# Feasability, Portability, Predictability and Efficiency : Four Ambitious Goals for the Design and Implementation of Parallel Coarse Grained Graph Algorithms

Isabelle Guérin Lassous, Jens Gustedt, Michel Morvan

# *Feasability, Portability, Predictability and Efficiency: Four Ambitious Goals for the Design and Implementation of Parallel Coarse Grained Graph Algorithms*

Isabelle Guérin Lassous, Jens Gustedt, Michel Morvan

**N°3885**

Février 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3885--FR+ENG

# Feasability, Portability, Predictability and Efficiency: Four Ambitious Goals for the Design and Implementation of Parallel Coarse Grained Graph Algorithms

Isabelle Guérin Lassous[*], Jens Gustedt[†], Michel Morvan[‡]

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n˚3885 — Février 2000 — 34 pages

**Abstract:** We study the relationship between the design and analysis of graph algorithms in the coarsed grained parallel models and the behavior of the resulting code on todays parallel machines and clusters. We conclude that the coarse grained multicomputer model (CGM) is well suited to design competitive algorithms, and that it is thereby now possible to aim to develop portable, predictable and efficient parallel algorithms code for graph problems.

**Key-words:** parallel algorithms, graph algorithms, coarse grained models, PC clusters

*(Résumé : tsvp)*

[*] INRIA Rocquencourt, F-78153 Le Chesnay, France. Email: `Isabelle.Guerin-Lassous@inria.fr`.
[†] LORIA and INRIA Lorraine, BP 239, F-54506 Vandœuvre-lès-Nancy, France. Email: `Jens.Gustedt@loria.fr`.
[‡] LIAFA and Université Paris 7 and Institut Universitaire de France, Case 7014, F-75251 Paris Cedex 05, France. Email: `morvan@iafa.jussieu.fr`.

# Faisabilité, Portabilité, Prédictibilité et Efficacité : Quatre Buts Ambitieux pour la Conception et l'Implentation d'Algorithmes Parallèles à Gros Grain pour les Graphes

**Résumé :** Nous étudions les liens entre, la conception et l'analyse d'algorithmes pour les graphes dans les modèles parallèles à gros grain, et le comportement du code résultant sur les machines parallèles actuelles et les grappes. Nous arrivons à la conclusion que le modèle à gros grain CGM est bien adapté pour concevoir des algorithmes concurrentiels, et qu'il est ainsi maintenant possible de vouloir développer du code parallèle portable, prédictif et efficace pour les problèmes concernant les graphes.

**Mots-clé :** algorithmes parallèles, algorithmes de graphes, modèles de gros grain, grappes de PC

## 1 Introduction

This article is situated at the intersection of two fields: the research on parallel models and the implementation of parallel graph algorithms on various platforms. Our aim is to give partial answers to the two following questions:

**Question 1** *Which parallel model allows to develop algorithms that are*

|  |  |
|---|---|
| ***feasible**:* | *they can be implemented with a reasonable effort,* |
| ***portable**:* | *the code can be used on different platforms without rewriting it,* |
| ***predictable**:* | *the theoretical analysis allows the prediction of the behavior in real platforms and* |
| ***efficient**:* | *the code runs correctly and is more efficient than the sequential code?* |

**Question 2** *What are the possibilities and limits of graphs handling in real parallel platforms?*

Of course these two questions are extremely ambitious in their generality and so we will only be able to answer them partly. We will do so by dealing with them jointly: we will choose a parallel model and some graph algorithms written in this model (we will justify these choices), then we will report on implementations and experiments of these algorithms on parallel machines of different structure. At the end, we will try to extract some elements to answer to the two initial questions:

- Can the model, beyond its use to handle graphs, be adapted generally to the constraints given in Question 1?
- Can we hope to conceive a library of parallel graph algorithms which is portable and efficient in a setting that extends the cases we have studied?

Algorithms that handle graphs have become basic tools in computer science and many problems in this field can be formalized in terms of graphs. Graphs form a class of objects that is general enough to deserve a special attention: its analysis can provide solutions or at least indicate promising candidates to many problems.

Some of these graph algorithms have a very high complexity and therefore it is often not sufficient to use a single computer to execute them: the times requirements are too large. Other algorithms have a lower complexity but they handle so many objects that these do not fit into the memory of a single computer. Some algorithms even have both drawbacks. Parallel computation is an attempt to solve these different problems.

Parallel graph algorithms are a field that has a rich development since the early beginning of parallel computation. But if there are a lot of theoretical studies in this area, relatively few implementations have been presented for all these algorithms that were designed.

Here in this article we try to start a comparative analysis of parallel graph algorithms. Our approach seems to be original in the sense that

- it consists in proposing new algorithms in one hand and
- in studying the behavior of these algorithms as well as other algorithms from the literature by experiments on different parallel platforms.

This allows to point out the possibilities and the limits of the parallel graph algorithms and their implementation.

The first difficulty that we encountered was to choose the algorithms to implement. For some problems, there are already good algorithms that were proposed whereas for others, either no algorithm exist or no existing algorithms is satisfactory. In such cases, we had to propose new algorithms. The analysis of the implementations is not our main goal, but we wanted also to demonstrate that it is now possible to write parallel code for graphs that respects the later three of the four programming goals as formulated above. Part of our aims is to constitute a portable library of parallel graph algorithms, so the choice of the algorithms is then fundamental to realize these different points abreast.

As stated above, the other main concern is the model to use as a starting point.

- The model has to take as many parallel platforms as possible into account to allow the writing of portable code.

It has to be realistic enough

- such that the implementation of the algorithms are actually feasible and
- such that the theoretical predictions give a good approximation of the real behavior of the programs.
- Also, the model should state some rules for the design of algorithms that are well adapted to the constraints of parallel computation and are likely to give efficient code from this type of algorithms.

The will of being realistic immediately turns our choice towards the coarse grained models like BSP, LogP and CGM that were proposed relatively recently. Among these, we chose the CGM model: it offered the simplest realization of the goals we had in mind.

Once the model chosen, we restricted the work to algorithms written in this model. We classify them according to the problem they solve into three main categories:

1. of general use as a subroutine,
2. general graph algorithms, and
3. algorithms for special graph classes.

Section 2 builds the foundations of this work. After giving a brief summary of the used terms, we review the main classes of the coarse grained models, that are BSP, LogP and CGM. Then we motivate the choice of the CGM model. We continue this section with a state of the art on the existing implementations of parallel graph algorithms. We will see that there are few works on the subject and most of them do not maintain the principles of efficiency, portability and predictability. Then we list the problems we have tackled and we justify these choices. We also describe the platforms we used, that are PC clusters and a parallel machine Cray T3E. By this choice of parallel platforms with very different architectures we are able to demonstrate the portability of the code.

Section 4 presents the results obtained for sorting. We will see that these results are positive and promising for the other algorithms that follow. Section 5 deals with the problem of list ranking. We will see that it is difficult to obtain efficient solutions for this kind of problems. In fact, we think that this problem is the most challenging from the point of view of feasibility (known algorithms are quite involved) and from the point of view of efficiency (the speedup is quite restricted).

Section 6 shows that it is possible to solve the connected components problem on dense graphs efficiently without the use of list ranking. This problem is basic for graph algorithmics.

Section 7 then shows that an algorithm with $\log p$ supersteps ($p$ is the number of processors) can be efficient in practice.

The following table reviews the different problems that we tackled. Here in this paper, we restrict ourselves to sorting, list ranking, connected components on dense graphs, and to a permutation graph problem. For the algorithms, results and analysis obtained on the prefix sum and some problems on interval graphs see Guérin Lassous (1999) and Ferreira et al. (1998).

| Problem | Portability | Efficiency | Predictability |
|---|---|---|---|
| Prefix sum | × | PC clusters × <br> T3E − | PC clusters × <br> T3E partly |
| Sorting | × | × | × |
| List ranking | × | × (restricted) | × |
| Connected Components for dense graphs | × | few vertices × <br> many vertices − | × (partly) |
| Number of oriented in-edges permutation graphs | × | × | × |
| Interval graph problems | × | × | × |

We present the results *as they are*, i.e. with all the variations and roughness that the obtained curves may experimentally show. We think that such variations that still remain after averaging over suitably many experiments are worth to be noted even if we are perhaps not able to explain them directly.

## 2 Parallel models

Other than just writing a program, the design of an algorithm needs a level of abstraction — a model. Such a model captures within a few parameters the machine(s) on which the algorithm is supposed to run and the rules of execution of the program resulting from an implementation. If the model is realistic enough, it is possible to know whether or not the algorithm is a good practical candidate. Algorithms for the same problem may also be compared with respect to a model to determine which one is the best for a given criterion (the execution time for instance).

If a model itself must be chosen, the different goals as defined above (feasibility, portability, predictability and efficiency) allow its evaluation. The choice must depend on the intended use: for instance, if very efficient results have to be obtained on a specific platform, a realistic model close to the machine might be preferred at the cost of portability and feasibility. If the study of the complexity of different algorithms is the topic, a usable model adapted to complexity needs will be selected.

### 2.1 Defi nitions and terminology

In this section, we give the basic terms which will be used in the following. The full definitions can be found in Jájá (1992) and Foster (1995).

A *parallel machine* is a set of processors which cooperate to solve a problem. This definition includes *parallel supercomputers* having hundred or thousand processors, *clusters* which represent a set of machines (workstations or PCs) interconnected by a network or *multiprocessors workstations*. Nowadays, essentially two different kinds of memory exist in parallel machines: in *shared memory machines*, the processors are connected via an interconnection network to a memory they share; in a *distributed memory machine*, each processor has its own memory called *local memory*, and the messages exchanges are most of the time realized by *message passing* on the interconnected network. Sometimes, the data exchanges can be done by *remote direct memory access*: a processor can directly access and handle a part of the memory of an other processor. In a *global communication*, each processor communicates with all the other processors, whereas in a *local communication*, a processor communicates with a limited set of processors. A communication is called *point-to-point* when a transmitter processor sends a message to a receiver processor. A machine runs in *synchronous mode* when all processors work under the control of a common clock. In asynchronous mode, each processor works under its own specific clock. In such a mode, the programmer has to put synchronization points where necessary via a *synchronization barrier*.

**Performance analysis** Performances of parallel algorithms can be characterized by several measures. The most natural is the *execution time* of the algorithm. In the following, the execution time of the parallel algorithm on $p$ processors will be denoted by $T_p$. The execution time can also be defined as the sum of the computation time, the communication time and the idle time of an arbitrary processor $j$:

$$T_p = T^j_{comp} + T^j_{com} + T^j_{idle}, \tag{1}$$

or as the average time compared to the first sum

$$T_p = \frac{1}{p}\left(\sum_{j=1}^{p} T^j_{comp} + \sum_{j=1}^{p} T^j_{com} + \sum_{j=1}^{p} T^j_{idle}\right). \tag{2}$$

A simple way to evaluate communications is to give the number of transmitted messages and their size in bytes or machine words. The communication time of a message of $k$ bytes will take a time

$$T_{mesg} = s + k * g \tag{3}$$

where $s$ is the *startup time* of the communication and $1/g$ is the *throughput* of the communication channel between two processors in bytes per second. We will find below that this is exactly the measure which BSP proposes to evaluate communications. The reality is in general more complex because the network structure also has an impact and so not all communications have equal cost. A parameter that is a characteristic of the network is the *latency* $L$. It corresponds to the time that is required to communicate a unitary message (the smallest size message of the network which is not necessary a byte) between two processors. To send a message will take a time greater than the latency.

The *absolute speedup* corresponds to the ratio between the execution time of an (hypothetical) optimal sequential algorithm $T_s$ and the execution time of the parallel algorithm on $p$ processors:

$$S_p = \frac{T_s}{T_p}. \tag{4}$$

The possibility of simulating a parallel algorithm on a sequential machine enforces that $p$ is the best absolute speedup which may be obtained. The *absolute efficiency*

$$E_p = \frac{S_p - 1}{p - 1} \tag{5}$$

measures the absolute speedup on a normalized scale and is expected to be in the range of $[-1, 1]$. Here a negative value actually means “speeddown” and a value close to $+1$ means an optimal speedup.

$S_p$ and $E_p$ have two main inconveniences. First, they can not be measured directly since they rely on $T_s$ which is a theoretical invariant of the problem and not of a specific algorithm. Second, they might not be observable at all for a given input size on a given platform:

- Every machine has its particular limits of what size of data it may process and so a problem of a specific size might be intractable for a sequential machine, but tractable with several processors.

Therefore we have to rely on more realistic grounds and introduce two new parameters. The *relative speedup* measures the speedup that is obtained when we execute the algorithm with $p$ processors instead of $p'$ $(p > p')$ and is given by

$$S^{rel}_{p',p} = \frac{T_{p'}}{T_p}. \tag{6}$$

Observe that for $p' = 1$ we usually hope to get an approximation of $S_p$. It is again desirable to normalize $S^{rel}_{p',p}$ and define the *relative efficiency* as

$$E^{rel}_{p',p} = \frac{1}{\frac{p}{p'} - 1}(S^{rel}_{p',p} - 1). \tag{7}$$

Again, this quantity will in general be in the range of $-1$ to $+1$, negative values meaning that passing to $p$ processors was not of much use, and values close to $+1$ meaning that we achieve a good (local) improvement on the running time. But, beware also that artifacts may take place and we may even bserve values that are greater than 1, called *superlinear speedup*. This will in general be the case when the problem was too large for our machine with only $p'$ processors. We will discuss such cases below.

## 2.2 The parallel models

The first parallel models that have been proposed are fine grained models. In these models, we suppose that $p = \theta(n)$ with $n$ the size of the problem and $p$ the number of processors. Among the fine grained models, two main classes have emerged: the PRAM model (Parallel Random Access Machine), see Fortune and Wyllie (1978) or Karp and Ramachandran (1990), and the distributed memory machines models, see Leighton (1992) or Ferreira (1996). If the PRAM model is very useful to point out the possible parallelism of a problem, it is quite unrealistic. Therefore the implementation on a real platform of a PRAM algorithm is not straightforward and often leads to performances that are not competitive. The distributed memory models are more realistic

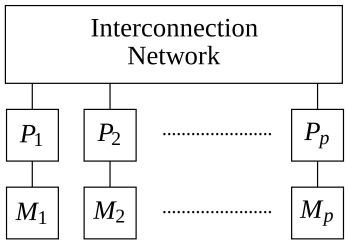


Figure 1: *Multiprocessor Machine in a Coarse Grained Model*

because they are close to the network structure of the considered machine and so in general they lead to code that is more efficient. But the price for this efficiency is high: such code is usually not portable because it strongly relies on the interconnection infrastructure.

**Coarse Grain** Nowadays, most of the used parallel machines are coarse grained, that is every processor has a substantial local memory. Recently, several works tried to give models that take realistic characteristics of existing platforms into account while covering at the same time as many parallel platforms as possible. In particular, they consider costs of communications (which is not the case in the PRAM model) but without specifying the topology of the communication medium (as the distributed memory models). Proposed by Valiant (1990), BSP (Bulk Synchronous Parallel) is the originating source of this family of models. It formalizes the architectural features of existing platforms in very few parameters. The LogP model proposed by Culler et al. (1993) considers more architectural details compared to BSP, whereas the CGM model (Coarse Grained Multicomputer) initiated by Dehne et al. (1993) is a simplification of BSP.

Note also that if we add the fine grained models in the scale of the details of the description, PRAM has a higher abstraction than CGM (it gives less details), and the distributed memory models have the lowest abstraction (they describe very precisely the target machine). We chose CGM because it has a high abstraction that easily enables the design of algorithms. But we do not hesitate to use BSP or LogP costs when CGM could not afford a satisfying explanation. In this work, we try to justify that this chosen level of details is well suited to handle graphs in parallel.

The three models of parallel computation have a common machine model: a set of processors that is interconnected by a network. A processor can be a monoprocessor machine, a processor of a multiprocessors machine or a multiprocessors machine. The network can be any communication medium between the processors (bus, shared memory, Ethernet, etc). An exact description of the network topology is not part of the model. In Figure 1, the processor $P_i$ has its memory $M_i$ and is linked with the other processors via the interconnection network.

In the following we give a short description of the three models. We refer the reader to the references as given above for a detailed analysis of these models.

**BSP** An algorithm written in the BSP model is composed of a sequence of *supersteps*. During a superstep, a processor can do local computations and communications (composed of send and receive). Two consecutive supersteps are synchronized by a synchronization barrier. The BSP model uses different parameters to characterize the supersteps, like $L$ the synchronization period that corresponds to $L$ time units required to synchronize all the processors, $g$ the bandwidth, $h$ the maximal number of words that a processor can send or receive during a superstep, and $s$ the overhead of any communication. These parameters give a first approximation on the communication cost. A communication step requires a time $gh+s$. The local computations cost corresponds to the sum of the costs of each elementary operation. The complexity of a superstep is the sum of the cost for local computations, the cost for communications and $L$ the synchronization barrier. The complexity of a BSP algorithm is the sum of the costs of its supersteps.

**LogP** The LogP model tries to reflect the features of real machines more precisely. The four letters of the word LogP are composed from the communications costs: $L$ the latency, $o$ the overhead of a communication, $g$ the *gap* that is the minimum time interval between two consecutive sends or two consecutive receipts of messages on one processor and $P$ the number of processors. This model assumes that the interconnection network has a finite capacity: at each superstep, a processor can only send or receive at most $\lceil \frac{L}{g} \rceil$ messages. $L$, $o$ and $g$ are assumed to be multiples of the processor cycle (time of an elementary operation).

The processors work asynchronously. The communications between the processors are point-to-point. To limit the communications, it encourages a good partitioning of the data and the tasks, as well as a good scheduling but it does not give such a partitioning explicitly.

Some extensions of LogP have been proposed, as LogGP by Alexandrov et al. (1995) or generalized LogP by Löwe et al. (1997) and Keeton et al. (1995). It is possible to simulate BSP by LogP and vice-versa. But the conclusions of Bilardi et al. (1996) seem to show that BSP is simpler to use and leads to more portable code than LogP. The simulation of BSP in LogP is not straightforward and gives results that are rather slow in practice. Nevertheless, the authors show that the two models are equivalent asymptotically.

**CGM** The CGM model, a simplified version of BSP, does not take the parameters $l, g, h$ and $s$ and the synchronization step into account. It gives the number of data per processor explicitly. Indeed, for a problem of size $n$, it assumes that:

$$\text{The processors can hold } O(\tfrac{n}{p}) \text{ data in their local memory.} \tag{8}$$

$$1 \ll \frac{n}{p}. \tag{9}$$

Usually the later requirement is put in concrete terms by assuming that

$$p \leq \frac{n}{p}, \tag{10}$$

because each processor has to store information about the other processors, and this assumption is quite reasonable in practice.

The algorithms are, like in BSP, an alternation of supersteps. In a superstep, a processor can send or receive

- once to and from each other processor and
- the amount of data exchanged in a superstep by one processor in total is at most $O(\frac{n}{p})$.

Unlike BSP, the supersteps are not assumed to be synchronized explicitly. Such a synchronization is done implicitly during the communications steps. In particular it is well possible (and intended) that for some number of supersteps the processors are divided into subgroups that don't communicate which each other and thus would not (and should not) be synchronized.

CGM makes an abstraction of the different parameters that enter into the cost functions for communications in BSP or LogP. The price for that simplification is that in CGM we have to ensure that the number $R$ of supersteps is particularly small compared to the size of the input.

To see that let us compute the BSP communication cost of a CGM algorithm of all the data sent by an individual processor. Let $h_1, \ldots, h_r$ be the messages that are sent by this processor. By the restrictions of the model we know that at most $p-1$ messages can be sent by one processor in a superstep and so we must have

$$r \leq R(p-1). \tag{11}$$

So the total BSP cost caused by these sends is $\sum_1^r (g \cdot h_i + s + L)$ if we insist that for each such communication we have to synchronize the processors. With (11) this sum is bounded:

$$g \sum_1^r h_i + \sum_1^r (s+L) \leq g \sum_1^r h_i + R(p-1)(s+L). \tag{12}$$

So the total communication cost of an algorithm that communicates $M$ data in total can be split as follows:

$$\underbrace{g \cdot M}_{\text{bandwidth requirements}} + \underbrace{R \cdot (p^2 - p)(s + L)}_{\text{communication overhead}} . \tag{13}$$

Now if we ensure that $R$ is a function that only depends on $p$ (and not $n$) we see that for a *fixed* number of processors $p$ the communication overhead is constant. So for a total communication $M$ that is suitable large the total cost is completely dominated by the bandwidth requirements.

There have been some discussions about what functions $R$ should be allowed for efficient algorithms. In particular, it has been formulated as a goal that $R$ has to be as small as possible or at most polylogarithmic in $p$, but the above estimation gives no rationale for that. In fact, the experience in the BSP literature shows well that we can be quite permissive for $R$. E.g. for matrix multiplication a known efficient practical algorithm has $O(\sqrt{p})$ supersteps, see Goudreau et al. (1996).

### 2.3 The choice of the CGM model

CGM was chosen as a starting point for this work. This choice is justified by several points:

- To allow experimental studies, the model has to be realistic enough to represent existing platforms. This rules out the use of the PRAM model.
- One of the final goal is to conceive a portable library of parallel graph algorithms that is not dedicated to a specific platform. It must run on a diversity of platforms. Therefore, it was not possible to use distributed memory models.
- Unlike BSP or LogP, CGM has very few constraints on the train of supersteps and on the realization of the communications. For instance, the choice between point-to-point communications or of using multicast is left to the programmer and only depends on the specific routines offered by any given platform.
- The inherent tendency of CGM to have large local computation steps allows (and demands) the use of good existing sequential algorithms during the local computations.

One of the weak points of CGM is the way the complexity is expressed. It models the time required for communications quite imprecisely. In the following, will try to find answers to the following questions:

- Is the simplification about the complexities of the different communication routines excessive?
- Is the number of supersteps $R$ a good parameter for predictions?
- Can we give good predictions without using the total amount of communication $M$?

## 3 Problems and methodology

On one hand a lot of parallel graph algorithms are written in high level models like the PRAM model or the coarse grained models. On the other hand, as far as we know, few implementations of these algorithms have been carried out on parallel machines or clusters and even less results on experiments clearly noting the parameters that were investigated have been published.

### 3.1 Synthesis of existing implementations

At the beginning of parallel implementations of graph algorithms in the early nineties, it was first of all necessary to investigate the feasibility of certain algorithms on the platforms that existed at that time. What was offered by these platforms was by no means homogeneous, for example no good portable communication libraries were available. This changed only relatively recently such that nowadays stable, reliable platforms with

| Problem | Model | Reference | Porta-bility | Efficiency | Predic-tivity |
|---|---|---|---|---|---|
| Sorting | DM | Blelloch et al. (1991) | | parallel only | × |
| Sorting | LogP | Dusseau (1996) | | parallel only | × |
| Sorting | BSP | Krizanc and Saarimaki (1999) | × | × | × |
| List ranking | PRAM | Reid-Miller (1994) | | × | × |
| List ranking | not specified | Sibeyn (1997) | | × | × |
| List ranking | not specified | Sibeyn et al. (1999) | | × | × |
| Connected components | PRAM | Hsu et al. (1997) | | parallel only | × |
| Connected components | PRAM | Krishnamurthy et al. (1997) | | × | |
| Connected components | PRAM | Lumetta et al. (1995) | | × | |
| Connected components | PRAM | Kumar et al. (1997) | | large data | |
| Connected components | PRAM | Greiner (1994) | | parallel only | |
| Connected components | BSP | Bäumer and Dittrich (1996) | × | large data | |
| Spanning tree<br>Minimum spanning tree<br>Ear decomposition | PRAM | Hsu et al. (1995) | | large data | × |
| Minimum spanning tree | BSP | Dehne and Götz (1998) | × | dense graphs | × |
| Minimum spanning tree<br>Shortest path | BSP | Goudreau et al. (1996) | × | large data | × |

Table 1: Summary of the actual implementations of parallel graph algorithms

portable software components are available. Caused by these restrictions, most implementations had to be restricted to case studies on particular platforms and up to our knowledge there are only few works dealing with parallel graphs algorithms which lead to efficient, portable and predictive implementations.

Table 1 reviews the references that we found and gives for each implemented problem an indication of the model that was used and the goals that were obtained. For the field "Model", "*DM*" denotes a distributed memory model. "Portability" means that the authors do not use specific languages or libraries dedicated to one platform, but languages and tools known to be portable. For the field "Efficiency", the term "*parallel only*" means that some different parallel algorithms are compared to each other and no comparison is done with a sequential algorithm. The terms "*large data*" and "*dense graphs*" point out that the implementation is efficient for large graphs or dense graphs, only. A cross in this field means that the parallel algorithm is efficient compared to the sequential one. The field "Predictivity" means that the authors do or do not draw comparisons between the obtained results and theoretical predictions.

**Sorting** Blelloch et al. (1991) start from sorting algorithms written in distributed memory models and give a modification of the algorithm according to the architecture of the target platform, whereas Dusseau (1996) compares different algorithms written in the LogP model. In Krizanc and Saarimaki (1999), a BSP sort is implemented on various platforms. In Blelloch et al. (1991), Dusseau (1996), the algorithms are implemented with the Split-C language that does lead to a complete portability. These two works do not compare the obtained results with the sequential execution times but give a complete comparison of the chosen algorithms. They also deal with the predictivity question. On the other hand, the work of Krizanc and Saarimaki (1999) presents a portable, efficient and predictive sort algorithm written in the BSP model. In fact, Krizanc and Saarimaki (1999) implement the same BSP algorithms as we do ourselves and with the same results concerning portability, efficiency and predictivity. This indicates, first of all, that this sorting algorithm is really of practical relevance and that the simplification when only using CGM and not BSP caused no harm.

**List ranking** Reid-Miller (1994) proposes several versions of a PRAM algorithm modified according the architecture of the target platform, the Cray C-90. She also deals with the prediction side of her algorithms and achieves very good results for one of the presented algorithms. Sibeyn (1997) and Sibeyn et al. (1999)

start from techniques used in PRAM algorithms and some new ones. They fit them to distributed memory machines and especially to the Intel Paragon. Several features of the interconnection network of the Paragon are considered to fine-tune their algorithms. They compare the obtained results with theoretical predictions on the Paragon. For large lists, speedups of $\frac{p}{4}$ and $\frac{p}{3}$ are obtained.

**Connected components** This is probably the problem studied the most. Hsu et al. (1997), Krishnamurthy et al. (1997), Lumetta et al. (1995), Kumar et al. (1997), Greiner (1994) propose one or several modified versions of PRAM algorithms, whereas the algorithm of Bäumer and Dittrich (1996) is directly written in the BSP model. Hsu et al. (1997), Kumar et al. (1997), Greiner (1994) give a modification of their algorithm in function of the chosen platform, whereas Krishnamurthy et al. (1997), Lumetta et al. (1995) modify their algorithm for distributed memory machines without specifying a special topology. Bäumer and Dittrich (1996) use a BSP library, whereas the other implementations use specific languages that do not allow much portability. Hsu et al. (1997), Greiner (1994) do not compare the execution time of their parallel algorithms with the execution time of the sequential algorithm. In Kumar et al. (1997), Bäumer and Dittrich (1996), the results become efficient for very large graphs, whereas good results are obtained for any kind of graph in Krishnamurthy et al. (1997), Lumetta et al. (1995). Only in Hsu et al. (1997), Bäumer and Dittrich (1996), the predictivity question is tackled.

**Spanning forest** Hsu et al. (1995) propose a modification of a PRAM algorithm, whereas in Dehne and Götz (1998), Goudreau et al. (1996) the algorithms are written in the BSP model. Both, Dehne and Götz (1998) and Goudreau et al. (1996), use a BSP library, whereas Hsu et al. (1995) use the MPL language dedicated to the MasPar platform. Efficiency seems difficult to reach for Hsu et al. (1995), whereas the works of Dehne and Götz (1998), Goudreau et al. (1996) obtain efficient results, especially for large data in Goudreau et al. (1996) and for dense graphs and with few processors in Dehne and Götz (1998). In all these articles, the problem of predictivity is dealt with.

**Other problems** Hsu et al. (1995) also deal with the ear decomposition problem and Goudreau et al. (1996) with the lowest path problem.

All the implemented algorithms are basic algorithms for graphs. Note that only the BSP algorithms satisfy the three points efficiency, portability and predicitvity. So the coarse grained approach seems to be a good choice to achieve these three goals.

**The different problems we tackled** We have tackled different problems to give preliminary answers to Questions 1 and 2 given in Section 1. As already stated above, we find it convenient to establish a hierarchy amongst the algorithms that are treated. We use the three main classes as described in Section 1. We have realized the following implementations:

1. parallel prefix sum and maximum, sorting and list ranking,
2. connected components for dense general graphs,
3. maximum weighted clique and connected components for interval graphs and indegree in a permutation graph.

Figure 2 gives the different classes of problems on which we have worked and the dependencies between these problems. Here $a \to b$ means that the problem $b$ resorts to the problem $a$.

For the present paper, we only present the results of four of the problems in detail. First, we will deal with sorting because it is a basic problem which is very often used in graph algorithms. Next we will present the results of the list ranking problem. We will point out that it is a fundamental obstacle to have efficient and portable implementations on parallel graph algorithms. We will show thereafter how to obtain efficiently

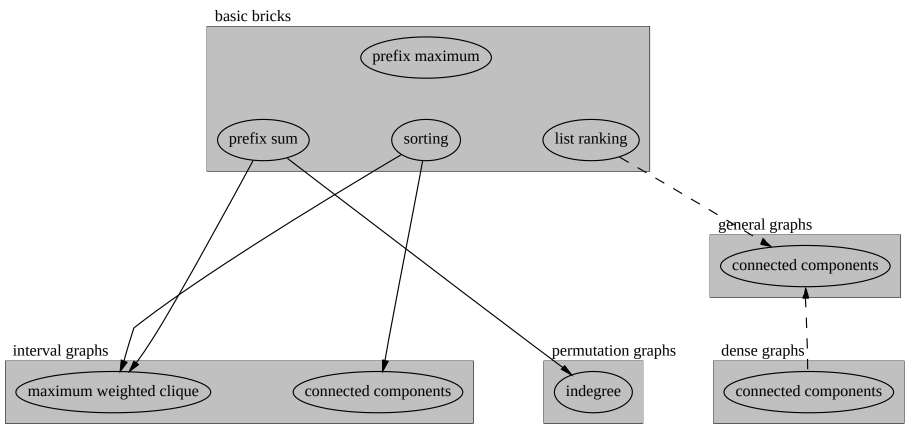


Figure 2: Considered algorithms

connected components without the use of the list ranking problem on dense graphs. At last, we will see with permutation graphs how a sparse encoding due to properties of a special graph class can lead to efficient results in parallel.

## 3.2 Implementation background

We have implemented these algorithms such that they run on different platforms: two PC clusters and a distributed memory parallel machine. Note that we also ran some of our codes on an Origin2000[1] but we do not report the associated results in this paper. We have deliberately chosen platforms of different kinds in order to show that our code could run on various architectures. Moreover, the use of two clusters with different interconnection networks should allow to check that the code is not too dependent on the underlying network.

**PC clusters** The first cluster[2] consists of 13 *PentiumPro* 200 PCs with 128 MB memory each. The PCs are interconnected by a 100 Mb/s full-duplex *Fast Ethernet* network, see Tanenbaum (1997), having 93 $\mu$s latency. In the following, we refer to it as *PF*. The second cluster[3] consists of 12 *PentiumPro* 200 PCs with 64 MB of memory each. The interconnection network is a *Myrinet*[4] network of 1.28 Gb/s and with 5 $\mu$s latency. We refer to it as *POPC*.

**T3E** The parallel machine[5] we used is a *Cray* T3E with 256 DEC $\alpha$ EV5 processors of 300 MHz. Each processor has a local memory of 64 MB to 2 GB. The interconnection network is a three dimensional torus with 1.0 $\mu$s latency and a bandwidth of 480 Mb/s.

**The library CGM** The implementations use a special library CGM that we have built. It implements the level of abstraction that is necessary for algorithms designed in the CGM model. It manages the CGM processes and supplies the main communication routines between these processes without that it is necessary to rely on a specific communication library. This library and the code of the implemented algorithms use C++ as a programming language. The communication libraries that were used are PVM, see Geist et al. (1994), and MPI, see Forum (1993).

[1] http://cch.loria.fr/
[2] http://www.inria.fr/sophia/parallel
[3] http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html
[4] http://www.myri.com/
[5] http://www.idris.fr

## 3.3 An outline of the analysis

Very few papers dealing with the implementations of parallel graph algorithms describe very precisely how they realize their tests, what they measure, and with which system routines they perform their measures. In this section, we want to specify what kind of experiments we have carried out on each of algorithms and how they were realized.

To keep a unity between the different algorithms, we define the same outline for the analysis. We check if the four goals as claimed by the CGM model are verified in practice. More precisely, for each problem we study the points given previously: feasibility, predictability, portability and efficiency. For the efficiency, we can study :

- The execution time related to different parameters (input size, processors number, number of supersteps).
- The absolute speedup compared to the best sequential implementation. The relative speedup if we use more processors.
- The memory requirements. The bandwidth requirements.

Some of the points are subdivided further if necessary. For instance, the execution time may be decomposed into the local computation time and the communication time. It can also be useful to have the time of the different operations of an algorithm in order to extract the majoring steps in this algorithm.

The execution time is not the only measure of the performance of an algorithm. It is also important to study the other resource requirements of the program like memory or bandwidth. When the memory of a machine becomes saturated, the machine begins to swap, and execution times start to degrade drastically. It can be useful to use several machines to get rid of this swapping effect. This is not always possible because the programs require sometimes a lot of memory space. Therefore, we will see if it is possible or not to handle large data which generate swap effect in sequential. The bandwidth requirement strongly impacts on the architectures on which a code can be run successfully.

This question list is not restrictive. According to each problem, we will study other parameters. Nevertheless, we will try to keep this outline.

All the tests have been carried out ten times for each input size. The results given are an average of the ten tests. All the execution times are in seconds. The times have been measured by the system function *gettimeofday* which is a kind of chronometer. By that, other processes external to the program are also measured, but nevertheless, we have found this to be the most realistic. As a consequence, we always tried to run the programs when the machines were barely loaded.

Each execution time is taken as the maximum value of the execution times obtained on each of the $p$ processors. The measurement of the time required by the communications is a trickier point. No articles on the different implementations explain their principle of measurement when they show the communication times. We have decided to measure the time of the communication step in each superstep rather than for individual communications. And in such a step, the processors send and receive some data, can do some local computations as to rearrange the data, to put them in buffers, etc. Therefore, we do not only measure the time when the data is communicated, but also all the required operations for the communication step. In each superstep, each processor starts its chronometer just before the beginning of the communication round and stops it at the end of this round (when it has received all the expected messages). Note that this measure takes into account the overhead $o$ given in LogP.

All of our algorithms use arrays to maintain their data. To explain the algorithms, we often assume that the data are put in a *global array* and that each processor works on a part of this array. In fact, each processor has a *local array* and the union of these arrays forms the global array (which is virtual). A *local* (resp. *global*) *index* corresponds to an index of a local (resp. global) array. From a global index of an element, it is easy to compute the processor number which has this element and its local index. For instance, if each processor has exactly the same number of elements (which is often the case in the CGM model), the element of global index $k$ is on the processor $\lfloor \frac{k}{p} \rfloor$ with the local index $k \bmod p$. To get the global index from the local one consists in multiplying the number of elements on a processor by the processor number and in adding the local index.

If, actually, the processors do not have the same number of elements, a parallel prefix sum on the number of elements on each processor can be done. The $i$th element of this sum corresponds to the smallest global index of the part of the global array of the processor $i$. It is also easy to switch from the local index to the global one with this sum and vice versa.

To test and instrument our code we generated input objects randomly. Most of these objects were constructed from random permutations. See Guérin Lassous (1999), Guérin Lassous and Thierry (2000) for more details. The time required for the generation of an object is not included in the times as they are presented.

## 4 A basic operation: sorting

The basic operations, including the prefix operations and sorting, are not specific to graphs, but nevertheless they are very often used in graph algorithms. These operations are basic because they are called at least once in most graph algorithms, sometimes even several times during the same algorithm. Thus, the implementations of these algorithms have to be efficient and the impact of these routines on the algorithms that use them later is crucial.

We restrict the presentation to sorting. We briefly present the algorithm and its impact, and will give a survey of the results obtained on POPC, PF and the T3E afterwards.

### 4.1 The choice of the algorithm

The choice of the sorting algorithm is a critical point due to its widespread use to solve graph problems. The synthesis of all existing sortings is out of scope of this paper, but we will nevertheless try to justify our choice. For a review on parallel sorts see Goodrich (1996).

Among the sorts proposed in the BSP model, there are deterministic ones, see Goodrich (1996), Adler et al. (1995), Gerbessiotis and Siniolakis (1996), and randomized ones, see Gerbessiotis and Valiant (1994). In the CGM setting, all these algorithms translate to have a constant number $R$ of supersteps. The algorithm proposed by Goodrich (1996) is theoretically the most performing, but is complicated to implement and quite greedy in its use of memory.

The algorithms of Gerbessiotis and Siniolakis (1996) and Gerbessiotis and Valiant (1994) are both based on the sample technique: some elements are selected to form the splitters that will allow to cut the input elements in packets having the following property: if $i < j$ then each element of the packet $i$ is below all elements in packet $j$. During the execution of the algorithm, the packet $i$ is assigned to processor $i$, which sorts then the data locally.

Algorithm 1 describes the sort of Gerbessiotis and Valiant (1994) which we implemented. The tricky point of this kind of algorithm is to guarantee that the elements are evenly distributed within the packets. The choice of the splitters is then essential to ensure that the packets have more or less the same size. The algorithm is randomized and bounds the packets size by

$$\left(1+\frac{1}{\sqrt{\ln n}}\right)\left(\frac{n-p+1}{p}\right) \tag{14}$$

with high probability, only. Nevertheless, we have chosen this one because it is conceptually simple and requires very few supersteps (3 in total and only one is of large size).

The local sort used in Steps 2 and 4 can be any convenient sequential sort, and we will see below that it might be adequate to chose different routines in both steps. $T_S(n)$ denotes the running time of such a sequential sort on $n$ data henceforth. The analysis of this algorithm is in fact simple. The first two steps and the send phase of the third have deterministic running times. Step 1 requires $O(k)$ local computations, Step 2 $O(T_S(pk))$ and Step 3 $O(\frac{n}{p}\lceil\log(p-1)\rceil)$. Step 4 realizes its running time with high probability in $O(T_S(\frac{n}{p}))$.

**Algorithm 1:** Sorting

```
Input: T array of $n$ elements ($\frac{n}{p}$ elements per processor), an integer $k$
Output: the elements of T, sorted
1 foreach processor do
      Choose $k$ splitters;
      Send the splitters to processor 1
2 for processor 1 do
      Sort the $p \cdot k$ splitters;
      Evenly select $(p-1)$ elements among the $p \cdot k$ splitters;
      Send all $p-1$ chosen values to all other processors
3 foreach processor do
      Distribute the data according to the splitters;
      for $(i = 1; i \leq p; i = i+1)$ do
          send the packet $i$ to processor $i$
4 foreach processor do
      Sort the received data locally
```

To equilibrate the size of the packets, $k$ has to be chosen large enough, but at the same time it must not be too large so that Steps 1 and 2 don't dominate the computation. Gerbessiotis and Valiant (1994) advise to take $k$ equal to $\lceil 1.8 \ln^2 n \rceil$ if $n > 3500$ and $p^3 \leq \frac{n}{log_2^2 n}$, for $p \geq 2$. If $p \leq 20$, this condition is satisfied for $n \geq 2^{21}$. If $p = 32$, the condition is satisfied for $n \geq 2^{23}$, whereas if $p = 64$, it is the case for $n \geq 2^{26}$. So with many processors, we should only try to sort data of large size.

The CGM model assumes that each processor has a local memory of size $O(\frac{n}{p})$. Since there are $pk$ splitters on processor 1 at Step 2, $pk \leq \frac{n}{p}$ or equivalently $p^2 \lceil 1.8 ln^2 n \rceil \leq n$ must be fulfilled. This condition is still satisfied if we consider the pairs of values $(p, n)$ given above.

The following theorem can be easily deduced from what is stated in Gerbessiotis and Valiant (1994) in terms of BSP.

**Theorem 1 (Gerbessiotis and Valiant (1994))** *In the CGM model, sorting can be solved with probability* $1 - o(1)$ *in* $O(T_S(\frac{n}{p}) + \frac{n}{p}\lceil \log(p-1) \rceil)$ *local computations on each processor and with a constant number of supersteps.*

Observe that this lets us expect that the invariants $S_p$, $E_p$, $S_p^{rel}$ and $E_p^{rel}$ should be independent on the choice of the particular sequential sorting routine that we chose as a basis.

If we study the operations done in the sort more precisely, we see that if we use the counting sort (Cormen et al. (1990)), Step 4 requires $3\frac{n}{p}$ local computations and Step 3 requires $\frac{n}{p}\lceil \log(p-1) \rceil$ computations. Therefore the absolute speedup can not be greater than

$$\frac{3}{3 + \lceil \log(p-1) \rceil} p. \tag{15}$$

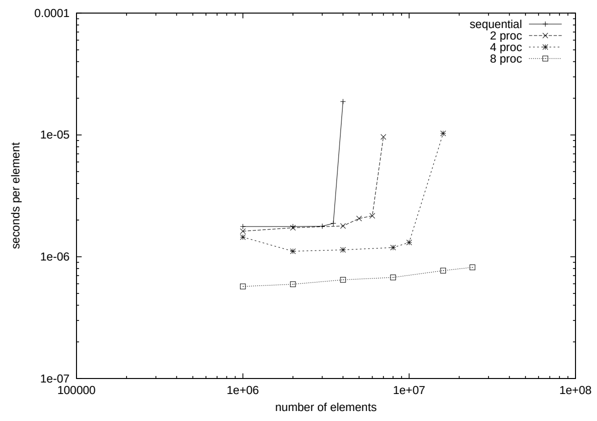
Formula (15) gives us an estimation of what we can expect for the parameters $S_p$, $E_p$, $S^{rel}$ and $E^{rel}$, see Table 16. Here the parameters $S^{rel}$ and $E^{rel}$ are taken for consecutive values of $p'$ and $p$, eg the values in column '12' are $S_{8,12}^{rel}$ and $E_{8,12}^{rel}$, respectively.

| $p$ | 2 | 4 | 8 | 12 | 32 | 64 |
|---|---|---|---|---|---|---|
| $S_p$ | 2 | 2.4 | 4.0 | 5.1 | 12 | 21.3 |
| $E_p$ | $\frac{0}{0}$ | 0.35 | 0.43 | 0.37 | 0.35 | 0.32 |
| $S^{rel}$ | $--$ | 1.2 | 1.7 | 1.3 | 2.3 | 1.75 |
| $E^{rel}$ | $--$ | 0.2 | 0.7 | 0.6 | 0.78 | 0.75 |

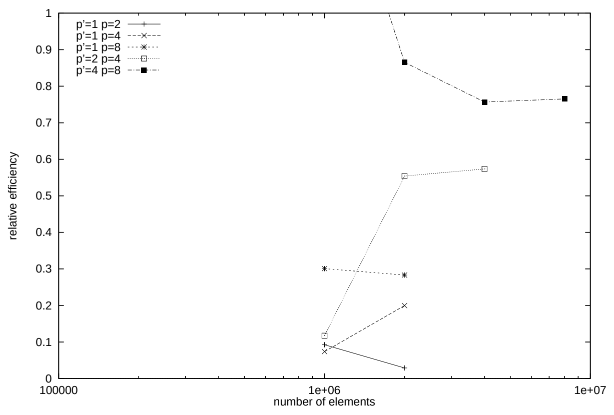
(16)

## 4.2 The implementation

We realized the implementation of Algorithm 1 to sort integers in parallel. The sorted integers are the standard 32 bit `int` types of the machines. We used counting sort, Cormen et al. (1990), as the sequential sorting subroutine in Step 4. This sort assumes that each key is an integer of the interval $[1,l]$. In practice, it is used when we have that $l = O(n)$ and by that it gives $O(n)$ for the execution time. We tested the program on random permutations. As the elements are well balanced among the processors with high probability, we have to sort $O(\frac{n}{p})$ consecutive data at Step 4, which corresponds to an execution time of $O(\frac{n}{p})$.



(a) Execution times



(b) $E^{rel}_{p',p}$

Figure 3: Sorting on POPC

To sort the potential splitters, we use quicksort. Counting sort is not feasible, because the keys of the potential splitters are included between 1 and $n$ which would give an execution time and memory requirement of $O(n)$ per processor. So this step takes us a time ($O$-notation omitted)

$$\begin{aligned} kp\log(kp) &\approx \log^2(n)p\log\left(\log^2(n)p\right) \quad (17)\\ &\approx \log^2(n)p\left(\log\log(n)+\log(p)\right). \quad (18)\end{aligned}$$

This is only growing very slowly in $n$ and so its impact should be neglectable for large data.

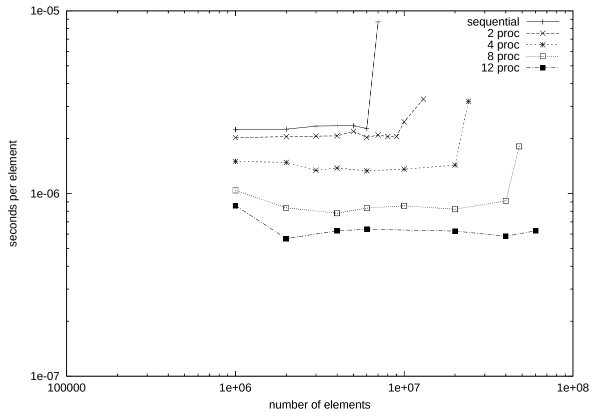
To distribute the data according the splitters, it would be possible to first sort the data on the processors locally and then to rank them according to the splitters in time $O(\frac{n}{p})$. As for the sort of the splitters, we would have to use quicksort and thus introduce a factor of $\log\frac{n}{p}$ and so this would give $O(\frac{n}{p}\log\frac{n}{p})$ for the execution time at Step 3. To avoid this log-factor in $n$, we do a dichotomic search on $p-1$ to find the destination packet of each element. By that we only introduce a factor of $\log p$.

In all the given figures, the x-axis corresponds to $n$, the number of integers to sort, and the y-axis gives the execution time in *seconds per element*. Both scales are logarithmic.

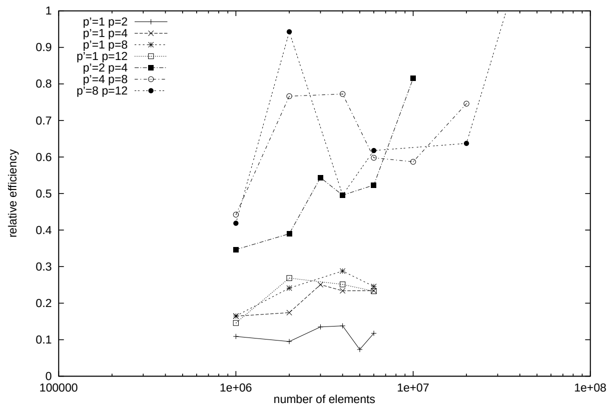
**PC clusters** Figures 3 and 4 give the execution times per element of the program with $1,2,4$ and 8 PC for POPC, and with $1,2,4,8$ and 12 PC for PF, respectively. The right ends of the curves for the execution times clearly demonstrate the swapping effects. Measures begin at one million elements to satisfy the inequalities given by Algorithm 1 as explained before. The memory of an individual PC in PF is two times larger than the one for POPC, therefore PF can sort two times more data.

As expected, we see that the curves (besides swapping) are near constant in $n$ and the execution times are neatly improved when we use $2,4,8$ or 12 PC.

We see that this parallel sort can handle very large data efficiently, whereas the sequential algorithm is stuck quite early due to the swapping effects. For instance, on PF, before the swapping effects, one PC can sort until 7



(a) Execution times

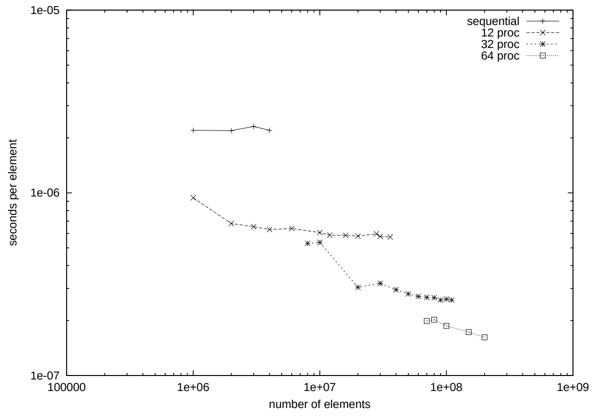


(b) $E^{rel}_{p',p}$

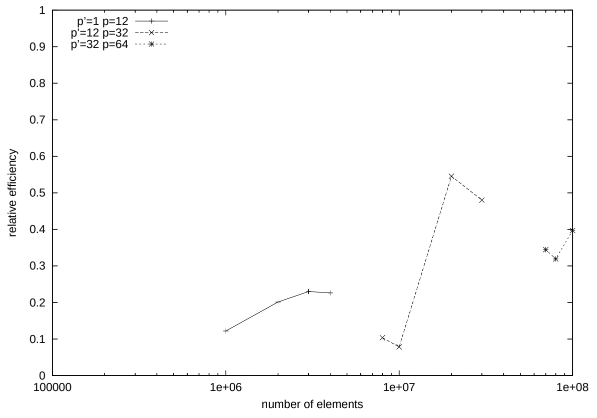
Figure 4: Sorting on PF

million integers, whereas 2 PC can sort until 12 million elements, 4 PC until 23 million data and 12 PC until 76 million integers. If the sequential sort can solve the problem on $n$ data within a sensible time, then the parallel sort will solve it on a little bit less than $pn$ data with $p$ processors before the swapping effects.

To compare the experiments to the theoretical bounds of Table (16), observe that the curves for the relative efficiencies $E^{rel}_{1,2}, E^{rel}_{2,4}, E^{rel}_{4,8}$ and $E^{rel}_{8,12}$ tend to the values we expect, i.e. to 0.2 for $p' = 1, p = 2$ and around 0.7 for the others. The curves for $E^{rel}_{1,p}$ are also situated where we can expect them to be, near the value of $E_p$, which is in the range of 0.2 – 0.3. Note that we only give the values for $E^{rel}_{p',p}$ where the behavior of the execution times per processor is constant.



(a) Execution times



(b) $E^{rel}_{p',p}$

Figure 5: Sorting on the T3E

**T3E** Figure 5 gives the execution time for 1, 12, 32 and 64 processors on the T3E. Each curve stops before the local memory of each processor becomes saturated. To satisfy the above inequalities, the measures begin with 1 million integers for 12 processors, 8 million elements for 32 processors and 70 million elements for 64 processors. The curves are also linear in $n$. Moreover, the execution times again improves when we use more processors. In addition we are able to sort very large data since 64 processors can sort up to 200 million integers in less than 35 seconds.

**Assessment** We can say that Algorithm 1 leads to an efficient code which can sort large data as well on PC clusters as on the T3E. If we compare the execution times with 12 PC of PF and with 12 processors of the T3E, we see that the results are almost the same. This might be surprising because the processors of the T3E are faster. The different processor speed seems to be ruled out by a much better C++ compiler for the PC.

## 5 The list ranking problem

The list ranking problem frequently occurs in parallel algorithms that use dynamic objects like lists, trees or graphs. The description of the problem is very simple: given a linked list of elements, for each element $x$ we want to know the distance from $x$ to the tail of the list. To be useful as a subroutine for other problems, it is desirable to extended to a more general setting where the list is cut into sublists. Then it consists in determining for each node its distance to the last node of its sublist. Figure 6 gives an example of this problem. The circled nodes are the last nodes of sublists. If it is easy to solve it sequentially, it seems much more difficult in parallel.

$$\begin{tikzcd}
\text{list} & 3 \arrow[r] & 5 \arrow[r] & \textcircled{1} \arrow[r] & 7 \arrow[r] & 6 \arrow[r] & 2 \arrow[r] & \textcircled{4} \\
\text{distance} & 2 & 1 & 0 & 3 & 2 & 1 & 0
\end{tikzcd}$$

Figure 6: Example of the List Ranking problem

The first proposed algorithms were formulated in the PRAM model. The first author to have mentioned the problem is Wyllie (1979). He uses the pointer jumping technique that consists in for each element of the list to point to the successor of its successor and to iterate this process. This technique solves the list ranking problem in time $O(\log n)$ time and with a total number of computations of $O(n \log n)$ (called workload henceforth). This is not work optimal, because the list ranking can be solved sequentially in $O(n)$ computations. Many authors worked on this issue: Anderson and Miller (1988), Cole and Vishkin (1988, 1989), Reid-Miller (1994), Anderson and Miller (1990), Dehne and Song (1996), Caceres et al. (1997), Sibeyn (1997), Sibeyn et al. (1999). The first work optimal algorithm was proposed by Cole and Vishkin (1989). For a review on the different PRAM algorithms on the subject, see Jájá (1992), Karp and Ramachandran (1990). To reach optimality, these algorithms are based on the same idea: select some elements of the initial list such that the distance between two selected elements is neither too large nor too small; then solve the list ranking on the new linked list of the selected elements with the pointer jumping technique; update the distance of the non selected elements.

In the coarse grained models, several algorithms were proposed, too. These algorithms are also based on the idea described above. Dehne and Song (1996) give an algorithm having, with high probability, $O(k \log p)$ supersteps with $k \leq \log^* n$. Caceres et al. (1997) are the first to propose a deterministic algorithm with $(\log p)$ supersteps and $O(\frac{n}{p})$ local computations. The workload and the total bandwidth are in $O(n \log p)$ that is at a factor $\log p$ from the optimal.

As far as we know, few implementations have been realized, and none of them seems to be portable. Reid-Miller (1994) gives a very detailed implementation on the Cray C-90. She starts from several PRAM algorithms that she modifies for the target platform. If the obtained results are very good, the code is only intended for the C-90. In Dehne and Song (1996), some simulations have been done, but they only give some results on the number of supersteps (that can reach as much as 25). Sibeyn (1997) and Sibeyn et al. (1999) show the results obtained with some of their algorithms on an Intel Paragon. They optimize them for Intel Paragon interconnection networks which have a high degree of independence in the communication links between the processors. The results are good and promising, but no details concerning the implementation (like the language for instance) are mentioned and we think that the implementation might be too specific to the Intel Paragon to lead to efficient results on different architectures where e.g all communications pass through a common communication media.

Given the good performance of the sequential algorithm, it is necessary to be cautious with respect to the overhead in communications and local computations. That is why we proposed a new deterministic algorithm for which communication and total work is only a factor of $\log^* p$ away from the optimal. It requires $O(\log p \log^* p)$ supersteps, $O(\frac{n}{p}\log^* p)$ local computations and the workload/bandwidth are in $O(n\log^* p)$. But the implementation of this algorithm did not give good results: it was quite involved to implement and no speedup was obtained. We proposed then a randomized algorithm that requires $O(\log p)$ communication rounds and $O(n)$ for the total communication cost and local computations. We will see that this algorithm leads to better results, even if the obtained speedups are quite restricted.

## 5.1 A deterministic algorithm

In this section, we give the main outline of a deterministic algorithm (called $\texttt{ListRanking}_k$). Given the scope of this article, we decided to not show all the details of the techniques and the proofs. For a detailed description, see Guérin Lassous and Gustedt (1999).

The idea to solve the problem is the same as the one given in the PRAM algorithms Jájá (1992): it is recursive and based on the building of *k-ruling set*. Such a set $S$ is a sub-set of the elements of the initial list $L$ such that

1. each element $x \in L \setminus S$ has a distance at most $k$ in $L$ to an element $s \in S$,
2. there are not two element $s,t \in S$ that are neighbors in $L$.

First we construct a new linked list made of the elements of the $k$-ruling set. If this list can be stored in the main memory of one processor, then we solve the problem on this new list sequentially, otherwise we iterate the process (we call recursively the building of a $k$-ruling set on this list).

The point here is to reach a small number of supersteps compared to the $O(\log n)$ needed in the PRAM algorithms using the same technique.

**Theorem 2 (Guérin Lassous and Gustedt (1999))** *In the CGM model, if $p \geq 17$, the algorithm* $\texttt{ListRanking}_k$ *can be implemented such that it uses $O(\lceil \log_2 p \rceil\, log_2^* p)$ supersteps and requires an overall bandwidth and processing time of $O(n log_2^* p)$.*

We implemented this algorithm on the two PC clusters and the T3E. This implementation was quite long to realize. Our first tests used the PVM library and the results were rather bad: we obtained no speedups on the PC clusters, whereas the program did not complete on the T3E. We think it came from the PVM library that has difficulties to manage many different buffers. When we switched on MPI, we obtained results on the three machines. But though the execution times were improved when we used more processors, no speedup was obtained on PC clusters and on the T3E the algorithm became a little bit faster than the sequential one from 64 processors on. The single advantage was the handling of very large lists with this algorithm.

## 5.2 A randomized algorithm with better performance

Now we will describe a randomized algorithm for which we will have a better performance than for the deterministic one. It uses the technique of *independent sets*, as described in Jájá (1992).

An *independent set* is a subset $I$ of the list-items such that no two items in $I$ are neighbors in the list. In fact we need such a set $I$ for the algorithm that only has *internal items* i.e. that does not contain a head or tail of one of the sublists. These items in $I$ are 'shortcut' by the algorithm: they inform their left and right neighbors about each other such that they can point two each other directly. Algorithm 2 solves the list ranking problem with this technique in the CGM model.

**Algorithm 2:** IndRanking(L) List Ranking by Independent Sets

```
    Input: Family of doubly linked lists $L$ (linked via $l[v]$ and $r[v]$) and for each item $v$ a distance value
           $dist[v]$ to its right neighbor $r[v]$.
    Output: For each item $v$ the end of its list $t[v]$ and the distance $d[v]$ between $v$ and $t[v]$.
    if $L$ is small then send $L$ to processor 1 and solve the problem sequentially;
    else
independent set    Let $I$ be an independent set in $L$ with only internal items and $D = L \setminus I$;
$\longrightarrow D$        foreach $i \in I$ do Send $l[v]$ to $r[v]$ ;
$\longrightarrow D$        foreach $i \in I$ do Send $r[v]$ and $dist[v]$ to $l[v]$ ;
$I \longrightarrow$        foreach $v \in D$ with $l[v] \in I$ do
            Let $nl[v]$ be the value received from $l[v]$;
            Set $ol[v] = l[v]$ and $l[v] = nl[v]$;
$I \longrightarrow$        foreach $v \in D$ with $r[v] \in I$ do
            Let $nr[v]$ and $nd[v]$ be the values received from $r[v]$;
            Set $r[v] = nr$ and $dist[v] = dist[v] + nd[v]$;
recurse            $IndRanking(D)$;
$\longrightarrow I$        foreach $v \in D$ with $ol[v] \in I$ do Send $t[v]$ and $d[v]$ to $ol[v]$ ;
$D \longrightarrow$        foreach $i \in I$ do
            Let $nt[v]$ and $nd[v]$ be the values received from $r[v]$;
            Set $t[v] = nt[v]$ and $d[v] = dist[v] + nd[v]$;
```

It is easy to see that Algorithm 2 is correct. The following is also easy to see with an argument over the convergence of $\sum_i \varepsilon^i$, for any $0 < \varepsilon < 1$.

**Lemma 1** *Suppose there is an* $0 < \varepsilon < 1$ *for which we ensure for the choices of* $I$ *in "independent set" that* $|I| \geq \varepsilon |L|$. *Then the recursion depth and number of supersteps of Algorithm 2 is in* $O(\log_{1/(1-\varepsilon)} |L|)$ *and the total communication and work is in* $O(|L|)$.

Note that in each recursion round each element of the treated list communicates a constant number of times (at most two times). The values for *small* can be parametrized. If, for instance, we choose *small* equal to $\frac{n}{p}$, then the depth of the recursion will be in $O(\log_{1/(1-\varepsilon)} p)$, and Algorithm 2 will require $O(\log_{1/(1-\varepsilon)} p)$ supersteps. Also the total bound on the work depends by a factor of $1/(1-\varepsilon)$ on $\varepsilon$.

On the other hand the communication does not depend on $\varepsilon$. Every list item is member of the independent set at most once. So the communication that is issued can be directly charged to the corresponding elements of $I$. We think that this is an important feature that in fact keeps the communication costs of any implementation quite low.

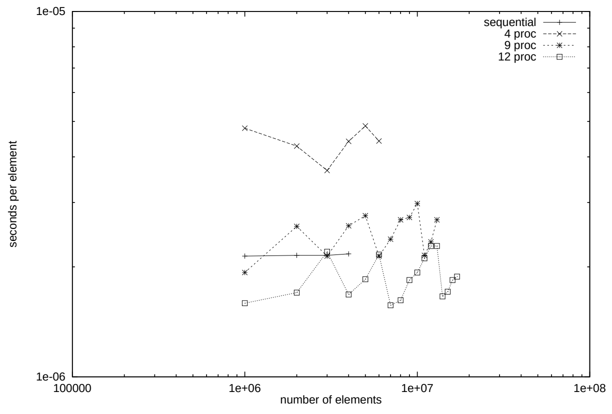
The following lemma ensures the choice of a good (i.e. not too small) independent set. For the proof of this lemma and the implementation side to obtain $I$ see Guérin Lassous and Gustedt (2000).

**Lemma 2** *Suppose every item* $v$ *in list* $L$ *has value* $A[v]$ *that is randomly chosen in the interval* $1, \ldots, K$, *for some value* $K$ *that is large enough. Let* $I$ *be the set of items that have strictly smaller values than its right and left neighbors. Then* $I$ *is an independent set of* $L$ *and with probability approaching* $1$ *we have that* $|I| \geq \frac{1}{4}|L|$.

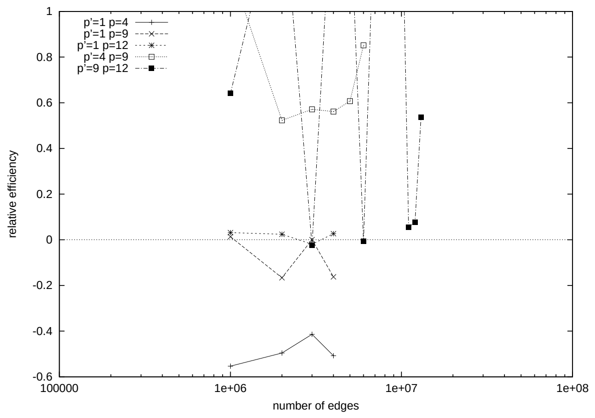
## 5.3 The implementation

We implemented Algorithm 2. The lists are encoded in an array. The array gives the successor for each element: the $i^{th}$ element of the array corresponds to the successor of element $i$ in the list. If we would have only one entire list, the sequential algorithm would consists in going along the array starting from the head of the list, and going from successor to successor. Since we suppose that the groundset can be split into several sublist, we have to do a bit more work than that, but we can get away by visiting each element at most twice.

To generate lists, we used permutations: when a permutation $\Pi$ is generated, each element $\Pi(i)$ of the permutation takes the element $\Pi(i+1)$ in the permutation as successor in the list. The building of the successors array from the permutation requires only one communication round. All the elements are standard integers encoded in 4 bytes.



(a) Execution times



(b) $E^{rel}_{p',p}$

Figure 7: List ranking on the POPC

To not overload the study of the results, we only present the experiments on POPC. For the other results, see Guérin Lassous and Gustedt (2000). Figure 7 gives the execution times *per element* in function of the list size for Algorithm 2. To better cover the different orders of magnitude, both axis are given on a *logarithmic* scale. $p$ varies from 4 to 12, because the memory of the processors is saturated when we use 2 or 3 PC. All the curves stop before the memory saturation of the processors. We start the measures for lists with 1 million elements, because for smaller size, the sequential algorithm performs so well that using more processors is not very useful.

A positive fact that we can deduce from the plots given in Figure 7 is that the execution time for a fixed amount of processors $p$ shows a linear behavior as expected (whatever the number of used processors may be). One might get the impression from Figure 7 that Algorithm 2 deviates a bit from linearity in $n$. But this is only a scaling effect: the variation between the values for a fixed $p$ and $n$ varying is very small (less than $1\mu s$).

We see that from 9 PC the parallel algorithm becomes faster than the sequential one. The parallel execution time decreases also with the number of used PC. The absolute speedups are nevertheless small since for 12 PC for instance the obtained speedup is equal to 1.3.

We see that this algorithm performs well on huge lists. Due to the swapping effects, the sequential algorithm changes its behavior drastically when run with more than 4 million elements. For 5 millions elements, the execution is a little bit bigger than 3000 seconds (which is not far from one hour), whereas Algorithm 2 does it in 9.24 seconds with 12 PC. We also see that to handle lists with 17 millions elements with Algorithm 2 we need 18 seconds.

**Assessment** This problem is the most challenging from the point of view of feasibility: it took us some time to have algorithms really implementable and the implementations were not straightforward. Moreover, as shown previously, this problem is also challenging from the point of view of efficiency: speedups are obtained but are quite restricted.

Nevertheless, it was possible to predict the behavior of the curves and no dramatic deviations from the expectations were obtained. We also showed that it is possible to solve this problem on very large lists. Moreover, as far as we know, it is the first portable code on list ranking that runs on PC clusters as well as on mainframe parallel machines.

## 6 Connected Components for dense graphs

Given $G = (V,E)$ a graph with $n$ vertices and $m$ edges, where each vertex $v \in V$ has a unique label belonging to $[1,n]$. Two vertices $u$ and $v$ are connected if it exists a path that links them. A connected subset of vertices is a sub-set of vertices where each couple of vertices is connected. A *connected component* of $G$ is defined as being a maximal connected subset with respect to inclusion.

Searching for the connected components of a graph is one of the basic graph operations. This problem has been and is still extensively studied. Most of the proposed parallel algorithms are written for the PRAM model. For a review of the different parallel algorithms on the subject see Jájá (1992).

On the other hand, few algorithms for the coarse grained models have been proposed. In Caceres et al. (1997), the first deterministic CGM algorithm is presented. It requires $O(\log p)$ supersteps and is fundamentally based on list ranking.

The algorithm uses an existing PRAM algorithm: it simulates $O(\log p)$ phases of the PRAM algorithm of Shiloach and Vishkin (1983) and then uses a specific CGM algorithm to complete the computations.

According to our experience, it seems that the simulation of PRAM algorithms is complex to implement (lack of *feasibility*), computationally complex in practice (poor *efficiency*) and predictable with difficulty (lack of *predictability*). Moreover, this algorithm is based on the Euler tour computation which uses the list ranking. Seen the difficulties of the list ranking problem alone, we think that this algorithm would currently not lead to an efficient implementation.

### 6.1 The choice of the algorithm

On the other hand, the part of the algorithm that is specific to CGM doesn't have these constraints. It computes the connected components for graphs where $n \leq \frac{m}{p}$, that is to say for graphs that are relatively dense, and does this without the use of list ranking. Therefore we implemented this part of the algorithm.

**Algorithm 3:** Connected Components for relatively Dense Graphs

```
   Data: A graph $G = (V,E)$ given by the values $parent(v)$ for all vertices $v \in V$
   Result: The processor 1 has the connected components of $G$
   $active = p$;
   Call a processor $P_i$ active if $i \leq active$;
   while $active > 1$ do
1      foreach active processor do
           compute the partial connected components of its subgraph;
           keep only edges of spanning tree of these components.
2      foreach active $P_i$ such that $i > \lceil active/2 \rceil$ do
           send its partial connected components to processor $P_{i-\lceil active/2 \rceil}$.
       foreach active $P_i$ such that $i \leq \lceil active/2 \rceil$ do
           rebuild the subgraph according its partial connected components computed at Step 1 and those
           eventually received at Step 2.
       $active = \lceil active/2 \rceil$;
```

Algorithm 3 computes the connected components for graphs with $n \leq \frac{m}{p}$. Each processor has a local memory of $O(\frac{m}{p})$. At the beginning, each processor stores the $n$ vertices and $\frac{m}{p}$ edges of the graph in its memory. At the end of the algorithm, processor 1 stores the connected components of the input graph. At the first step of Algorithm 3, each processor computes the connected components on the part of the graph it stores. Now it may drop all edges that are not needed for a spanning tree of the part that it has in hand, and thus reduces the number of edges to at most $n$. Then each processor numbered in the upper half of the processor range sends the connected components it has just computed to a specific processor in the other half. Only processors numbered in the lower half of the range will continue in the following phase. Then we halven the number and reiterated this process during $\lceil \log p \rceil$ steps. At the end of these $\lceil \log p \rceil$ steps, processor 1 owns the connected components of the whole input graph.

At the first step, each processor computes the connected components on a part of the graph having $n$ vertices and $\frac{m}{p}$ edges. This step requires $O(n+\frac{m}{p})$ local computations. Then in the next steps, the active processors build their new subgraphs from the connected components they have computed and from those they have received. This construction is clearly linear in $n$. Then, the processors compute the connected components of this new subgraph, which is also linear in $n$ because the new subgraph has $2n$ edges. Therefore, Algorithm 3 requires $O(\frac{m}{p}+\lceil \log p \rceil n)$ local computations. At each step, there is exactly one communication step, then Algorithm 3 has $\lceil \log p \rceil$ supersteps. That gives the following theorem:

**Theorem 3** *Algorithm 3 computes the connected components of a graph $G=(V,E)$ in the CGM model with $n$ vertices and $m$ edges such that $n \le \frac{m}{p}$. Each of the $p$ processors requires a memory of $O(\frac{m}{p})$. The algorithm requires $\lceil \log p \rceil$ supersteps and $O(\frac{m}{p}+\lceil \log p \rceil n)$ local computations.*

The use of more processors lowers the execution time only for the first step of the algorithm. Indeed, in this first step, on each processor the connected components are computed on a subgraph with $n$ vertices and $\frac{m}{p}$ edges. So the more processors there are, the faster the computation will be. In the next steps, the connected components are realized on a subgraph with $n$ vertices and $2n$ edges and this whatever the number of processors may be. Moreover, there are $\lceil \log p \rceil$ such steps, and the more processors there are, the longer these steps will last in total. Therefore, when using many processors the initial gain is lost in the other steps, because the density property for the graph is not maintained by the algorithm. We will observe this phenomenon in the experiments.

## 6.2 The implementation

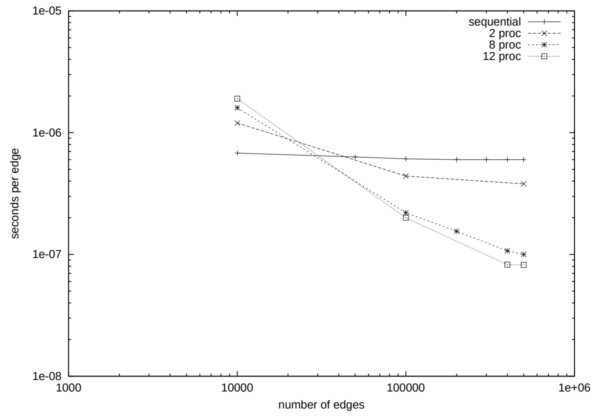
To our knowledge, most of the existing implementations on the search of the connected components are specific to some platforms and thus are not portable. The only portable implementation (Bäumer and Dittrich (1996)) concerns the connected components for two or three dimensional images, that is to say on specific classes of graphs.

Our implementation uses adjacency lists to encode the graphs. This choice is justified by the fact that we can reuse sequential depth first search for the local computation of the connected components. An array $T$ of size $n$ represents the connected components: each vertex $i$ belongs to the component $T[i]$. So if $T[i]=T[j]$ during the algorithm, then $i$ and $j$ belong to the same connected component as found so far and this will hold until the end for the components of the whole graph. During the algorithm, some processors send this array $T$ to their neighbor. The size of this array never changes, so the size of these messages is constantly $n$ for a given graph. Constructing the new subgraph resulting from the merge of the two trees is easily realized in time $n$.

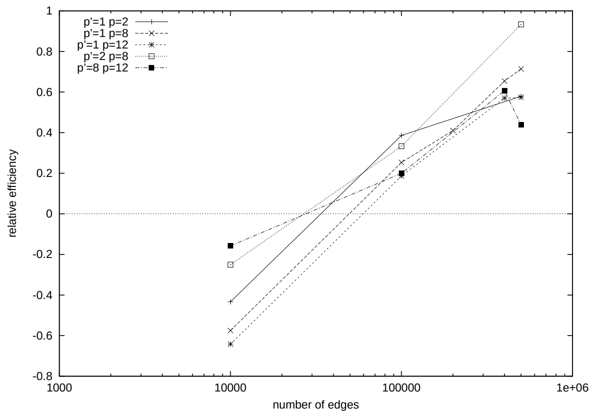
The tests were issued with randomly chosen multi-graphs: two vertices are chosen randomly to form a new edge of the graph. The use of multi-graphs for these tests is not a drawback because the algorithm touches each edge exactly once unless it belongs to the spanning tree.

For this problem, there are two parameters $n$ and $m$ to vary. As the code has the same behavior on the clusters as on the T3E, we only show the results for graphs with 1000 and 10 000 vertices on the PF cluster and with 1000 and 100 000 vertices on the T3E.

**PC clusters** Figures 8 and 9 give the execution times in *seconds per item* with $1,2,8$ and 12 PC for a graph having 1000 and 10 000 vertices, respectively. For $n=1000$, $m$ ranges from 10 000 to 500 000. For $n=10\,000$, $m$ ranges from 10 000 to 36 millions.



(a) Execution times



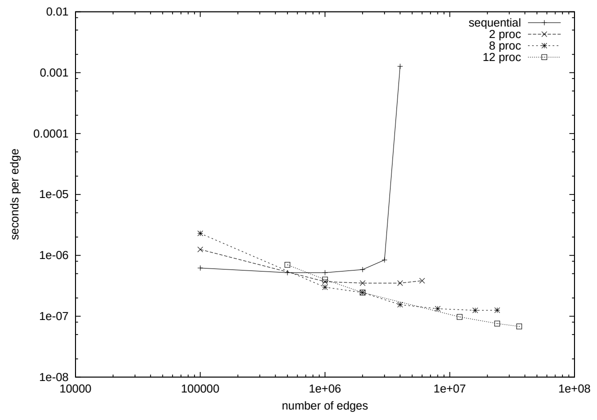
(b) $E^{rel}_{p',p}$

Figure 8: Connected components on the PF, $n = 1000$

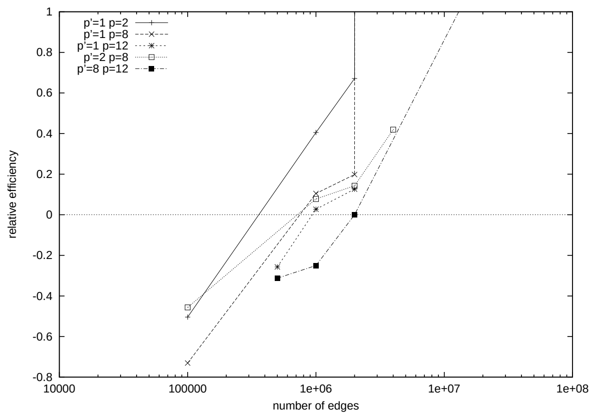
We see that for a fixed $p$ the curves of the parallel program decrease with $m$. If we study the results obtained with $n = 1000$ more precisely, we see that when the graph has more than 50 000 edges then there is always a speedup compared to the sequential implementation, and the more processors we use, the faster the execution is. It is possible to evaluate the minimal number of edges of the graph in order to obtain a speedup: If $p = 2$, then $m$ has to be at least 6 000 and if $p = 8$ then $m \geq 10\,300$.

For the relative efficiency $E^{rel}_{1,p}$, we see that for $n = 1000$, it approaches 0.6 for 2 and 12 PC and 0.75 for 8 PC. The relative efficiency $E^{rel}_{2,8}$ is particulary high: it is well worth to invest in a little bit more than 2 processors.

With $n = 10\,000$, the observed relative efficiencies seem less good than for $n = 1000$. This is due a 'staircase' effect: for two given values $p'$ and $p$ ($p' < p$) the tractability interval for $p'$ processors ends before the 'good end' of the interval for $p$ starts (the 'good end' of an interval being the point where the execution times curve stops decreasing).



(a) Execution times

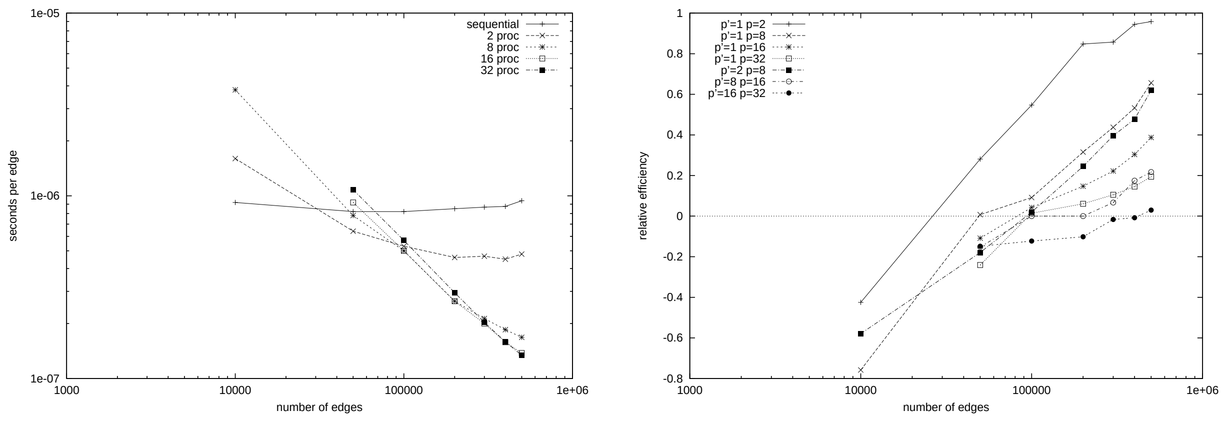


(b) $E^{rel}_{p',p}$

Figure 9: Connected components on PF, $n = 10\,000$

Note that with $n = 10\,000$ it is possible to handle very large graphs by using several PC. In sequential, the PC begins to swap with about 3.2 millions edges. For 4 millions edges, it takes more than 5 000 seconds whereas it takes only a few seconds with several PC. 8 PC can handle up to 24 millions edges, whereas 12 PC can handle until 36 millions edges. The connected component computation on a graph with 36 millions edges can be solved in 2.5 seconds with 12 PC.

Note also, that for this problem, with $n = 10\,000$, the local computation times are greater than the communication times.



(a) Execution times (b) $E^{rel}_{p',p}$

Figure 10: Connected components on the T3E, $n = 1000$

**T3E** We show the results obtained on the T3E. The code is exactly the same as for the PC clusters. We present the results obtained for $n = 1000$ and for $n = 100\,000$. The results for $n = 10\,000$ are the same as for the PC clusters.

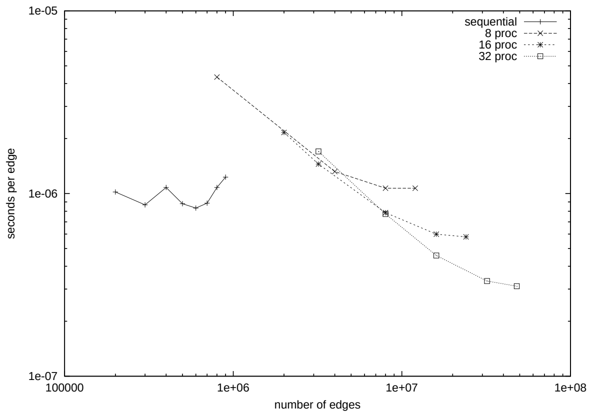
Figure 10 gives the execution times with $1,2,8,16$ and 32 processors for a graph having 1000 vertices. $m$ ranges from $10\,000$ to $500\,000$. For 16 and 32 processors, the curves begins with $m = 50\,000$ because for $m = 10\,000$ the inequality $n \leq \frac{m}{p}$ is not satisfied.

As for the PC clusters, there is a minimal number of edges such that we can expect a speedup. For the T3E, this minimal number is equal to $75\,000$. Until $m = 80\,000$, execution on two processors is the fastest, thereafter with 8 processors until $m = 200\,000$. At that point, it becomes faster with 16 processors. The execution times for 16 and 32 processors are almost alike: we see from the figure that for 32 processors the most efficient range can not be reached. In fact, the last measure is for $500\,000$ edges which corresponds to an average degree of 1000 per vertex. Since we only have 1000 vertices the graph is then just a huge clique. So all acceleration must stop before all 32 processors really can show their muscels. In fact, the relative efficiency $E^{rel}_{16,32}$ turning around 0 shows that it is not worth doubling the number of processors in that case.

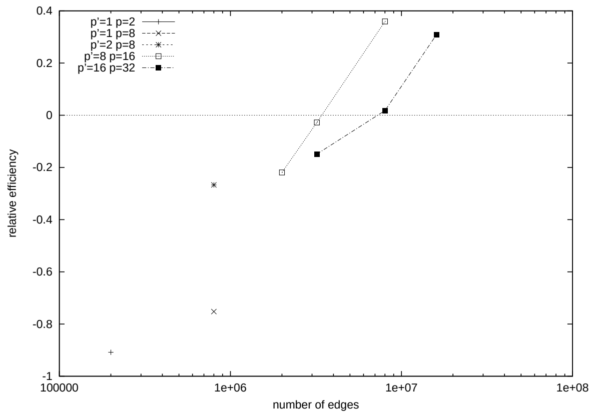
Figure 11 gives the execution times with $1,2,8,16$ and 32 processors for a graph having $100\,000$ vertices. $m$ ranges from $10\,000$ to 48 millions. Each curve stops before the memory of the processors becomes saturated. The PC clusters behave identically for those values of $p$ that are possible.

When the number of processors increases, the share of edges of the input graph decreases for each processor, but the number of rounds increases. Figure 11 shows that the execution time is lowered only for graphs having a very large number of edges. But for large graphs, the relative efficiency $E^{rel}_{16,32}$ turns around 0.2 and compared to graphs with 1000 vertex it is worth using more than 16 processors in that case. It is possible to solve the problem on very large graphs, since 32 processors do it on graphs of 48 millions edges in less than 16 seconds.

Comparing the results for the connected components on the clusters and the T3E, we see that the 12 PC of PF are faster than 16 or 32 processors of the T3E. These differences seem to be due to the different performances of the software components of the platforms such as the quality of the compilers and the dynamic management of the memory.



(a) Execution times



(b) $E^{rel}_{p',p}$

Figure 11: Connected components on the T3E, $n = 100\,000$

## 7 Permutation graphs

Permutation graphs have been intensely studied from an algorithmic point of view as well as from a structural point of view Baker et al. (1971), Colbourn (1981), Golumbic (1980), Pneuili et al. (1971), Spinrad and Valdes (1983). Given a permutation $\Pi$ of the numbers $0,\ldots,n-1$, the permutation graph associated with $\Pi$ is the undirected graph $G = (V,E)$ where $\{i, j\} \in E$ if and only if $i < j$ and $\Pi(i) > \Pi(j)$.

By definition, permutation graphs have a compact encoding of size $n$, namely the permutation $\Pi$. Some graph algorithms require a classical representation of the graph (adjacency matrix or adjacency lists). To apply such an algorithm it is necessary to be able to change from one representation to one other. Spinrad (1994) was the first to mention the problem of computing efficient representations in a more general setting that is the one of partial orders of dimension $d$. Permutation graphs correspond to the case $d = 2$ in that context.

Passing from the permutation to the graph and vice versa is done easily in a sequential time of $O(n^2)$. In parallel, Gustedt et al. (1995) show how to pass from the permutation to the graph in the PRAM and their approach easily translates to the CGM models. This leads to a new compact representation of permutation graphs. The main step of this algorithm is to compute the number of *transpositions* for each value $i = 0,\ldots,n-1$, i.e. the cardinality of

$$\{j \mid i < j \text{ and } \Pi(i) > \Pi(j)\}\,. \tag{19}$$

### 7.1 The choice of the algorithm

We restrict the representation to the computation of these transpositions, see Algorithm 5. In fact, the data that have to be kept for the compact representation correspond to the $\log_2 n$ intermediate tables that are computed here. For more details of the compact representation see Gustedt et al. (1995), Viennot (1996), Guérin Lassous (1999). A first version of this work in a setting quite different has been presented in Guérin Lassous and Morvan (1998).

The algorithm can be informally described as follows: as we use the set of values from 0 to $n-1$, a pivot which divides this set into two sets of equal size can always be found in constant time. At each step, the elements greater than the pivot are put in the right half of the table, whereas the elements less than the pivot are put on the left. The algorithm is then recursively called on the two halves. After $\lceil \log_2 p \rceil$ steps, the data is sorted relatively to the processors: processor 1 has the $\frac{n}{p}$ least data, processor 2 has the following $\frac{n}{p}$,..., processor $p$ has the $\frac{n}{p}$ greatest. Then each processor can run the optimal sequential algorithm to compute the transpostions between elements on the same processor.

**Algorithm 4: Partition** for the processor of number *num*

```
    Data: a part of the permutation $\Pi$ beginning at $x_0$ and of size $size$, a processor range $p_0, \dots, p_0 + r - 1$
          and a pivot $pivot$
    Result: a split of $\Pi$ at $pivot$ and the number of transpostions relative to the split
    foreach $0 \leq x < size$ do
        compare each element $\Pi(x_0 + x)$ with $pivot$;
        if $\Pi(x_0 + x) < pivot$ then $P(x) = 1$ else $P(x) = 0$
    on processors $p_0, \dots, p_0 + r - 1$ compute the prefix sum $S(0) = 0$, $S(x) = \sum_{y=0}^{x-1} P(y)$ for $1 \leq x \leq size$
    foreach $0 \leq x < size$ do
        if $P(x) = 0$ then $transp(x_0 + x) += S(x)$
    foreach $0 \leq x < size$ do
        if $P(x) = 1$ then the element $\Pi(x_0 + x)$ will be send to the processor number $p_0 + S(x)/\frac{n}{p}$;
        else the element $\Pi(x_0 + x)$ will be send to the processor number $p_0 + \lfloor \frac{r}{2} \rfloor + (S(size) - S(x))/\frac{n}{p}$
1   send the data to each processor;
2   receive its data
```

At each step of Algorithm 5, each processor works on a part of the permutation which is given by its size and the beginning of the part in the permutation. In Algorithm 4, the computations of $P$ and of the new position that each element will have in the next phase, like the prefix sum, require $O(\frac{n}{p})$ local computations and one communication round. Lines 1 and 2 of Algorithm 4 correspond to one communication round. The sequential algorithm needs $O(\frac{n}{p} \log_2 \frac{n}{p})$ local computations. Therefore, as there are $\lceil \log_2 p \rceil$ steps, Algorithm 5 requires exactly $\lceil \log_2 p \rceil$ supersteps and

$$O\left(\frac{n}{p}\left(\log_2 \frac{n}{p} + \lceil \log_2 p \rceil\right)\right) = O\left(\frac{n \log_2 n}{p}\right) \tag{20}$$

local computations. The data that are communicated in each round are of size $n$ in total and so the overall communication is in $O(n\lceil \log_2 p \rceil)$. Note that the local computation cost is greater than the communication one. Since the size of the output is also in $O(n \log_2 n)$, both communication cost and local computation of this algorithm are in fact optimal and we should expect a good efficiency.

It is possible to write a CGM algorithm that computes the number of transpositions with a constant number of supersteps, but this algorithm deletes the $\lceil \log_2 p \rceil$ intermediate steps which enable us to store the required data to determine the compact representation.

**Algorithm 5: Transpositions** for the processor of number *num*

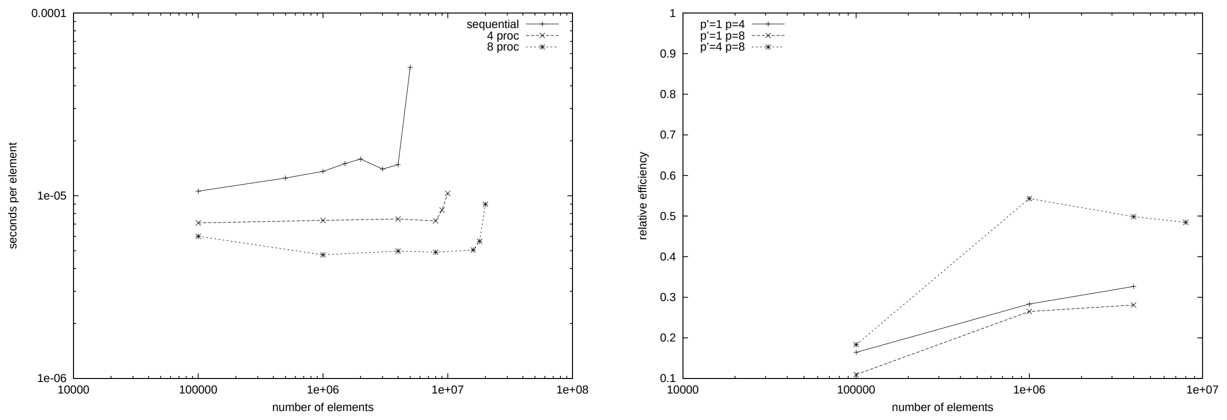
```
    Data: a permutation $\Pi$ of size $n = 2^k$
    Result: the number of transpositions for each value $x = 0, \dots, n - 1$
    $p_0 = 0$; $r = p$; $l = k$;
1   foreach $1 \leq i \leq \lceil \log_2 p \rceil$ do
        $size = 2^l$; $x_0 = \frac{n}{p} \cdot p_0$; $pivot = \frac{n}{p} \cdot (p_0 + \lfloor \frac{r}{2} \rfloor)$;
        Partition($x_0, size, p_0, r, pivot$);
        $l = l - 1$;
        if $num < p_0 + \lfloor \frac{r}{2} \rfloor$ then $r = \lfloor \frac{r}{2} \rfloor$;
        else $p_0 = \lfloor \frac{r}{2} \rfloor$; $r = r - \lfloor \frac{r}{2} \rfloor$;
2   sequentially compute the number of transpostions on its local part.
```

## 7.2 The implementation

We implemented Algorithm 5 on the PC clusters and the T3E. To represent the permutation of size $n$, we used an array: each processor stores an array of size $\frac{n}{p}$ that is a part of the permutation (processor 1 has the elements from 1 to $\frac{n}{p}$ of the permutation, processor 2 has the elements from $\frac{n}{p}+1$ to $2\frac{n}{p}$, etc).

During the supersteps, care must be taken on the receipt order of the data. Indeed, the data sent by processor $i$ have to be put in the received buffer before the data sent by processor $j$ if $i < j$, because for all the elements smaller than the pivot or for those greater, the order they had at the previous step has to be taken into account to compute the number of oriented in-edges for each vertex.

To simplify the implementation and without loss of generality, we assume that p is a power of 2. The generated inputs are random permutations. The elements are unsigned long integers.



(a) Execution times (b) $E^{rel}_{p',p}$

Figure 12: The permutation graph algorithm on the PF

**PC clusters** Figure 12 and Figure 13 show the execution times in *seconds per element* for $1, 4$ and 8 PC. For the two clusters, we do not use more than 8 PC because we have only considered powers of 2 for $p$. For PF, the size of the permutation ranges from 100 000 to 16 millions, whereas for POPC it ranges from 100 000 to 8 millions (due to the memory size of the PC on each cluster). The right end of the curves show the beginning of the swapping effects.

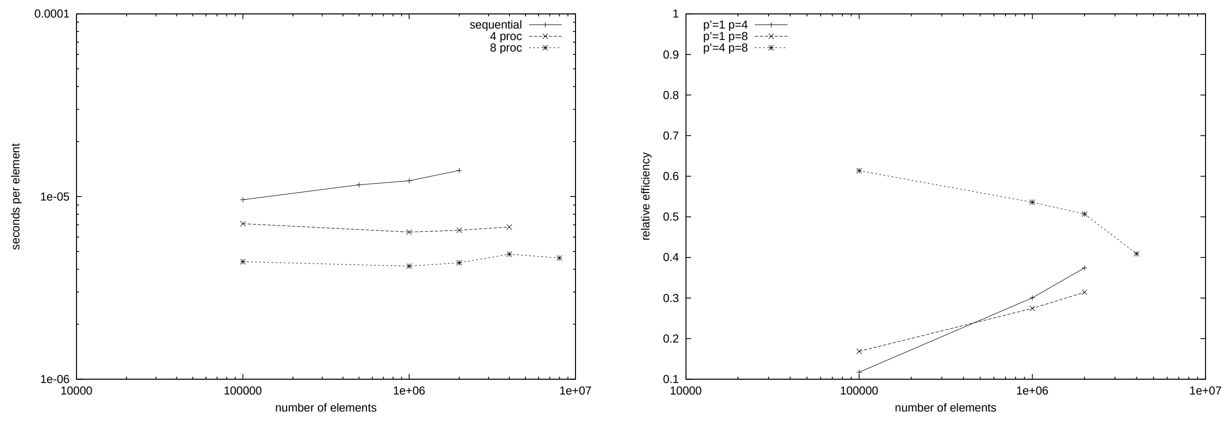
First, the curves have the expected behavior: they are constant in $n$. The execution time is also lowered when we use more processors, as expected. We see that the relative efficiency $E^{rel}_{1,4}$ and $E^{rel}_{1,8}$ are around 0.3 whereas $E^{rel}_{4,8}$ is around 0.5.

Again, it is possible to solve this problem on very large data. For PF, one PC begins to swap after 4 millions data, whereas 4 PC can solve it on a permutation of size smaller than 8 millions and 8 PC do it on a little bit less than 17 millions elements.

For this problem, the local computations time is greater than the communications time. Note that all results are almost alike between the two clusters.

**T3E** The code executed on the T3E is the same as the one used on the clusters. Figure 14 gives the execution times for $1, 4, 8, 16, 32$ and 64 processors. The size of the permutation ranges from 100 000 to 64 millions. Each curve stops before the processor memory is saturated.

As for the PC clusters, the curves have the expected behavior because they are constant in $n$. Moreover, the execution times are lowered when we use more processors, as expected. To simplify the reading of Figure 14 concerning the relative efficiency, we do not give $E^{rel}_{8,16}$ and $E^{rel}_{16,32}$. These two parameters are around 0.8. We

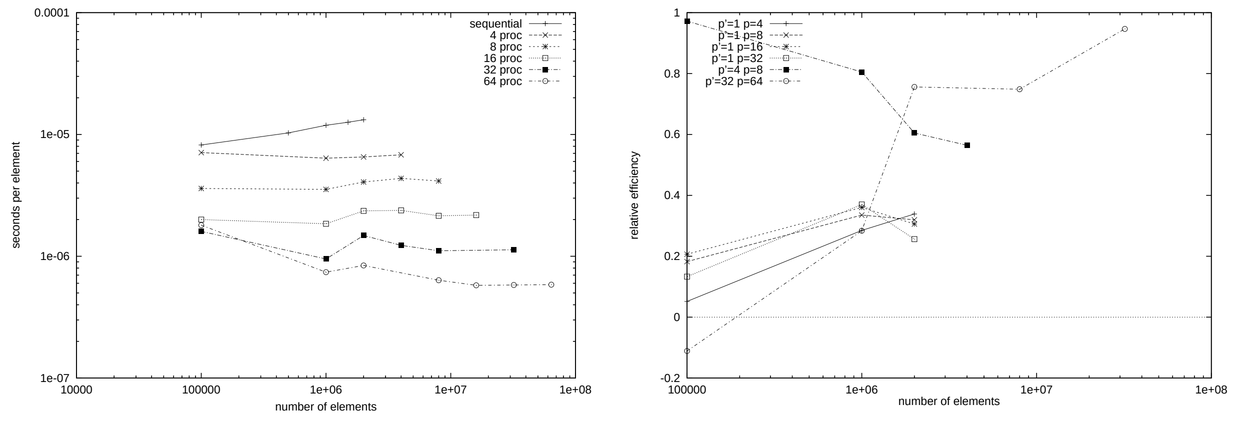


(a) Execution times (b) $E^{rel}_{p',p}$

Figure 13: The permutation graph algorithm on the POPC

see that $E^{rel}_{1,p}$ is 0.2 for all the $p$ used and $E^{rel}_{4,8}$ and $E^{rel}_{32,64}$ are in the range of 0.6 - 1. This shows that it is worth using 64 processors on such algorithms.

We can handle very large permutations with this program. In sequential, one processor becomes saturated after 2 millions elements, whereas 8 processors can handle 8 millions data, 32 processors 32 millions and 64 processors 64 millions. As for the clusters, the local computation time is greater than the communication one.



(a) Execution times (b) $E^{rel}_{p',p}$

Figure 14: The permutation graph algorithm on the T3E

**Assessment** The CGM algorithm is efficient whatever the size of the permutation may be. It is also feasible to solve the problem on very large permutations. As for the search of the connected components, the local computations are predominant compared with the communications. This problem shows that algorithms with $\log p$ supersteps can lead to efficient results.

If we compare, the execution time on 8 PC of POPC and 8 processors of the T3E, we see that it is slightly faster on the T3E than on POPC. As the processors memory size is alike, we see also that they can solve the problem on permutations of the same size.

## 8 Conclusion

We have presented in this article experimental studies of parallel graphs algorithms in the coarse grained models. These studies were useful to point out some points concerning the possibilities and the limits of such implementations and the practical use of these coarse grained models.

### Overview and analysis of the results

First, we are going to study the common points (called invariant) to the different tackled problems, and then we will give the differences.

**Invariants** For all the considered algorithms, we have found some general features independent of the problem. These invariants are the following:

**The curves have the expected behavior:** It was always possible to predict the behavior of the programs in actual platforms in function of the input on the large. Nevertheless, note that the time scale used by the programs could not have always been predicted.

**It is possible to handle very large data:** As we have seen, all the CGM programs can handle data that when processed on a single node would have caused this node to swap on disk. So with CGM we are able to take advantage of the fact that *the network throughput to communicate with other processors is faster than the one for its own disk.*

**Memory saturates before the interconnection network:** For all the studied problems, we never reach the limit of the network bandwidth. Therefore, *the CGM assumption that considers an unlimited bandwidth is justified for these problems*.

**Computation time decreases faster than communication time:** For most of the problems the local computations time is lowered by $\frac{p'}{p}$ when passing from $p'$ to $p$ processors ($p' < p$) because the processor load decreases when we use more processors. This is not the case for the communication time because the total amount of communicated data remains the same. This implies that we obtain *a relative speedup that is less than expected*.

**The two PC clusters don't differ too much:** Whatever the problem may be, the two clusters have very similar results. That shows that the execution of CGM programs does not depend a lot on the interconnection network structure as far as the networks have high rates. In particular, possible conflicts to access the (shared) network are not dominating. A network like Ethernet already has a good enough behavior for this kind of code. *The fact that CGM model does not deal with the conflict problems is well justified in our context*.

**The code is the same on the different platforms:** In all the cases, *it was possible to use the same code on the different platforms*, which proves that the CGM model leads to portable code.

According to these invariants, we conclude that the coarse grained models make the design of code easier, portable and predictive and that we have the additional advantage to handle very large data, at least for graphs.

**Differences** On some points, we noticed some differences in the studied situations. They concern mainly the influence of $p$ the number of processors and the influence of communications and local computations.

Concerning the influence of $p$, the CGM program becomes faster when we use more processors for the sort algorithm, the algorithm concerning the permutation graphs and the search of the connected components on relatively dense graphs having not too many vertices. For the connected components search on dense graphs having many vertices, the more processors we use, the slower the CGM program is.

The split of the communications and the local computations was often useful to understand the results. Especially, we could point out the overhead in some algorithms that lead to some inefficient results. For the connected components on dense graphs with many vertices, we could notice that the local computations to build the subgraphs in each step were too expensive to have an efficient program. Speedups were obtained when the program was run on very large graphs for the connected components problem, whereas we observed speedups when at least 9 PC were used for the list ranking problem.

It is also interesting to note that *in some problems, the local computations time is greater than the communication one*. That says that the limitation of local computations has to be taken into account while designing a parallel algorithm in coarse grained models.

## Partial answers to the initial questions

We are now going to see if this work can give some answers to the questions we raised at the beginning.

- Which parallel model can lead to a feasible, portable, predictable and efficient algorithms and code?

  Given the analysis of the results, it seems that coarse grained models are very promising to that regard. If there is still a lot of work left over, these first steps go towards practical and efficient parallel computation.

- What are the possibilities and the limits of the actual graph handling in parallel environments?

  This work shows that it is now possible to write portable code that handles very large data, and that for some problems, this code is efficient. *The most challenging problem from the view of feasibility and efficiency that we encountered is the list ranking problem*. It is possible that this singularity comes from the specific irregular structure of the problem. Nevertheless, it seems obvious that these results can have an impact on many parallel graph algorithms that are based on list ranking.

The three following questions concern the CGM model itself and were asked in Section 2.

- Is the simplification about the complexities of the different communication routines excessive?

  For all the problems studied, we can answer that it is not.

- Is the number of supersteps $R$ a good parameter for predictions?

  We showed that the results had the expected behaviors. We have seen algorithms for which the number of supersteps varied from some small constant to some logarithmic function in $p$. All the algorithms with a constant number of supersteps were efficient. Concerning algorithms with $\log p$ supersteps. some were quite efficient and others not. We are currently studying algorithms with $p$ supersteps to see wheter they can lead to efficient results or not.

- Can we give good predictions without using the total amount of communication $M$?

  If the goal is to have an idea of the actual behaviors of the programs, then we think that this parameter can be neglected. On the other hand, if we want to describe the time scale of the program and the results more deeply (as to compare local computations and communications), then it is worth to consider the total amount of communication. But note that for all the studied problems the communication size did not exceed the total number of the pure local computations with only one exception: the deterministic list ranking. Therefore its impact is very relative and problem dependent.

We can conclude with these three points that $R$ gives a first idea on the efficiency of the algorithms but to have some acute predictions, it is necessary to compute $M$.

## References

[Adler et al., 1995] Adler, M., Byers, J. W., and Karp, R. M. (1995). Parallel sorting with limited bandwith. In *7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 129–136.

[Alexandrov et al., 1995] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1995). LogGP: Incorporating long messages into the logP model - one step closer towards a realistic model for parallel computation. In *Proc 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105.

[Anderson and Miller, 1988] Anderson, R. and Miller, G. (1988). Deterministic parallel list ranking. In Reif, J., editor, *Proceedings Third Aegean Workshop on Computing, AWOC 88*, pages 81–90. Springer-Verlag.

[Anderson and Miller, 1990] Anderson, R. and Miller, G. (1990). A simple randomized parallel algorithm for list-ranking. *Information Processing Letter*, 33(5):269–279.

[Baker et al., 1971] Baker, K., Fishburn, P., and Roberts, F. (1971). Partial orders of dimension 2. *Networks*, 2:11–28.

[Bäumer and Dittrich, 1996] Bäumer, A. and Dittrich, W. (1996). Parallel Algorithms for Image Processing: Practical Algorithms with Experiments. In *Proc. of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 429–433.

[Bilardi et al., 1996] Bilardi, G., Herley, K., Pietracaprina, A., Pucci, G., and Spirakis, P. (1996). BSP vs LogP. In *Proceedings of 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 25–31.

[Blelloch et al., 1991] Blelloch, G., Maggs, C. L. B., Plaxton, C., Smith, S., and Zagha, M. (1991). A Comparison of Sorting Algorithms for the Connection Machine CM2. In *Symposium on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16.

[Caceres et al., 1997] Caceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputer and BSP. In *Proceedings of the 24th International Colloquium ICALP'97*, volume 1256 of *LNCS*, pages 390–400.

[Colbourn, 1981] Colbourn, C. (1981). On testing isomorphism of permutation graphs. *Networks*, 11:13–21.

[Cole and Vishkin, 1988] Cole, R. and Vishkin, U. (1988). Approximate parallel scheduling, part I: The basic technique with applications t optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, 17(1):128–142.

[Cole and Vishkin, 1989] Cole, R. and Vishkin, U. (1989). Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344–352.

[Cormen et al., 1990] Cormen, T., Leiserson, C., and Rivest, R. (1990). *Introduction to Algorithms*. MIT Press.

[Culler et al., 1993] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. In *Proceeding of 4-th ACM SIGPLAN Symp. on Principles and Practises of Parallel Programming*, pages 1–12.

[Dehne et al., 1993] Dehne, F., Fabri, A., and Rau-Chaplin, A. (1993). Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputer. In *ACM 9th Symposium on Computational Geometry*, pages 298–307.

[Dehne and Götz, 1998] Dehne, F. and Götz, S. (1998). Practical Parallel Algorithms for Minimum Spanning Trees. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 366–371.

[Dehne and Song, 1996] Dehne, F. and Song, S. W. (1996). Randomized parallel list ranking for distributed memory multiprocessors. In Verlag, S., editor, *Proc. 2nd Asian Computing Science Conference ASIAN'96*, volume 1179 of *LNCS*, pages 1–10.

[Dusseau, 1996] Dusseau, A. C. (1996). Fast Parallel Sorting under LogP: Experience with the CM5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805.

[Ferreira, 1996] Ferreira, A. (1996). Parallel and communication algorithms for hypercube multiprocessors. In Zomaya, A., editor, *Handbook of Parallel and Distributed Computing*. McGraw-Hill.

[Ferreira et al., 1998] Ferreira, A., Guérin Lassous, I., Marcus, K., and Rau-Chaplin, A. (1998). Parallel computations on interval graphs using pc clusters: algorithms and experiments. In *Proc. of International EuroPar Conference*, pages 875–886.

[Fortune and Wyllie, 1978] Fortune, S. and Wyllie, J. (1978). Parallelism in Random Access Machines. In *10-th ACM Symposium on Theory of Computing*, pages 114–118.

[Forum, 1993] Forum, M. P. I. (1993). Document for a standard message-passing interface. Technical report, University of Tennessee, Knoxville, Tenn.

[Foster, 1995] Foster, I. (1995). *Designing and building parallel programs*. Addison Wesley.

[Geist et al., 1994] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderman, V. (1994). *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*.

[Gerbessiotis and Siniolakis, 1996] Gerbessiotis, A. V. and Siniolakis, C. J. (1996). Deterministic Sorting and Randomized Median Finding on the BSP model. In *8th ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 223–232.

[Gerbessiotis and Valiant, 1994] Gerbessiotis, A. V. and Valiant, L. G. (1994). Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267.

[Golumbic, 1980] Golumbic, M. C. (1980). *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York.

[Goodrich, 1996] Goodrich, M. (1996). Communication-efficient parallel sorting. In *Proc. of 28th Symp. on Theory of Computing*.

[Goudreau et al., 1996] Goudreau, M., Lang, K., Rao, S., Suel, T., and Tsantilas, T. (1996). Towards Efficiency and Portability: Programming with the BSP Model. In *Proc. of the 8th Annual ACM Symposium in Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12.

[Greiner, 1994] Greiner, J. (1994). A Comparison of Parallel Algorithms for Connected Components. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, pages 16–25.

[Guérin Lassous, 1999] Guérin Lassous, I. (1999). *Algorithmes parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale*. PhD thesis, Université Paris 7.

[Guérin Lassous and Gustedt, 1999] Guérin Lassous, I. and Gustedt, J. (1999). List ranking on a coarse grained multiprocessor. Technical Report 3640, I.N.R.I.A.

[Guérin Lassous and Gustedt, 2000] Guérin Lassous, I. and Gustedt, J. (2000). List ranking on PC clusters. Technical Report 3869, I.N.R.I.A.

[Guérin Lassous and Morvan, 1998] Guérin Lassous, I. and Morvan, M. (1998). Some results on ongoing research on parallel implementation of graph algorithms: the case of permutation graphs. In Arabnia, H. R. et al., editors, *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 325–330.

[Guérin Lassous and Thierry, 2000] Guérin Lassous, I. and Thierry, É. (2000). Generating random permutations in the parallel coarse grained models framework. Working paper.

[Gustedt et al., 1995] Gustedt, J., Morvan, M., and Viennot, L. (1995). A compact data structure and parallel algorithms for permutation graphs. In Nagl, M., editor, *WG '95 21st Workshop on Graph-Theoretic Concepts in computer Science*. Lecture Notes in Computer Science 1017.

[Hsu et al., 1995] Hsu, T.-S., Ramachandran, V., and Dean, N. (1995). Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *Proc. 9th International Parallel Processing Symposium*, pages 106–112.

[Hsu et al., 1997] Hsu, T.-S., Ramachandran, V., and Dean, N. (1997). Parallel Implementation of Algorithms for Finding Connected Components in Graphs. In *Parallel Algorithms : Third DIMACS Implementation Challenge*, volume 30 of *DIMACS Series*, pages 395–416. American Math. Soc.

[Jájá, 1992] Jájá, J. (1992). *An Introduction to Parallel Algorithm*. Addison Wesley.

[Karp and Ramachandran, 1990] Karp, R. and Ramachandran, V. (1990). Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science Volume A: Algorithms and Complexity*, pages 869–942. Elsevier.

[Keeton et al., 1995] Keeton, K. K., Anderson, T. E., and Patterson, D. A. (1995). Logp Quantified: The Case for Low-overhead Local Area Networks. In *Hot Interconnects III: A Symposium on High Performance Interconnects*.

[Krishnamurthy et al., 1997] Krishnamurthy, A., Lumetta, S., Culler, D. E., and Yelick, K. (1997). Connected Components on Distributed Memory Machines. In Bhatt, S., editor, *Third DIMACS Implementation Challenge*, volume 30 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21.

[Krizanc and Saarimaki, 1999] Krizanc, D. and Saarimaki, A. (1999). Bulk synchronous parallel: practical experience with a model for parallel computing. *Parallel Computing*, 25:159–181.

[Kumar et al., 1997] Kumar, S., Goddard, S. M., and Prins, J. F. (1997). Connected-Components Algorithms for Mesh-Connected Parallel Computers. In Bhatt, S., editor, *Third DIMACS Parallel Challenge*. Academic Press.

[Leighton, 1992] Leighton, F. T. (1992). *Introduction to Parallel Algorithms and Architectures: Arrays . Trees . Hypercubes*. Morgan Kaufmann.

[Löwe et al., 1997] Löwe, W., Zimmermann, W., and Eisenbiegler, J. (1997). On Linear Schedules of Tasks Graphs on Generalized LogP-Machines. In LNCS, editor, *Europar'97*, volume 1300, pages 895–904.

[Lumetta et al., 1995] Lumetta, S. S., Krishnamurthy, A., and Culler, D. E. (1995). Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines. In *Proceedings of Supercomputing'95*, San Diego, California.

[Pneuili et al., 1971] Pneuili, A., Lempel, A., and Even, W. (1971). Transitive orientation of graphs and identification of permutation graphs. *Canad. J. math*, 23:160–175.

[Reid-Miller, 1994] Reid-Miller, M. (1994). List ranking and list scan on the Cray C-90. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 104–113.

[Shiloach and Vishkin, 1983] Shiloach, Y. and Vishkin, U. (1983). An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, (1):57–67.

[Sibeyn, 1997] Sibeyn, J. F. (1997). Better Trade-offs for Parallel List Ranking. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 221–230.

[Sibeyn et al., 1999] Sibeyn, J. F., Guillaume, F., and Seidel, T. (1999). Practical Parallel List Ranking. *Journal of Parallel and Distributed Computing*, 56:156–180.

[Spinrad, 1994] Spinrad, J. (1994). Dimension and algorithms. In Bouchité, V. and Morvan, M., editors, *Orders, Algorithms and Applications*, LNCS, pages 33–52. Springer Verlag.

[Spinrad and Valdes, 1983] Spinrad, J. and Valdes, J. (1983). Recognition and isomorphism of two dimensional partial orders. In 10th Coll. on Automata, Language and Programming., number 154 in Lecture Notes in Computer Science, pages 676–686. Springer-Verlag.

[Tanenbaum, 1997] Tanenbaum, A. (1997). *RESEAUX*. Prentice Hall, 3ème edition.

[Valiant, 1990] Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, Vol. 33(8):103–111.

[Viennot, 1996] Viennot, L. (1996). *Quelques algorithmes parallèles et séquentiels de traitement des graphes et applications*. PhD thesis, Université Paris 7.

[Wyllie, 1979] Wyllie, J. (1979). *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY.

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