sophisticated method in [11] would be tedious; but an inefficient list ranking on n processors (see [38, 43] for instance) is obviously also of irregularity $O(1)$.

2.3 Granularity and irregularity

The irregularity of one call to the scheduler is the ratio of the number of parallel tasks generated by the number of processors. Consequently, for a given input size, if the number of task creations remains constant for a size-dependent algorithm (following the terminology in [45]), the irregularity may increase when the number p of processors decreases. To make irregularity bounded independently of the number of processors, it is thus necessary to consider a size-independent algorithm such that the number of tasks is related to p.

For instance, let us consider the two previous examples on a p-PRAM with $p \leq n$ processors. The n-point FFT is easily implemented using p parallel tasks, in time $\theta(n \log n/p)$: the irregularity remains $O(1)$. To modify the granularity allows to keep the same irregularity. More generally, for any given size of problem, if the algorithm is static (the execution graph remains the same regardless of indata), using clustering and scheduling techniques it is possible in a compilation phase, to produce a graph for a p tasks parallel algorithm [37]. The irregularity is always $O(1)$. But the efficient list-ranking using the scheduler has irregularity $O(n/p)$ on p processors. The number of tasks generated by the algorithms remains equal to n.

To conclude this section, we may point out that remarks in [13] may also be applied to scheduling: synchronizations should be avoided between computations and scheduling and it is important to distinguish between regular and irregular behaviours of the load-balancer itself. We above all suggest that load-balancing should not be considered separately but as part of the algorithm itself as it is done for communications.

3 Efficiency and irregularity

Dynamic algorithms are defined to be algorithms that execute differently depending on the input. It may be that only the execution graph depends on the indata with the execution time remaining constant, but in a more general context we must consider situations where also the execution time varies. Indeed, for instance for algorithms such as branch-and-bound, used to solve NP-complete problems, a worst-case execution can be efficiently parallelized (in EP). Instead, a main concern from a practical point of view is to ensure that, for fixed entries, the execution will be efficient and in particular to ensure that a good speed-up is obtained with respect to a sequential execution with the same entries. *How much the execution time depends on the indata* will be a measure of irregularity.

3.1 Local and global inefficiency.

In the following, we extend the definitions in [45] to take into account the above comments. For a sequential algorithm $\mathcal{A}$, we denote by $t(x, n)$ its sequential running time on an input x of size $|x| = n$. For a parallel size-independent [40] algorithm $\mathcal{B}$ that solves the same problem, we denote by $t_p(x, n)$ its running time on a p-PRAM with $1 \leq p \leq h(n)$. Thus, t_p is a function of x, n and p, p being a free parameter. As in [42], the parallel work $w_p(x, n)$ denotes the number of operations effectively performed by the algorithm $\mathcal{B}$. We have for all x:

$$w_p(x, |x|) \leq p t_p(x, |x|).$$

Let $t(n) = \max_{x,|x|=n} t(x,n)$ be the worst-case sequential running time and similarly define $t_p(n)$ and $w_p(n)$, the number p of processors being constant for a constant input size n. We are interested in the performances of algorithm $\mathcal{B}$ with respect to the yardstick sequential algorithm $\mathcal{A}$ (which will be when possible the best known sequential algorithm). In this framework, we focus on parallel algorithms that are in EP when worst-case complexity is considered.

To take into account execution time variations, we define the *global inefficiency* $\eta_p(n)$ of $\mathcal{B}$ with respect to $\mathcal{A}$ to be the maximum of the local inefficiency:

$$\eta_p(n) = \max_{x,|x|=n} \frac{w_p(x,n)}{t(x,n)}.$$

Assuming that $\mathcal{B}$ is in EP and that the worst-case is attained, for all n there exist x_0, $|x_0| = n$, such that $t(x_0,n) = t(n) = w_p(x_0,n) = w_p(n)$. If one assume that $t_p(x,n) \leq t(x,n)$ then $\eta_p(n) \leq p$; if $w_p(x,n) \geq t(x,n)$ then $\eta_p(n) = \Omega(1)$.

We have defined the global inefficiency as a maximum, this seems to be reasonable from a practical point of view [2]. For difficult problems or highly dynamic ones, difficulties arise when every particular case has to be handled efficiently. Despite the fact that often no analytic information on the variations of the execution time is available, the minimization of the global inefficiency is related to the scheduling mechanism, nothingly when speculative parallelism is involved. This makes that efficiency relies on frequent calls to a scheduler and thus increases the irregularity of the algorithm (following definition 1).

The main concern is then to build an algorithm that firstly minimizes the global inefficiency and secondly the irregularity in order to limit the use of the scheduler.

3.2 Iterative executions and highly unstructured problems

A similar model of execution than above, that breaks the distinction between static and dynamic approaches, is evoked in [36] for the N-body problem and can be generalized to many iterative solutions. The performances of a static scheduling deteriorates when the solved problem is dynamic, a "static scheduling" can be re-computed dynamically when needed. Examples also include back-tracking search and branch-and-bound optimization [17, 5, 66]: algorithms alternate between computational phases, "expansion phases", where the repartition of the load becomes skewed and load-balancing phases.

In many of these applications the duration of the tasks is not known, this information slackness makes that only heuristics can be developped. An interesting question that is often raised is *how frequent one need to call the load-balancer* (this frequency may vary dynamically). Following the definition, if on p processors the load-balancer is called every $\nu(n)$ units of time on $\#\mathcal{T}(n)$ tasks for an algorithm with parallel time $t_p(n) = w(n)/p$, the irregularity could be defined as:

$$\iota_w(n) = (t_p(n)/\nu(n)) \times (\#\mathcal{T}/p)$$

[2] Other possible choices are to define an *average inefficiency* or to compute the *variance of the inefficiency*. Many strategies tend in fact to reduce these latter quantities.

but, as underlined previously, this is currently of limited interest since it seems difficult to establish such a formula for real highly unstructured problems. But if it is known that scheduling is called on tasks of equal size $s(n)$, *i.e.* on every task at a given level of granularity, then we deduce that $\iota_w(n) = t_p(n)/s(n)$.

3.3 Gaussian elimination

For an input square matrix A of dimension n, we are going to consider parallel algorithms that implement the following one to compute the rank of A.

```
r:=n;
for k, 1..n
   if A(k,k) = 0 then search a row l such that A(l,k) != 0;
   if A(l,k) = 0 then r:=r-1
   else
       swap row k and row l;
       for i, k+1..n if A(i,k) != 0 then zero A(i,k) using row k; endfor;
enfor;
```

The sequential cost is at least $O(n^2)$ (number of tests if A is a regular upper triangular matrix) and $O(n^3)$ in the worst-case. It is obvious to give a p-PRAM algorithm that runs in $O(n^3/p)$ using $p \leq n$ tasks, which is both efficient and polynomially fast following the classification in [45] and having irregularity $O(1)$. This is not satisfying from a practical point of view since such an algorithm may run in the same time $O(n^2)$ in parallel than in sequential for certain matrices in input (*e.g.* consider a row repartition of A and a matrix having diagonal and lower diagonal unity with zeros elsewhere). Its global inefficiency is thus $\eta_p(n) = O(n)$.

Instead, the operations can be scheduled at each phase k of the elimination. One will easily get convinced that a balanced load-distribution is obtained using p tasks at each step. Since n steps are performed this gives a satisfying algorithm with irregularity $O(n)$ and a global inefficiency $\eta_p(n) = O(1)$.

3.4 Array and lists redistribution

The *Task Scheduling Problem* and the *Object Redistribution Problem* as specified in [11] leads to useful and efficient practical implementations. We also refer to [53] for the *Token Distribution Problem* and for other formulations to [14] in the synchronous case and to [7] in the asynchronous case. A plethora of examples of such implementations could be chosen. Among them we find works for ray tracing [27, 51], image processing [49], particles movement simulation [54, 64] or for dictionary machine [15, 18]. These solutions consist in dynamically load-balance tasks in arrays or lists at barrier synchronizations, by solving in particular cases the two former scheduling problems, with irregularity $O(1)$ (the scheduler's job is hand-coded).

If we go back to Gaussian elimination, we have previously obtained a balanced load-distribution by scheduling p new tasks at each elimination step. Instead, we may now use an array redistribution to balance "by hand" the tasks after each elimination step. Such a computation involves an $O(n)$ work overhead at each step, this does not affect the asymptotic cost of the whole algorithm. Moreover, we can write this new algorithm using p tasks, each task performing a sequence of elimination steps, each followed by a redistribution step. The irregularity is now $O(1)$ with a global inefficiency remaining constant.

4 Parallel programmation of irregular algorithms

From the previous sections, some "paradigms" (key observations) for parallel programmation can be given if target algorithms are irregular:

- load-balancing should be considered as part of the algorithm itself,
- there is no major reason that lead to distinguish between static and dynamic scheduling, a mixed approach may be considered,
- the formalization mainly rely on the notion of task,
- attributes such as cost informations should be associated to tasks,
- attributes such as scheduling informations should be associated to tasks,
- even if highly irregular, many applications may execute efficiently with synchronized load-balancing.

These remarks have directed the design of the library *Parallel Algebraic Computing ++*, PAC++, that provides high level facilities to program and execute efficiently *irregular algorithms* on distributed memory machines [33, 34]. The library itself will be briefly described at next section. The main target application of the library is computer algebra [30, 58, 35], the example we have chosen to illustrate the PAC++ programming model is taken from this field. However, as it will be shown, the way PAC++ takes advantage of underlying automatisms such as *static scheduling* or *load-balancing*, can be used in various other areas involving algorithms which behaviours at execution cannot be statically predicted.

We now focus on main concepts only to provide a "high-level" description of the model which aim is to free the user from task scheduling. We rely in part on preliminary studies in [59, 60, 69] for the notions of *cost prediction informations* and of *poly-algorithm* and on [55, 10] for the run-time support ATHAPASCAN of PAC++ (this support provides a *fork/join* mechanism of *threads* and a load-balancer). The main objective is to write programs such that a description of the precedence graph can be easily computed (*e.g.* by a symbolic execution or by detecting static parts [47]) and scheduled (statically or during execution), or such that tasks are well specified so that they can be handled by a load-balancer [32]. Our purpose is not to give a model of a *scheduler* and of a *load-balancer*, but to see how they can be easily interfaced in a common framework.

4.1 Overview

We consider that an execution of a parallel algorithm is a set $\mathcal{T}$ of tasks (a task will be a function) that execute either sequentially or simultaneously on n

processors. Such an execution can be represented as a dependence directed graph, an *execution graph*, which vertices are sequential tasks and which arcs indicate precedence constraints between these tasks. If the graph does not depend on the values of the entries of the algorithm, it is viewed as a representation of the algorithm itself.

Given a set $\mathcal{T}$, if we assume that no task migration is possible, the problem of executing the graph on n processors (of determining completely a parallel execution), reduces to specify a scheduling for each task of $\mathcal{T}$ (see section 2.1):

- *a date of execution* that indicates when the task will execute, this date can be for instance an absolute time if a clock is available or a relative time (*e.g.* task T_j will execute when T_i has completed),

- *a site of execution* that gives the processor that will execute the task.

Once the date and the site have been fixed we will say that the task has been *scheduled* for parallel execution, the couple (date, site) will be called a *schedule* of the task.

Various strategies can be used to schedule the tasks on a target parallel machine. But obviously the strategy that can be used heavily depends on the knowledge available on $\mathcal{T}$ before execution. A static scheduling can be used if the graph and the costs of the tasks are known. Conversely, any decision concerning the date and the site will be taken during execution by a dynamic load-balancer if no information is available before execution. Anyway, whatever the stategy that is used, the key objects are the tasks and their schedules. This appears directly in the model of programmation we propose, as explained at section 4.2 below, *functions* and *algorithms* will play the role of tasks. Since the notions naturally extend to graphs of tasks [7], we construct and use *weighted graphs* at section 4.3. Once these are defined, the job of either a static scheduler or a load-balancer is to assign values to schedules of functions, algorithms or graphs. In addition, a load-balancer also take structural decisions concerning the execution graph. Depending on the load of the machine and depending on the indata, a load-balancer will have to choose between several algorithms to solve the same problem, the one that is currently the best [65]. It may also indicate if a problem has to be splitted (and in how many parts) or not. These aspects will be developped at section 4.3.

The programming model is based on a C-like programming language: a sequential program is a function that may recursively call other functions. We assume the parallel machine to be a set of n processors that work simultaneously. Since programs will reduce to n-ary *Remote Procedure Calls* each processor is viewed as a computational *server* and is associated to a unique identification P_i, $1 \leq i \leq n$. Each server is able to execute a given set of functions, this set may vary from a server to another. A server has its own memory that can be addressed by all the functions it executes. There is no global memory: a function cannot access the memory of a server but the one it executes on.

4.2 Functions and algorithms

Any function f executing on a given server can ask for the execution of another function g on another server. In a simple sequential framework such a request would be mainly characterized by the actual values of the arguments of g. From a parallel point of view, independently of the function and of its arguments a request will also be associated to a *site* and a *date*.

These informations will be given by a description d of the request on g: a description of the actual execution of g. As said previously, such a description will be called a *schedule* of g. Once a schedule d has been updated to give relevant informations it can be used to start a request `Y:=g(X)` where X stands for values of the arguments of g and Y for the corresponding returned values. We use `call` as usually to manipulate threads in the following way:

- $Y := d.\texttt{call}(g(X))$, the function g will execute as specified by d, the result is assigned to Y.

Parallelism between such calls will be generated by n-ary calls at next section.

Clearly for these calls, at least the site has to be known. We will see later how the updating of d will be let to static schedulers and dynamic load-balancers, but we can notice that, using the statically known identifications P_i of the servers to assign the site indicated by d, this yet gives us a standard model based on *Remote Procedure Call*, RPC.

To reach the notion of task, to a function must be associated characteristics understandable by the mechanism that will handle the tasks to schedule them. Further, we can assume that such a mechanism will take decisions considering only these informations *i.e.* without considering the function itself. For instance, using a *Unit Execution Time* model, graphs are scheduled independently of the operations that tasks actually realize. Consequently, to couple together informations and functions appears to be a key point. We introduce for that the notion of *algorithm*.

An algorithm is a couple formed by a function g and an information concerning this function. To simplify, we will assume that this information is a cost information and is a function C_g of the arguments of g, the values returned by such a function will be used as inputs of schedulers and load-balancers. These include *static informations i.e.* that do not depend on the values of the arguments of g (C_g is a constant function): for instance resource requirements (*e.g.* the subset of the servers that can execute g) or a static cost (all basic functions in a *Unit Execution Time* model).

More generally, the cost of a function may not be known as a constant value but may depend on its arguments. In this case C_g may be a function that gives a cost *a priori* of the algorithm given by g in terms of the sizes of the arguments [59, 60]. If X is a value of the arguments of g, we will denote by $|X|$ its size then $C_g(|X|)$ is the algorithmical cost of the computation of $g(X)$. For instance, to formulate that the cost of a matrix product is $O(n^3)$, is an information that can be given statically and which can be relevant at execution and exploited automatically as soon as n is known. We will say that such an information is *quasi-dynamic*: it depends on the sizes of the arguments but not on their values.

The updating of the static and quasi-dynamic informations will partly rely on the user; it may also be done by a symbolic execution.

Conversely, when an algorithm leads to highly dynamic executions, if no static or even quasi-dynamic information is known, the cost of the algorithm may be updated only by the load-balancer (for instance following statitics), in this case we say the information to be *dynamic*.

From now we will denote by $G = [g, C_g]$ the algorithm defined by functions g and C_g. In the same way we have associated to a function a description of a request on it, we associate to an algorithm G and values X of the arguments of g, a schedule D of a corresponding execution. Once D is updated, the algorithm can be manipulated as follows:

- $Y := D.\mathtt{call}(G(X))$, the function g will execute as specified by D.

We will refer to the information function C_g concerning G by:

- $G.\mathtt{cost}(|X|)$, returns $C_g(|X|)$.

The notion of algorithm corresponds to a task and its cost, we now construct graphs.

4.3 Scheduling and load-balancing

As specified above either a function, an algorithm and we will see, a graph of algorithms have their executions described by a schedule. This schedule can be assigned "by hand" by the users for instance if a description of the machine is known. In the general case this can be done automatically. A scheduler will take in input a static graph of algorithms. A load-balancer will manage expressions involving dynamic choices. In both cases they will be invoked using the instruction `schedule` (from the p-PRAM model at section 2.2).

For a graph of algorithms $\mathcal{G} = \{G_1, \ldots, G_k\}$ (each with input X_k), provided a corresponding weighted graph $\mathcal{C}|\mathcal{X}| = \{C_1(|X_1|), \ldots, C_k(|X_k|)\}$ of informations is known, a static scheduling can be computed. The execution will be directed by a graph of schedules $\mathcal{D} = \{D_1, \ldots, D_k\}$ that can be initialized using the instruction `schedule`:

- $\mathcal{D}.\mathtt{schedule}(\mathcal{C}|\mathcal{X}|)$, computes a scheduling and assigns it to $\mathcal{D}$.

If we denote by $\mathcal{X}$ the inputs of the input nodes of $\mathcal{G}$ and by $\mathcal{Y}$ the outputs of the output nodes, then execution can be started using:

- $\mathcal{Y} := \mathcal{D}.\mathtt{call}(\mathcal{G}(\mathcal{X}))$, executes the graph $\mathcal{G}$ using sites and dates given by $\mathcal{D}$.

We can see that this reduces to `call` for functions if the graph $\mathcal{G}$ is simply a function g with no informations attached to it and $\mathcal{D}$ is a schedule d of g. Thus, before calling a function g, the site can be chosen automatically using $d.\mathtt{schedule}()$. In the same way, an n-ary call of functions corresponds to a special case of graphs, for any fixed k the simultaneous execution of k functions may be specified as follows:

- $\mathcal{Y} := \mathcal{D}.\mathtt{call}\ (g_1(X_1) \wedge \ldots \wedge g_k(X_k))$.

In the above we have asked for the execution graph to be known. We will see at section 4.4 that this can be overcomed in some cases when the graph is not known: an interpretation of the program can generate the graph for the scheduler.

When a static approach is not possible, a lack of `schedule` instructions will make decisions fall into the load-balancer hands. As underlined previously, another goal is to offer alternatives. We consider for that the two basic operations with choices *or* and *split*.

Given two algorithms G_1 and G_2 to solve the same problem, depending on the current state of the machine when the solution of the problem is needed, the best choice (the cheapest one for instance) may be either G_1 *or* G_2. For input costs $C_i(|X|)$, $1 \leq i \leq k$, we propose an *or* operator on the algorithms. Choices are made using the instruction `choose`:

- G.`choose`$(G_1 \vee \ldots \vee G_k, |X|)$, assigns G to one of the G_i.

Then, as previously seen, we can schedule G and execute it. This implements the notion of *poly-algorithm*.

In the same way, let G be an algorithm to be executed with X in input and let the potential degree of parallelism that can be generated by splitting G be described by a set K of integers. More precisely, we assume that for any integer k chosen among l values in $K = \{k_1, \ldots, k_l\}$ executing $g(k, X)$ (g is the function given by G) consists of splitting X into $X(1), \ldots, X(k)$, next of simultaneously executing algorithms $\bar{G}(X(1)), \ldots, \bar{G}(X(k))$ and finally of merging the results $Y_1, \ldots, Y_k$ to recover $Y = g(X)$, then the choice of the best value k is let to the load-balancer using:

- G'.`choose`$(\wedge_K(G), |X|)$, assigns G' to an algorithm which consists in executing G with the chosen value k for the splitting.

The cost information $C(|X|)$ of G may indicate the overhead for the splitting and the costs of the sub-algorithm $\bar{G}$. Once the choice is made, we are led to the schedule and the execution of G'.

The principle that has been applied for any object is firstly to initialize a schedule of the object then to start the execution following indications of sites and dates given by this schedule. These phases can be done automatically once the user has described a graph or has written algorithms. Further, as briefly discussed below, mixed static/dynamic scheduling can be used by automatically constructing, at least partially, the execution graph

As announced, a preliminary version of a library using these concepts has been implemented under C++. This will be presented at section 5 where C++ classes lead to implementations of `Algorithms`, of `Schedule` representing the schedules and of `CostInfo` representing the cost informations. Further developments should be concerned with sequences of algorithms. Indeed, in the above we are limited to situations where cost information is related to the indata of a given algorithm. In general cases an algorithm may also give relevant informations on its outdata, that can be subsequently used as input cost of other algorithms.

4.4 Interpretation and execution

Once a program involving algorithms is written, it can be executed in a way mixing static and dynamic scheduling. A description of the execution graph may be obtained via a partial evaluation (symbolic unfolding) of the program. Two extreme cases may be distinguished. On the one hand, if the program is static, the obtained graph describes the whole execution, a unique schedule has to be computed. This schedule may be computed statically. On the other hand, when the program countains branching (resp. indirect accesses) where the conditions (resp. adresses) cannot be decided form the sole knowledge of the indata, the program is referred as *dynamic* [47]. In this case, a symbolic execution (also partial evaluation [47]) of the whole input program, generates a set of possible execution graphs, only one of them being valid for a given indata. In order to provide a valid graph, the execution of the program is dynamically splitted into successive static parallel steps. Each step firstly consists in building the graph description of the corresponding computations, then the scheduling of the graph is proceeded and the execution started.

At the highest level, the partial evaluation of a general program allows to build a graph, which nodes represent either elementary tasks, either nested subgraphs which will be dynamically built before their scheduling. Following section 2, the irregularity of the program, related to the number of calls to the scheduler, here appears as a consequence of the dynamic behaviour of the program.

5 An application: Parallel Algebraic Computing ++

To illustrate this section we have chosen a central problem in computer algebra: the *manipulation of algebraic numbers*. Let us look at an example. Following [50] the matrix

$$A = \begin{bmatrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{bmatrix}$$

can be brought into Jordan normal form [28]:

$$J = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Now how does J vary under a small perturbation A_ϵ of A? We take

$$A_\epsilon = \begin{bmatrix} 130\,\epsilon - 149 & -50 - 390\,\epsilon & -154 \\ 537 + 43\,\epsilon & 180 - 129\,\epsilon & 546 \\ 133\,\epsilon - 27 & -9 - 399\,\epsilon & -25 \end{bmatrix} \approx \begin{bmatrix} -148.9999 & -50.0003 & -154 \\ 537.0000 & 179.9999 & 546 \\ -26.9999 & -9.0003 & -25 \end{bmatrix}$$

where $\epsilon \approx 0.000000784$ is such that

$$4 - 5910096\,\epsilon + 1403772863224\,\epsilon^2 - 477857003880091920\,\epsilon^3 + 242563185060\,\epsilon^4 = 0.$$

In this case it should be computed that two distinct eigenvalues have collapsed to a double eigenvalue with one unique eigenvector [50]. Such a result can be obtained by means of computer algebra provided one can manipulate ϵ as an algebraic number *i.e.* as a root of the above polynomial. This an example of application of computations with algebraic numbers. After a short presentation of the problem we show below how its directly enters the scope of PAC++ as an application of *algorithms* and *cost information* for irregular algorithms.

5.1 Computations on algebraic numbers

An algebraic number over a field F is a root of a polynomial over F [41]. As proposed in [16], a convenient way to manipulate these numbers with a computer is precisely to represent them by polynomials whom they are roots. For instance $\sqrt{2}$ can be represented by $\lambda^2 - 2 = 0$. Now consider the problem of triangularizing the following matrix by a Gaussian elimination:

$$A(\mu) = \begin{bmatrix} (\mu^2+1)(\mu-3) & 1 \\ 1 & 0 \end{bmatrix}$$

where μ is an algebraic number such that $\chi(\mu) = (\mu^2+1)(\mu-1) = 0$. The computation leads to a "discussion": the first entry of the matrix is zero if μ is such that $\mu^2+1=0$ and is nonzero if $\mu-1=0$. These two cases may lead to two different upper triangular matrices:

$$T_1 = \begin{bmatrix} (\mu^2+1)(\mu-3) & 1 \\ 0 & 1/((\mu^2+1)(\mu-3)) \end{bmatrix}$$

by zeroing the entry $A_{2,1}$ with the nonzero pivot $A_{1,1}$ if $\mu-1=0$ ($\mu^2+1 \neq 0$); or to:

$$T_2 = \begin{bmatrix} 1 & (\mu^2+1)(\mu-3) \\ 0 & 1 \end{bmatrix}$$

after a column swap since the pivot is zero if $\mu^2+1=0$ ($\mu-1 \neq 0$). Thus for a $n \times n$ matrix $A(\mu)$ with μ being the root of a polynomial $\chi(\mu)$ of degree n, the elimination may produce new branches each time an equality to zero has to be tested for a pivot. A step k of the elimination, a test for pivoting may *split* the computation into two sub-computations that consist in continuing the elimination on two sub-matrices of dimension $n-k$ with respectively two polynomials $\chi_{k1}(\mu)$ and $\chi_{k2}(\mu)$ (divisors of the initial polynomial $\chi(\mu)$). These two branches of computations can be handled simultaneously and thus provide rough-grain parallelism:

- two tasks are created that have to be scheduled dynamically,

- two relevant cost information should be taken into account: the new tasks consist in triangularizing matrices of *dimension* $n-k$, the polynomials defining the algebraic numbers are of *new degrees* d_{k1} and d_{k2}.

Since it cannot be determined in advance when splittings will occur, we see that algorithms using algebraic numbers in this way will be irregular. From a theoretical point of view, manipulating algebraic numbers and algorithms involving algebraic numbers may well be classified in complexity classes like NC [61, 62] (worst-case studies). But we think that efficient parallel algorithms will strongly rely on dynamic scheduling as it is presented below. Notice that if we refer to section 2.2, the triangularization above may lead to at most n splittings since the degree of the polynomial $\chi(\mu)$ in input is n. In the worst-case these splittings may occur sequentially and give a worst-case irregularity $\iota_w(n) = O(n)$.

5.2 PAC++: Basic classes for parallel computation

We now describe an implementation of some of the concepts introduced at section 4. In two parts, we begin with the implementation of basic objects. Then at section 5.3 we focus on main features needed for parallel handling of algebraic numbers. For further insights the reader will refer to [33, 34, 31].

As shown with the examples above, data strutures and execution times may vary a lot: most of computer algebra algorithms are *irregular* but also provides *quasi-dynamic cost informations* (as defined at section 4.2). To take into account this irregularity and these informations, a set of C++ classes is offer to the user for developping portable programs that can be efficiently executed on different machines.

5.2.1 Remote calls of functions. A parallel program in PAC++ is written using a set of virtual processors mapped at execution on physical processors. Each processor is a server for algebraic computations: can execute a prescribed set of functions given by a library. This set is extended by the user's defined functions. The parallelism of an application is expressed through asynchronous or synchronous remote calls to these functions. The call of a function on a processor is executed by a thread using the above cited runtime support ATHAPASCAN [55, 10]. ATHAPASCAN is a parallel extension of C which includes an adaptative granularity scheme and static/dynamic scheduling mechanisms.

A function that can be remotely called is called an *entrypoint*. This is implemented via the `EntryPoint` class of objects. To execute a remote call, the arguments of the function are bufferized in an object of type `iBuffer` (*input Buffer*). When the call is completed, the result is got out an object of type `oBuffer` (*output Buffer*). Thus an entrypoint has the following prototype:

```
void MyEntryPointFunction (iBuffer& , oBuffer& ) ;
```

5.2.2 Algorithms. An *algorithm* is an entrypoint plus quasi-dynamic cost informations. For most of parallel algebraic algorithms, the arithmetic and communication costs are known at certain levels of granularity [59, 60]. In our current implementation a cost information is couple formed by these costs:

```
class CostInfo {
public:
   // CostInfo cstor :
   CostInfo ( double ArithCost, double CommCost ) ;
} ;
```

All algorithms in PAC++ are objects rather than functions. They have in common the following main features:

(i) informations to instanciate the computation for given input data depending on the algorithm;

(ii) quasi-dynamic cost informations;

(iii) a main function giving the task to perform;

(iv) temporary data on which the algorithm works;

(v) output data.

Only features (ii) and (iii) are implemented as virtual member functions of the basic class `Algorithm` from which derive all other algorithms. Since data structures depend on the algorithm, four functions for *packing* (to put into an `iBuffer`) and *unpacking* (to get out an `oBuffer`) are purely virtual:

```
class Algorithm {
public:
  virtual void main( ) = 0 ;
  virtual CostInfo cost(  ) const = 0 ;
  virtual packargs ( oBuffer& ) = 0 ;
  virtual packres   ( oBuffer& ) = 0 ;
  virtual unpackargs ( iBuffer& ) = 0 ;
  virtual unpackres  ( iBuffer& ) = 0 ;
} ;
```

Two global operators are defined over algorithms that provide structured calls to parallel sub-algorithms:

- `AND`, indicates that several algorithms will be executed simultaneously;
- `OR`, indicates that a choice (which depends only on the cost information and on the load of the machine) will be made between several algorithms.

In the current PAC++ prototype of an interface for handling irregular algorithms no other operator (*e.g.* sequence, composition) is offered. Only expressions involving `AND` and `OR` operators are valid.

5.2.3 Schedules. The schedules of functions and algorithms are implemented via the `Schedule` class. The member functions on an object of this type are `spawn`, `wait` and `call`. In addition, any object of the class must be initialized before used using the `schedule` member function. An outline of interface of the class `Schedule` is as following:

```
class Schedule {
public:
  // Initialization :
```

```
  void schedule ( const Algorithm& G ) ;
  // Asynchronous remote call
  Schedule& spawn( Algorithm& G ) ;
  // Synchronization
  void wait( Instance& Result ) ;
  // Synchronous call
  Schedule& call ( Algorithm& G ) ;
} ;
```

The implementation of this class relies on the tools available. The `schedule` function makes use of underlying static and dynamic schedulers. The three other functions are written upon the runtime support ATHAPASCAN.

5.2.4 Execution of programs.

From the user point of view the different phases to initialize and start an execution should be gathered. Thus an important function is

```
Algorithm& Execute ( const Algorithm& G ) ;
```

This function executes an expression of algorithms and returns the result as an other expression of algorithms. The choices between algorithms, the initialization of the schedules and the execution itself are automatically performed.

Let us look at an example. We derive an algorithm to compute the rank of a square matrix from the basic class `Algorithm`. In particular, the virtual member functions are re-defined. The algorithm takes in input a matrix and returns an integer:

```
class RankAlgorithm : public Algorithm {
public:
  // Ctsor and initialization
  RankAlgorithm ( const Matrix& inputMatrix ) ;

  // Return the cost information (arithmetic and communication)
  CostInfo cost(  )
  { int n = indata.rowdim() ;
     return CostInfo( n*n*n,  n*n ) } ;

  // The main function
  void main(  ) ;

  // Packing and unpacking of argument and result
  packargs ( oBuffer& B ) {B << indata } ;
  packres   ( oBuffer& B ) {B << outdata } ;
  unpackargs ( iBuffer& B ) { B >> indata } ;
  unpackres  ( iBuffer& B ) { B >> outdata } ;

protected:
```

```
  Matrix A ;  // input: a matrix
  int Step ;  // internal variable: the current step
public:
  int rank ;  // output: the (current computed) rank of the matrix
} ;
```

To execute such an algorithm for computing the rank of a given matrix M we proceed as follows. Firstly, an object `Rank` is declared and instanciated with the indata `M` (a matrix). Then, the above function `execute` is called:

```
Matrix M ;
// here some initializations of M
RankAlgorithm Rank ( M ) ;
Execute( Rank ) ;
if (Rank.outdata ==1) { .... }
else {....}
```

5.3 Parallel handling of algebraic numbers splitting

As described in introduction at section 5.1, in the example of the computation of the rank of a matrix $A(\mu)$ which entries are algebraic numbers, splittings during Gaussian elimination may occur when a nonzero pivot has to be chosen. If we denote by `Rank` the previous algorithm, then on a matrix $A(\mu)$ several values should be returned: each time a splitting occurs the number of returned values may be incremented by one. If $A(\mu)$ is given by

$$A(\mu) = \begin{bmatrix} 1 & 0 \\ 0 & \mu^2 - 1 \end{bmatrix}$$

where μ is an input algebraic number defined by $\chi(\mu) = (\mu^2+1)(\mu^2-1) = 0$, then the rank will be either equal to 2 if $\mu^2+1=0$ or equal to 1 if $\mu^2-1=0$.

Anyway, one may want to get *one of the possible results* or *all the results*. This choice is let to the responsability of the user. PAC++ offers two main managers of computation for this purpose: the computation with one root (arbitrarily chosen) and with all roots [19]. In this latter case, the whole computation (*e.g* the rank computation) is embedded in a manager which creates new parallel threads whenever a new splitting occurs. The completion is ensured by a synchronization barrier to recover all the results of the different parallel computations (the `AND` of algorithms provides such a synchronization).

In addition to the previously mentioned facilities, the implementation requires to spawn functions following the *fork* instruction of the PRAM model [26, 4]. Indeed, the *context* of the spawned function, *i.e.* the data needed to the computation following the splitting, must be recopied and associated to the spawned executions of the sub-algorithms. In the example of the rank, this context is essentially the sub-matrix that remains to eliminate after a nonzero pivot is found.

This context highly depends on the algorithm. In the current version of the library, this context is identified by the user-defined `restart` member function to *restart* the computation after a splitting. For such a type of algorithms, the class `ForkAlgorithm` is derived from the basic class `Algorithm`:

```
class ForkAlgorithm : public Algorithm {
public:
  virtual ForkAlgorithm* getstate() const = 0 ;
  virtual void restart() = 0 ;
} ;
```

When a splitting happens, the manager creates a copy of the current algorithm by calling the `getstate` member function and calls the `restart` function to continue the computation (on the same site or on another site depending on the scheduler answer).

On the rank example, the class `ForkRankAlgorithm` over algebraic numbers may be derived and implemented from the `RankAlgorithm`:

```
class ForkRankAlgorithm :
    public ForkAlgorithm, public RankAlgorithm
{
public:
    // Same ctsor than for RankAlgorithm :
    ForkRankAlgorithm ( const Matrix& M ) : RankAlgorithm (M) {} ;

    // getstate returns a copy at step k of the (n-k)x(n-k)
    // sub-matrix and save the current value of the rank
    ForkAlgorithm* getstate() const ;

    // restart the computation on a sub-matrix, at the end of
    // the computation, the returned rank if the one of the
    // input nxn matrix.
    void restart() ;

protected:
 int saved_rank ;
}
```

The example we have chosen demonstrates the use of algorithms and of cost informations for linear algebra. Notice that clearly, a cost information need not be complexity in terms of the input's size: a "cost" may be a priority, a statistical cost or any other relevant information on the computation. Together with schedules, this can implemented and used for a wide range of applications.

6 Conclusion

Inspired by the notion of locality we have proposed a definition of irregularity based on a scheduling complexity. By this preliminary study on the subject we want to emphasize that:

- routing and scheduling present many similar aspects and lead to somehow dual problems,

- scheduling should be considered as part of the algorithm itself.

This last point implies that no difference should be made between static or dynamic scheduling. In addition, while complexity definitions are usually given in a worst-case context, we believe that this is unappropriated to many irregular algorithms and other directions exist. Another way to define irregularity could have been to use task graphs. Intuitively, the irregularity of a nonstatic algorithm is related to the number of graphs corresponding to the possible executions and to the difficulty to compute these graphs. It seems harder to derive a satisfying definition from these aspects. This would need analysis tools and ways of comparing graphs and is an interesting direction for further studies.

References

1. A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAM's. *Theoretical Computer Science*, 71:3–28, 1990.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques and tools.* Addison-Wesley, 1986.
3. S. Aluru and J. Gustafson. Subtle issues of SIMD tree search. In *Parallel Computing: Trends and Applications, Proceedings of PARCO'93, Grenoble France*, pages 49–56. Elsevier Science, 1994.
4. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II.* Springer-Verlag, 1990.
5. Benaïchouche, M. Résolution parallèle de l'Affectation Quadratique (QAP) et de la Couverture Minimale d'un Graphe (VCP) par la méthode Branch & Bound. In *Proc. of FRANCORO, Rencontres Francophones de Recherche Opérationnelle*, 1995.
6. G. Bernard, D. Steve, and M. Simatic. Placement et migration de processus dans les systèmes reépartis faiblement couplés. *Technique et Science Informatiques*, 10(5):375–392, 1991.
7. D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and distributed computation: numerical methods.* Prentice-Hall International, 1989.
8. Powley. C., C. Ferguson, and R. Korf. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, 60, 1993.
9. T.L. Casavant and J.G. Khul. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, (14):141–154, 1988.
10. M. Christaller. Athapsacan-0a sur PVM.3. Technical Report APACHE 11, IMAG Grenoble France, 1994.
11. 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. Comput.*, 17(1), 1988.
12. S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64:2–22, 1985.
13. 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.

14. G. Cybenko. Dynamic load-balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2), 1989.
15. F. Dehne and M. Gastaldo. A note on the load-balancing problem for coarse grained hypercube dictionary machines. *Parallel Computing*, 16, 1990.
16. J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL'85*, LNCS 204, Springer Verlag, pages 289–290, 1985.
17. S. Dowaji and C. Roucairol. Influence of priority of tasks on load-balancing strategies for distributed branch-and-bound algorithms. In *Proc. of IPPS'95, Workshop on Solving Irregular Problems on Distributed Memory Machines, Santa Barbara, USA*, 1994.
18. T. Duboux, A. Ferreira, and M. Gastaldo. MIMD dictionary machine: from theory to practice. In *CONPAR 92, Lyon, France*, LNCS 634, September 1992.
19. D. Duval. *Diverses questions relatives au calcul formel avec des nombres algébriques.* Thèse de Doctorat d'Etat, Université de Grenoble, France, 1987.
20. H. El-Rewini and T.G. Lewis. *Introduction to Parallel Computing.* Springer-Verlag, 1990.
21. G. Authié et al. *Algorithmes parallèles, analyse et conception I.* Hermès, 1994.
22. G. Authié et al. *Algorithmes parallèles, analyse et conception II.* To appear, 1995.
23. Cung V.D. *et al.* Concurrent data structures and load-balancing strategies for parallel branch & bound/A* algorithms. In *Third Annual Implementation Chalenge Workshop, DIMACS, New-Brunswick, USA*, 1991.
24. 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.
25. C. Fonlupt. *Distribution dynamique de données sur machines SIMD.* PhD thesis, Université de Lille 1, France, 1994.
26. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.
27. A. Fujimoto, T. Tanaka, and K. Iwata. Accelerated ray tracing system. *IEEE Comp. Graph. and App.*, Apr. 1986.
28. F.R. Gantmacher. *Théorie des matrices.* Dunod, Paris, France, 1966.
29. M. Gastaldo. *Contribution a l'algorithmique parallèle des structures de données et des structures discrètes : machine dictionnaire et algorithmes pour les graphes.* PhD thesis, ENS Lyon et UCB Lyon I, France, Dec. 1993.
30. J. von zur Gathen. Parallel arithmetic computations: a survey. In *Proc. 12th Int. Symp. Math. Found. Comput. Sci.*, pages 93–112. LNCS 233, Springer Verlag, 1986.
31. T. Gautier. PAC++: presentation and experiments. In *International Symposium on Symbolic and Algebraic Computation, Montreal, Canada - Poster session*, July 1995.
32. T. Gautier, F. Guinand, J.L. Roch, and A. Vermeerbergen. Régulation de charge et adaptation de grain : Athapascan. Preprint IMAG Grenoble France, (Journées de Recherche sur le Placement Dynamique et la Régulation de Charge, GDR PRS, Mai 1995).
33. T. Gautier and J.L. Roch. PAC++ system and parallel algebraic numbers computation. In *PASCO'94, Hagenberg/Linz Austria, Sept. 1994.*
34. T. Gautier, J.L. Roch, and G. Villard. PAC++ v2.0: user and developer guide. Technical Report APACHE 14, IMAG Grenoble France, 1994.

35. K.O. Geddes, R. Czapor, and G. Labahn. *Algorithms for computer algebra.* Kluwer Academic Press, 1992.
36. A. Gerasoulis, J. Jiao, and T. Yang. *Scheduling of structured and unstructured computation.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, to appear, 1995.
37. A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAG's on multiprocessors. *J. Par. Distr. Comp.*, Dec. 1992.
38. A. Gibbons and W. Rytter. *Efficient parallel algrithms.* Cambridge University Press, 1988.
39. P.B. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 1989.
40. A. Gottlieb and C.P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *J. ACM*, 31:193–209, 1984.
41. N. Jacobson. *Basic Algebra I.* W.H. Freeman and Company, 1974.
42. J. Jájá. *An introduction to parallel Algorithms.* Addison-Wesley, 1992.
43. R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 869–941. North-Holland, 1990.
44. C.P. Kruskal, T. Madej, and L. Rudolph. Parallel prefix on fully connected direct connection machine. In *Proc. Int. Conf. on Parallel Processing, Illinois USA*, 1986.
45. C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, (71):95–132, 1990.
46. T.H. Lai and S. Sahni. Anomialies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.
47. B. Lisper. Detecting static algorithms by partial evaluation. In *Proc. ACM Sigplan Symposium on Partial Evaluation and Semi-Based Program Manipulation*, 1991.
48. B. Mans and C. Roucairol. Parallel branch & bound for discrete optimization problems. In *Workshop on Parallel Processing of Discret Optimization Problems, Mineapolis*, 1991.
49. S. Miguet and Y. Robert. Elastic load-balancing for image processing algorithms. In Parallel Computation H.P. Zima, editor, *1rst International ACPC Conference, Salzburg, Austria*, 1991.
50. C. Moler, 1993. Communication about Mathlab test example *gallery(3)*.
51. K. Nemoto and T. Omachi. An adaptative subdivision by sliding boundary surfaces for fast ray tracing. *Graphics Interface*, 1986.
52. M. Norman and P. Thanish. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, sep. 1993.
53. D. Peleg and E. Upfal. The token distribution problem. In *Proc. of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 418–427, 1986.
54. J.M. Pierson. A dynamic parallel implementation of a physically based particles models. In G. Hégron and O. Fahlander, editors, *Fifth Eurographics Workshop on Animation and Simulation, Oslo Norway*, 1994.
55. B. Plateau and al. Présentation d'APACHE. Technical Report APACHE 1, IMAG Grenoble France, 1993.
56. A. Ranade. A framework for analyzing locality and portability issues in parallel computing. In *Parallel Architectures and their Efficient Use*, LNCS 678, pages 185–194, 1993.
57. V.N. Rao and V. Kumar. Superlinear speed-up in ordered depth-first search. In *Proc. 6th Distributed Memory Computing Conference*, 1991.

58. J.L. Roch, F. Siebert, P. Sénéchaud, and G. Villard. Computer Algebra on a MIMD machine. *ISSAC'88, LNCS 358 and in SIGSAM Bulletin, ACM*, 23/11, p.16-32, 1989.
59. J.L. Roch, A. Vermeerbergen, and G. Villard. Cost prediction for load-balancing: application to algebraic computations. In *CONPAR 92, Lyon, France*, LNCS 634, September 1992.
60. J.L. Roch, A. Vermeerbergen, and G. Villard. A new load-prediction scheme based on algorithmic cost functions. In *CONPAR 94, Linz Austria*, LNCS 854, Sep. 1994.
61. J.L. Roch and G. Villard. Fast parallel computation of the Jordan normal form of matrices. *Parallel Processing Letters*. To appear.
62. J.L. Roch and G. Villard. Parallel computations with algebraic numbers, a case study: Jordan normal form of matrices. In *Parallel Architectures and Languages Europe 94, Athens Greece*, LNCS 817, July 1994.
63. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.
64. M. Smith and E. Renshaw. Parallel-prefix remapping for efficient data-parallel implementation of unbalanced simulations. In *Parallel Computing: Trends and Applications, Proceedings of PARCO'93, Grenoble France*, pages 215–222. Elsevier Science, 1994.
65. M. Snir. Scalable parallel computers and scalable parallel codes: from theory to practice. In *Parallel Architectures and their Efficient Use*, LNCS 678, pages 176–184, 1993.
66. R. Subramanian and I.D. Scherson. An analysis of diffusive load-balancing, 1995. Preprint - University of California, Irvine, USA.
67. L. Valiant. A bridging model for parallel computation. *Communication ACM*, 33:103–111, 1990.
68. L. Valiant. General purpose parallel architectures. In J. van Leuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 944–971. North-Holland, 1990.
69. A. Vermeerbergen. Les poly-algorithmes et la prévision de coûts pour une expression portable et extensible du parallélisme. In L. Bougé, editor, *Actes de RenPar'6, ENS Lyon, France*, pages 51–54, 1994.
70. G. Villard. Parallel general solution of rational linear systems using p-adic expansions. In *IFIP WG 10.3 Working Conference on Parallel Processing, Pisa Italy*, 1988.
71. M.-Y. Wu and D. D. Gajski. Hypertool: a programming aid for message-passing systems. *IEEE Trans. Soft. Eng.*, 1(3):330–343, 1990.
72. S. Zhou. A trace-driven simulation study of dynamic load-balancing. *IEEE Trans. on Software Engineering*, 14(9), 1988.