# List Ranking on PC Clusters

Isabelle Guérin Lassous, Jens Gustedt

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *List Ranking on PC clusters*

Isabelle Guérin Lassous and Jens Gustedt

**N°3869**

Janvier 2000

THÈME 1


ISSN 0249-6399 ISRN INRIA/RR--3869--FR+ENG

# List Ranking on PC clusters

Isabelle Guérin Lassous* and Jens Gustedt†

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n°3869 — Janvier 2000 — 14 pages

**Abstract:** We present two algorithms for the List Ranking Problem in the Coarse Grained Multicomputer model (CGM for short): if $p$ is the number of processors and $n$ the size of the list, then we give a deterministic one that achieves $O(\log p \log^* p)$ communication rounds and $O(n \log^* p)$ for the required communication cost and total computation time; and a randomized one that requires $O(\log p)$ communication rounds and $O(n)$ for the required communication cost and total computation time.

We report on experimental studies of these algorithms on a PC cluster interconnected by a Myrinet network. As far as we know, it is the first portable code on this problem that runs on a cluster. With these experimental studies, we study the validity of the chosen CGM-model, and also show the possible gains and limits of such algorithms for PC clusters.

**Key-words:** list ranking, PC clusters, parallel algorithms

*(Résumé : tsvp)*

* INRIA Rocquencourt, Projet HIPERCOM, F-78153 Le Chesnay, France. Email: Isabelle.Guerin-Lassous@inria.fr – Phone: +33 1 39 63 59 08

† INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

# Classement de listes sur des grappes de PC

**Résumé :** Nous présentons deux algorithmes pour le problème du classement de listes sur un multiprocesseur à gros grain (CGM): si $p$ est le nombre de processeurs et $n$ la taille de la liste, nous proposons un algorithme déterministe avec $O(\log(p)\log^*(p))$ phases de communication et un coût du temps de calcul et de communication de $O(n\log^*(p))$; et un algorithme randomisé qui demande $O(\log p)$ phases de communication et $O(n)$ pour le coût en temps de calcul et en communication. Nous exposons les études expérimentales de ces algorithmes effectuées sur une grappe de PC connecté avec un réseau Myrinet. Selon notre connaissance il s'agit du premier code pour ce problème qui tourne sur une grappe. Avec ces études expérimentales, nous étudions la validité du modèle CGM et montrons aussi les gains et limites possibles d'une telle approche.

**Mots-clé :** classement de listes, grappes de PC, algorithmes parallèles

# 1 Introduction and Overview

**Why List Ranking.** The *List Ranking Problem*, LRP, reflects one of the basic abilities needed for efficient treatment of dynamical data structures, namely the ability to follow arbitrarily long chains for references. In parallel, many graph algorithms use List Ranking as a subroutine. Before handling graph in parallel/distributed, it is useful to know the possibilities and the limits of the LRP in a practical setting.

A linked list is a set of nodes such that each node points to another node called its successor. The LRP consists in determining the rank for all nodes, that is the distance to the last node of the list. In this paper, we work in a more general setting where the list is cut into sublists. Then, the LRP consists in determining for all nodes its distance to the last node of its sublist. Figure 1 gives an example of the LRP. The circled nodes are the last nodes of sublists.

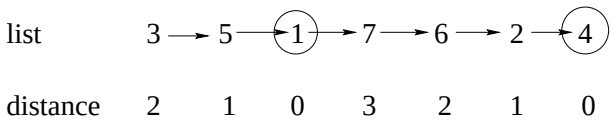


Figure 1: Example of the LRP

Whereas this problem seems (at a first glance) to be easily tractable in a sequential setting, techniques to solve it efficiently in parallel quickly get quite involved and are neither easily implemented nor do they perform well in a practical setting in most cases. Many of these difficulties are due to the fact that until recently no general purpose model of parallel computation was available that allowed easy and portable implementations.

**Some parallel models.** The well studied variants of the LRP, see Karp and Ramachandran (1990) for an overview, are *fine grained* in nature, and written in the PRAM model; usually in algorithms for that model every processor is only responsible for a constant sized part of the data but may exchange such information with any other processor at constant cost. These assumptions are far from being realistic for a foreseeable future: the number of processors will very likely be much less than the size of the data and the cost for communication — be it in time or for building involved hardware— will be at least proportional to the width of the communication channel.

Other studies followed the available architectures (namely interconnection networks) more closely but had the disadvantage of not carrying over to different types of networks, and then not to lead to portable code.

This gap between the available architectures and theoretical models was narrowed by Valiant (1990) by defining the so-called *bulk synchronous parallel* machine, BSP. Based upon the BSP, the model that is used in this paper, the so-called *Coarse Grained Multiprocessor*, CGM, was developed to combine theoretical abstraction with applicability to a wide range of architectures including clusters, see Dehne et al. (1996). It assumes that the number of processors $p$ is small compared to the size of the data and that communication costs between processors are high. One of the main goals for algorithms formulated for that model is to reduce these communication costs to a minimum. The first measure that was introduced was the number of communication rounds: an algorithm is thought to perform local computations and global message exchanges between processors in alternation. This is called rounds. This measure is relatively easy to evaluate but focusing on it alone may hide the real amount of data exchanges between processors, and, in addition the total CPU resources that an algorithm consumes.

The LRP easily kicks out of the game any algorithm that hides some factor in the overall communication or CPU: the obvious sequential algorithm performs so well, that the overhead would be —on the long run— better invested into a more powerful sequential machine or just more memory, be it RAM or disk. So care must be taken that only a small overhead arises in communication and CPU time.

**Previous algorithms in the coarse grained models.** The first proposed algorithm is a randomized algorithm by Dehne and Song (1996) that performs in $O(\log p \log^* n)$ rounds with a workload (total number of local

steps)/total communication of $O(n \log^* n)$ ($log^* n = \{i \mid log^{(i)} n \leq 1\}$). Then, Caceres et al. (1997) gives a deterministic algorithm that needs $O(\log p)$ rounds and a workload/total communication of $O(n \log p)$.

**Previous practical work.** Very few articles deal with the implementation sides of LRP. Reid-Miller (1994) presents a specific implementation optimized for the Cray C-90 of different PRAM algorithms that give good results. In Dehne and Song (1996), some simulations have been done, but they only give some results on the number of communication rounds. Sibeyn (1997) and Sibeyn et al. (1999) give several algorithms for the LRP with derived PRAM techniques and new ones. They fine-tune their algorithms according to the features of the interconnection network the Intel Paragon. The results are good and promising, since more than 10 processors are used.

In all these works, the implementations are specific to the interconnection network of the target machine and do not seem to be portable. For instance, Sibeyn (1997) and Sibeyn et al. (1999) give no details on the chosen language.

**This paper.** The use of PC clusters interconnected by high performance local networks is one of the major current trends in parallel/distibuted computing. If a lot of works have been done on system-level and programming environnement issues, little effort has been undertaken to the algorithmic level for this kind of network.

In this paper, we address ourselves to the problem of writing portable, efficient and predictive deterministic code for LRP running on PC clusters. We considered several goals:

- Is it feasible to solve the LRP on PC clusters, because no previous studies on LRP use clusters?
- Is it possible to write portable code, therefore not optimized for PC clusters and that is nevertheless efficient on this kind of machines? As far as we know, it would be the first portable proposed code on the subject and running on PC clusters.
- Are we able to predict the behavior of the program on such machines? If it is the case, then we can say that the chosen model is well adapted to PC clusters for this kind of problems.

We do not pretend to have the best implementation of the LRP on PC clusters, but we tried to respect the three given goals at the same time. Especially, the implementation was done carefully –as we think– to have the best possible results without loosing at portability level.

As explained previously, care must be taken that only a small overhead arises in communication and CPU times. The deterministic algorithm of Caceres et al. (1997) is a factor $\log p$ away from optimality, whereas Dehne and Song (1996) is a cator $\log^* n$. Therefore, in this paper, we also give two algorithms for the LRP:

- A deterministic algorithm that performs in $O(\log p \log^* p)$ communication rounds and the overall workload and communication costs are in $O(n \log^* p)$,
- A randomized algorithm that requires $O(\log p)$ communication rounds and $O(n)$ for workload/communication costs.

Consider Table 1 for an overview of these different measures for the mentionned algorithms. The algorithm of Sibeyn (1997), not expressed in the CGM model, has different measures. We mention them for its practical interest.

To verify that the proposed algorithm is of practical relevance, we implemented it on a PC cluster with a Myrinet interconnection network.

The paper is organized as follow: we give the main features of the CGM model in Section 2. Next, we present a deterministic algorithm for solving the LRP in Section 3, and then we present a randomized algorithm in Section 4. Section 5 concerns the results of the implementations on a PC cluster with a Myrinet interconnection network. Finally, we conclude in verifying whether our three goals given previously have been reached or not.

| reference | comm. rounds | CPU time & communication |
|---|---|---|
| Dehne and Song (1996) | $\log p \quad \log^* n$ | $n \quad \log^* n \quad$ rand |
| Caceres et al. (1997) | $\log p$ | $n \quad \log p \quad$ det |
| we | $\log p \quad \log^* p$ | $n \quad \log^* p \quad$ det |
| we | $\log p$ | $n \quad$ rand |
| Sibeyn (1997) | | $n \quad$ aver |

Table 1: Comparison of our results to previous work. *O*-notation omitted.

## 2 The CGM model for parallel/distributed computation

The basic ideas that characterize the CGM model are:

**uniformity** A CGM consists of a small number $p$ of uniform processing units (*processors*). 'Small' here means magnitudes smaller than the size $n$ of the input data.

**code sharing** All processors execute the same program.

**simultaneous alternation** The overall execution of an algorithm alternates concurrently between phases with only computations local to the processors and communications between those processors. This is called *rounds*.

**implicit synchronization** Synchronization of the different processors is only done implicitly during the communication rounds.

Besides its simplicity, this approach also has the advantage of allowing design of algorithms for a large variety of existing hardware and software platforms, and especially clusters. It does this without going into the details and special characteristics of such platforms, but gives predictions in terms of the number of processors $p$ and the number of data items $n$ only.

Usually the number of rounds is taken as the main cost of a CGM-algorithm and all efforts are made to minimize this value. If not possible to bind it by a constant, at least a slowly growing function depending on $p$ only is desired. But, as already discussed in the introduction, this may fall to short, since the main bottleneck for many parallel computation is the overall communication between the processors.

Nevertheless, we chose this model for at least two reasons:

- Given its features, algrithms written in this model should be portable on different parallel/distributed machines and then implementable on clusters whatever their interconnection network may be,
- It is simple and allows to concentrate on the difficulties of the LRP in contexts like clusters.

## 3 A Coarse Grained Algorithm for List Ranking

### 3.1 Basics of the List Ranking Problem

The algorithm we propose to solve the LRP is based on two ideas given in PRAM algorithms. The first and basic technique, called *pointer jumping*, was mentionned by Wyllie (1979). Algorithm 1 reflects the pointer jumping technique.

It is easy to see that `Jump` is correct and that Invariant A is always satisfied if it was true at the beginning. When called as $\texttt{Jump}(L, \{t\})$ where $L$ is the list in question and $t$ is the end of the list, it performs the while-loop at most $\log|L|$ times and is thus a natural candidate for a parallelization. But we can even give stronger statement than that.

**Algorithm 1:** `Jump`

**Input**: Set $R$ of $n$ linked items $e$ with pointer $e.\mathsf{succ}$ and distance value $\mathsf{dist}$ and subset $S$ of $R$ of marked items.

**Task**: Modify $e.\mathsf{succ}$ and $e.\mathsf{dist}$ for all $e \in R \setminus S$ s.t. $e.\mathsf{succ}$ points to the nearest element $s \in S$ according to the list and s.t. $e.\mathsf{dist}$ holds the sum of the original $\mathsf{dist}$ values along the list up to $s$.

**while** *there are* $e \in R \setminus S$ *s.t.* $e.\mathsf{succ} \notin S$ **do**
  **for** *all such* $e \in R$ **do**
A  **Invariant**: Every $e \notin S$ is linked to by at most one $f.\mathsf{succ}$ for some $f \in R$.
1  Fetch $e.\mathsf{succ} \to \mathsf{dist}$ and $e.\mathsf{succ} \to \mathsf{succ}$;
2  $e.\mathsf{dist}$ += $e.\mathsf{succ} \to \mathsf{dist}$;
3  $e.\mathsf{succ} = e.\mathsf{succ} \to \mathsf{succ}$

**Proposition 1** *Let $R$ and $S$ be inputs for `Jump`, and let $\ell$ be the maximum length of an element $x \in R \setminus S$ to the next element $s \in S$. Then `Jump`$(R,S)$ executes the while-loop $\lceil \log_2 \ell \rceil$ times.* □

Because of Invariant A we see that `Jump` can easily be realized on a CGM: just let each processor performs the statements inside the while-loop for the elements that are located at it. The invariant then guarantees that each processor has to answer at most one query for each of its items issued by line 1. So none of the processors will be overloaded at any time.

**Corollary 1** *`Jump` can be implemented in the CGM model such that it runs for $R$, $L$ and $\ell$ as above in $O(\lceil \log_2 \ell \rceil)$ communication rounds and requires an overall bandwith and processing time of $O(n \lceil \log_2 \ell \rceil)$.*

Corollary 1 shows that the CGM pointer jumping algorithm performs $O(\lceil \log_2 n \rceil)$ rounds to solve the LRP and needs $O(n \lceil \log_2 n \rceil)$ workload/total communications. According to the CGM model, this algorithm is unlikely to lead to efficient code. Therefore, we have to used other techniques to solve the LRP. Nevertheless, the pointer jumping technique is used as a subroutine of our algorithm.

### 3.2 The Algorithm

The second used PRAM techniques is a *k-ruling set*, see Cole and Vishkin (1989). Such a $k$-ruling set $S$ is a subset of the items in the list $L$ s.t.

1. Every item $x \in L \setminus S$ is at most $k$ links away from some $s \in S$.
2. No two elements $s,t \in S$ are neighbors in $L$.

The PRAM idea is to reduce the initial list by building a $k$-ruling set of size not too large and not too small; and then to solve the LRP on the list made of the elements of the $k$-ruling set with the pointer jumping technique; and finally computing the List Ranking on the elements of the intial list not selected in the $k$-ruling set.

Algorithm 2 implements this technique in the CGM model to solve the LRP: the goal is to reduce the size of the list with the build of a $k$-ruling set; since the new list can be stored in the main memory of one processor, then the problem is solved sequentially; otherwise the algorithm is called recursively. The point, here, is to have a small number of rounds (a slowy growing function depending on $p$ for instance), compared to the $O(\log n)$ rounds needed in the PRAM algorithms using the same techniques. At the same time, we have to pay attention to the workload, as the total communication bandwidth.

**Proposition 2** *Suppose we have an implementation `Ruling`$_k$ of a $k$-ruling set algorithm then `ListRanking`$_k$ can be implemented on a CGM such that it uses $O(\lceil \log_2 k \rceil)$ communication rounds per recursive call and requires an overall bandwith and processing time of $O(n \lceil \log_2 k \rceil)$ when not counting the corresponding measures that calls to `Ruling`$_k$ need.*

**Algorithm 2:** `ListRanking`$_k$

**Input**: $n_0$ total number of items, $p$ number of processors, set $L$ of $n$ linked items with pointer succ and distance value dist.

**Task**: For every item $e$ set $e$.succ to the end of its sublist $t$ and $e$.dist to the sum of the original dist values to $t$.

**if** $n \leq n_0/p$ **then**

1 Send all data to processor 0 and solve the problem sequentially there.

**else**

2 Shorten all consecutive parts of the list that live on the same processor. ;

3 **for** *every item* $e$ **do** $e$.lot = processor-id of $e$;

4 $S = \texttt{Ruling}_k(p-1, n, L)$;

5 $\texttt{Jump}(L, S)$;

6 **for** *all* $e \in S$ **do** set $e$.succ to the next element in $S$ ;

7 $\texttt{ListRanking}_k(S)$;

8 $\texttt{Jump}(L, \{t\})$;

**Proof:** The only critical parts for these statements are lines 5, 6 and 8. Corollary 1 immeadiately gives an appropriate bound on the number of communication rounds for line 5. After line 5, since every element $L \setminus S$ now points to the next element $s \in S$ line 6 can easily be implemented with $O(1)$ communication rounds. After comming up from recursion every $s \in S$ is linked to $t$, so again we can perform line 8 in $O(1)$ communication rounds. So in total this shows the claim for the number of communication rounds.

To estimate the total bandwidth and processing time observe that each recursive call is called with at most half the elements of $L$. So the overall resources can be bounded from above by a standard domination argument □

We use *deterministic symmetry breaking* to obtain a $k$-ruling set, see Jájá (1992). The inner (and interesting) part of such a $k$-ruling algorithm is given in Algorithm 3.

**Algorithm 3:** `RuleOut`

**Input**: item $e$ with fields lot, succ and pred, and integers $l_1$ and $l_2$ that are set to the lot values of the predecessor and successor, resp.

**if** $(e.lot > l_1) \wedge (e.lot > l_2)$ **then**

1 Declare $e$ *winner* and force $e$.succ and $e$.pred *looser*.

**else**

2 **if** $l_1 = -\infty$ **then** Declare $e$ *p-winner* and suggest $e$.succ *s-looser* ;

3 **else**

Let $b_0$ be the most significant bit, for which $e$.lot and $l_1$ are distinct;

Let $b_1$ the value of bit $b_0$ in $e$.lot;

$e$.lot $:= 2 * b_0 + b_1$.

Here an item $e$ decides whether or not it belongs to the ruling set by some local value $e$.lot according to two different strategies. By a *winner* (with $e$.lot set to $+\infty$) we denote an element $e$ that has already be choosen to be in the ruling set, by a *looser* (with $e$.lot set to $-\infty$) we denote an element $e$ that certainly not belongs to the ruling set. For technical reasons we also have two auxiliary characterizations, *p-winner*, potential winner, and *s-looser*, suggested looser. The algorithm will guarantee that any of these two auxiliary characterizations will only occur temporarily. Any *p-winner* or *s-looser* will become either *winner* or *looser* in the next step of the algorithm. We give some explanations on specific lines:

line 1 First $e$ looks whether this value is larger than the values for its two neighbors in the list. If this is so it belongs to the ruling set.

line 2 If this is not the case but its predecessor in the list was previously declared *looser* it declares itself a *p-winner* and its successor an *s-looser*.

line 3 The remaining elements update their value $e.\mathsf{lot}$ by basically choosing the most significant distinct bit from the value of the predecessor.

Line 2 is necessary to avoid conflicts with regard to Property 2 of a $k$-ruling set. Such an element can only be a winner if its successor has not simultanously decided to be a *winner*.

Line 3 ensures that –basically– the possible ranges for the values $e.\mathsf{lot}$ goes down by $\log_2$ in each application of `RuleOut`. The multiplication by 2 (thus a left shift of the bits by 1) and addition of the value of the chosen bit is done to ensure that neighboring elements always have different values $e.\mathsf{lot}$.

The whole procedure for a $k$-ruling set is given in Algorithm 4.

**Algorithm 4:** $\texttt{Ruling}_k$

**Constants**: $k > 9$ integer threshold

**Input**: Integers $q$ and $n$, set $R$ of $n$ linked items $e$ with pointer $e.\mathsf{succ}$, and integer $e.\mathsf{lot}$

**Output**: Subset $S$ of the items s.t. the distance from any $e \notin S$ to the next $s \in S$ is $< k$.

A **Invariant**: Every $e$ is linked to by at most one $f.\mathsf{succ}$ for some $f \in R$, denote it by $e.\mathsf{pred}$.

B **Invariant**: $q \geq e.\mathsf{lot} \geq 0$.

1 $\mathsf{range} := q$;

**repeat**

C **Invariant**: If $e.\mathsf{lot} \neq -\infty$ then $e.\mathsf{lot} \neq e.\mathsf{succ} \to \mathsf{lot}$.

B' **Invariant**: If $e.\mathsf{lot} \notin \{+\infty, -\infty\}$ then $\mathsf{range} \geq e.\mathsf{lot} \geq 0$.

**for** *all* $e \in R$ *that are neither* winner *nor* looser **do**

2 Communicate $e$ and the value $e.\mathsf{lot}$ to $e.\mathsf{succ}$ and receive the corresponding values $e.\mathsf{pred}$ and $l_1 = e.\mathsf{pred} \to \mathsf{lot}$ from the predecessor of $e$;

3 Communicate the value $e.\mathsf{lot}$ to $e.\mathsf{pred}$ and receive the value $l_2 = e.\mathsf{succ} \to \mathsf{lot}$;

4 `RuleOut`$(e, l_1, l_2)$;

5 Communicate new *winners*, *p-winners*, *loosers* and *s-loosers*;

6 **if** *e is* p-winner $\wedge$ *e is not* looser **then** declare *e winner* **else** declare *e looser*;

7 **if** *e is* s-looser $\wedge$ *e is not* winner **then** declare *e looser*;

8 Set $e.\mathsf{lot}$ to $+\infty$ for *winners* and to $-\infty$ for *loosers*;

9 $\mathsf{length} := 2\mathsf{range} - 1$; $\mathsf{range} := 2\lfloor \log_2 \mathsf{range} \rfloor + 1$;

**until** *(R contains only elements that are* winners *or* loosers*)* $\vee$ $(\mathsf{length} < k)$;

**return** *Set S of winners.*

**Proposition 3** $\texttt{Ruling}_k$ *can be implemented on a CGM such that it uses* $O(\log^* q)$ *communication rounds and requires an overall bandwidth and processing time of* $O(n \log^* q)$. *Moreover, k is the maximum distance of an element* $x \in R \setminus S$ *to the next element* $s \in S$.

**Proof:** Invariant C is always satisfied if it was true at the beginning: after line 4, neighboring elements have always different values $e.\mathsf{lot}$. After this line only winners and loosers modify their value $e.\mathsf{lot}$. Or according to the algorithm *RuleOut* and lines 6, 7 and 8, if $e$ is a winner then $e.\mathsf{succ}$ is a looser therefore the two values $e.\mathsf{lot}$ are different. Because of Invariant C, we can see that two elements of $S$ are not neighbors in $R$.

$\mathsf{range}$ is the maximum $e.\mathsf{lot}$ value that an element of $R$ may have. Let $b$ be the number of bits used to represent the value $e.\mathsf{lot}$. When considering only non-winner and non-looser elements, after line 4, the maximum possible value for $e.\mathsf{lot}$ is $2(b-1)-1$ that is $2\lfloor \log_2 \mathsf{range} \rfloor + 1$. Moreover, the number of bits used to represent $e.\mathsf{lot}$ is $\lceil \log_2 b \rceil + 1$. The number of bits decreases as long as $b > \lceil \log_2 b \rceil + 1$, that is $b > 3$. By recurrence, it is easy to show that, if $b_i$ is the number of bits to represent $e.\mathsf{lot}$ at step $i$, and $\left\lceil \log_2^{(i)}(q) \right\rceil \geq 2$,

then $b_i \leq \left\lceil \log_2^{(i)}(q) \right\rceil + 2$. Then, after $m = log_2^* q$ steps, $b_m \leq 3$ ($\left\lceil \log_2^{(m)}(q) \right\rceil \leq 1$ with the definition of $log_2^*$). Therefore, after $log_2^* q + 1$ steps, the maximum value range is always 5. Then, length which is the maximum length of an element $x \in R \setminus S$ to the next element $s \in S$, is equal to 9. Winners and loosers do not modify the values of range and length. Therefore, if it exists non-winner or non-looser elements, the loop is repeated at most until length be equal to 9 that is at most $\log^* q + 1$.

If the loop is exited when $R$ contains only winner and looser elements then we claim that the distance between two winners in $R$ is at most 3. Indeed, all the loosers have at least one neighbor that is a winner. An element $e$ can become a looser in two ways: either it has a winner neighbor, either it is a *s-looser* and it is not a winner (line 7). Or if it is a *s-looser*, its predecessor $f$ is a *p-winner* (line 2 of `RuleOut`). Moreover, $f$ is not a looser, because $f$.pred is a looser (line 2 of `RuleOut`) and $f$.succ $= e$ is not a winner by hypothesis. Then $f$ is a winner. Therefore the distance between two elements in $S$ is at most 3. Moreover the number of iterations of the loop is bounded by $(\log^* q + 1)$ and the maximum distance of an element $x \in R \setminus S$ is at most $2 (\leq k)$.

We can perform $O(1)$ communication rounds in lines 2, 3 and 5. So the total communication rounds number is bounded by $O(\log^* q)$. □

**Proposition 4** *If $p \geq 17$, $\texttt{ListRanking}_k$ can be implemented on a CGM such that it uses $O(\lceil \log_2 p \rceil \, log_2^* p)$ communication rounds and requires an overall bandwidth and processing time of $O(n log_2^* p)$.*

**Proof:** In each phase of $\texttt{ListRanking}_k$, $\texttt{Ruling}_k$ is called with the parameter $q$ equal to $p-1$. According to Proposition 2, $k$ is at most equal to 9, therefore if $p \geq 17, \lceil \log_2 k \rceil \leq log_2^* p$. Then, $\texttt{ListRanking}_k$ uses $O(log_2^* p)$ communication rounds per recursive call.

At each recursive call, the number of elements of $S$ is at most half the elements of $L$. After $\lceil \log_2 p \rceil$ steps, $n \leq \frac{n_0}{p}$. Therefore $\texttt{ListRanking}_k$ uses $O(\lceil \log_2 p \rceil \, log_2^* p)$ communication rounds. With the same argument, the overall bandwidth and processing time is bounded by $O(n log_2^* p)$. □

# 4 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, as shown in Section 5. 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 5 solves the LRP with this technique in the CGM model.

It is easy to see that Algorithm 5 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 5 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 5 will require $O(\log_{1/(1-\varepsilon)} p)$ communication rounds. Also the total bound on the work depends by a factor of $1/(1-\varepsilon)$ from $\varepsilon$.

The communication on the other hand 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.

So it remains to see, how we can ensure the choice of a good (ie not too small) independent set.

**Algorithm 5:** 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 $d[v] = dist[v] + nd[v]$;

**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 $I$ 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 $|I| \geq \frac{1}{4}|L|$.*

**Proof:** Clearly $I$ is an independent set. For the probability observe that if we chose $A[v]$ at random the probability that a neighbor $w$ has a random value $A[w]$ that is greater than $A[v]$ is $\frac{K-A[v]}{K}$ and that both neighbors have a greater value is

$$\left(\frac{K-A[v]}{K}\right)^2 = \frac{(K-A[v])^2}{K^2}. \tag{1}$$

Since the expected value for $A[v]$ is $K/2$ this gives an overall probability of

$$\frac{(K-K/2)^2}{K^2} = \frac{(K/2)^2}{K^2} = 1/4. \tag{2}$$

□

For the implementation side of Algorithm 5 we have to be careful not to spend too much time for

1. initializing the recursion, or
2. chosing (pseudo) random numbers.

In fact, we ensure 1 by an array that always holds the active elements, ie those elements that were not found in sets $I$ in recursion levels above. By that we do not have to copy the list values themselves and the recursion does not create any additional communication overhead.

For 2, we ensure at the beginning that the list is distributed randomly on the processors. Then every item $v$ basically uses its own (storage) number as value $A[v]$. To ensure that these values are still sufficiently independent in lower levels of recursion we choose for each such level $R$ a different large number $N_R$ and set $A[v] = N_R \cdot v \bmod K$.

# 5 Implementation

Our main tests for the two algorithms took place on a PC cluster[1]. It consists of 12 *PentiumPro* 200 PC with 64 MB memory each. The PC are interconnected by a *Myrinet*[2] network of 1.28 Gb/s and with 5 $\mu$s latency.

The implementation of the algorithm was done –as we think– quite carefully in C++ and based on MPI, one well-known library for message passing between processes. The cluster is equipped with the Linux OS, the g++ compiler from the EGCS project and the MPI-BIP implementation (that is an implementation of MPI over Myrinet). The use of C++ allowed us to actually do the implementation on different levels of abstraction:

1. one that interfaces our code to one of the message passing libraries,
2. one that implements the CGM model itself, and
3. the last that implements this specific algorithm.

One of our itentions for this hierarchical design is to replace message passing in 1 by shared memory later on. This is out of the scope of the study here, but this shows our wish to have portable code.

This later goal seems to be well achieved, since we have been able to run the code on a large variety of architectures: a PC cluster, SUN workstations, an SGI *Origin 2000* and a Cray *T3E*. There, the general outlook of the curves looks similar, certainly that the constant factors are dependent on the architecture.

There are many different aspects to consider when discussing an implementation of a parallel algorithm. The most prominent among these is certainly the gain of effective execution time that one expects. First, we will present the execution times obtained for the two algorithms. Then, we will show in the following that here the CGM model allows a relatively good prediction of what we may expect. Finally, we will show that PC clusters are also good candidates for handling very large lists.

## 5.1 Execution time.

Figure 2 gives the execution times *per element* in function of the list size for Algorithm 2, whereas Figure 3 is for Algorithm 5. To cover the different orders of magnitude better, both axis are given on a *logarithmic* scale. The lists were generated randomly with the use of rendom permutations.

For each list size, the program was run (at least) 10 times and the result is the average of these results. For a fixed list size, very small variations in time could be noted.

$p$ varies from 2 to 12 for Algorithm 2 and from 4 to 12 for Algorithm 5. Algorithm 5 is more greedy in memory, therefore 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.

According the chosen algorithm, absolute speedups (ratio between the sequential algorithm and the parallel one) can be obtained or not. Algorithm 2 is always slower than the sequential one. Nevertheless, the parallel execution time decreases with the number of used PC.

For Algorithm 5, 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.

If we compare the two algorithms, we can note that:

- The use of a larger amount of processors in all cases lead to an improvement on the real execution times of the parallel program, which proves the scalability of our approach,

[1] http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html
[2] http://www.myri.com/

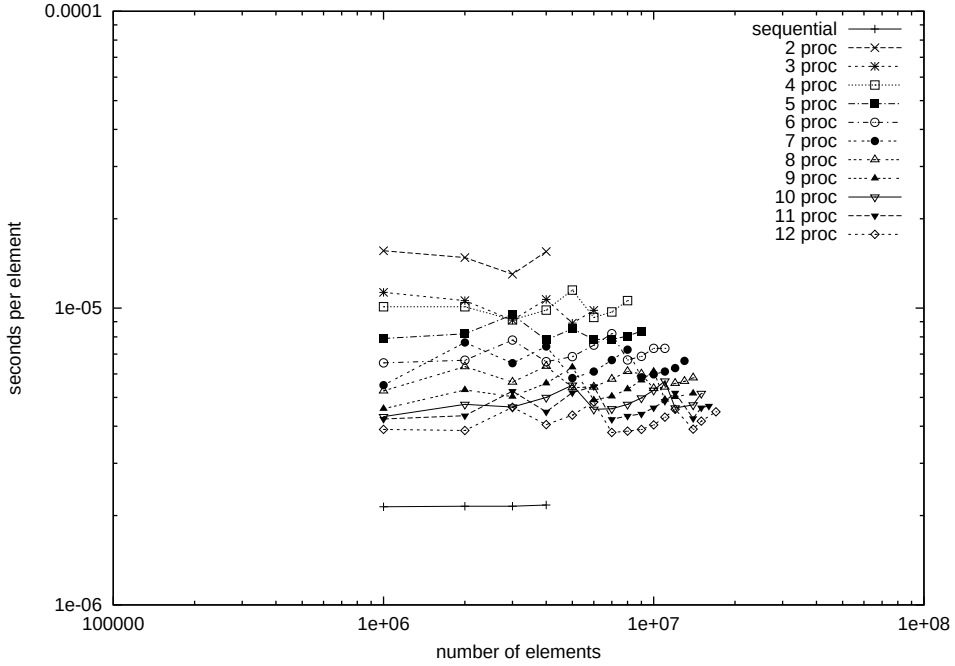


Figure 2: Execution times per element for Algorithm 2

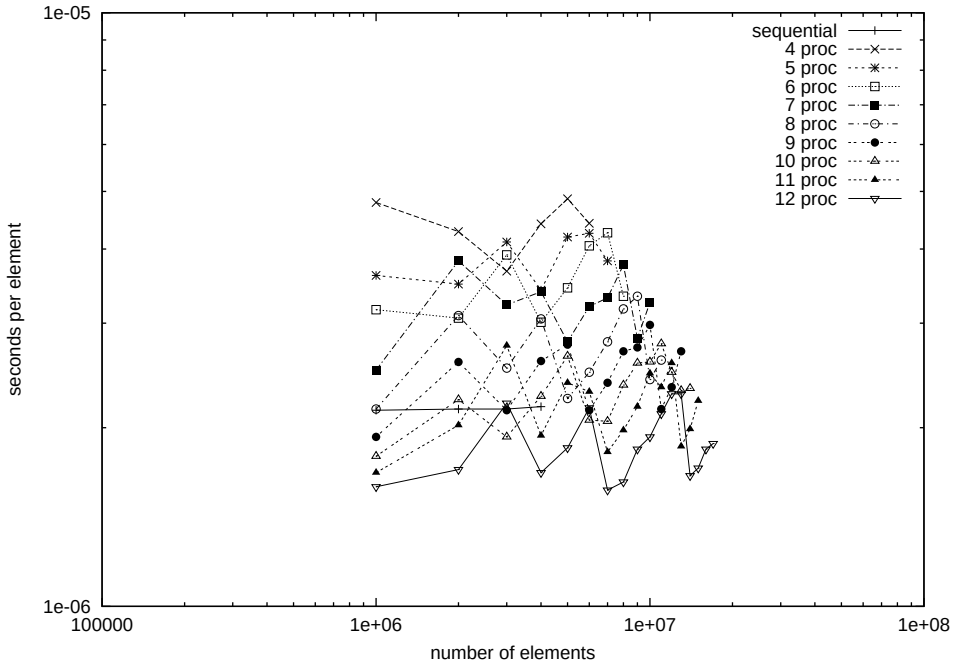


Figure 3: Execution times per element for Algorithm 5

- Algorithm 5 is more efficient than Algorithm 2. This easily can be explained by the fact that Algorithm 5 requires less communication rounds and smaller workload and communication costs. Moreover, the theoretical complexity of Algorithm 2 stands for $p \geq 17$, whereas we use at most 12 PC. This can explain these results. We also noted that for Algorithm 5, the size of $I$ is about one third of $L$ (compared to the theoretical $\frac{1}{4}$). If we are not able to explain so far, it results that Algorithm 5 works better than expected.

- Algorithm 5 is greedier in memory than Algorithm 2, therefore we can not use this algorithm with a cluster having less than 4 PC.

### 5.2 Verification of the complexity.

A positive fact that we can deduce from the plots given above 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 3, that Algorithm 5 deviates a bit more from linearity in $n$ (the list size) than Algorithm 2. But this is only a scaling effect: the variation between the values for a fixed $p$ and $n$ varying is very small (less than $1\mu s$).

For increasing amount of data and fixed $p$ the number of CGM-rounds remains constant. As a consequence, the total number of messages is constant, too. So do the costs for initiating messages, which in turn correspond to the offsets of the curves of the total running times. On the other hand, the size of messages varies. But from Figures 2 and 3, we see that the time for communicating data is also linear in $n$. Therefore, we can say that, for this problem (that leads to quite sophisticated parallel/distributed algorithms) and for PC clusters, the local computations and the number of communication rounds are good parameters to predict the qualitative behavior. Nevertheless, they are not sufficient to be able to predict the constants of proportionality and to know the algorithms that will give efficient results or not (as noticed for Algorithm 2).

If moreover, we take the overall workload and communication into account, we see that Algorithm 5 having a workload closer to the sequential one, leads to more efficient results.

### 5.3 Taking memory saturation into account

This picture brightens if we take the well known effects of memory saturation into account. In Figure 2 and Figure 3, all the curves stop before the swapping effects on PC. Due to these 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 can solve the problem in 21.8 seconds with 12 PC and Algorithm 5 does it in 9.24 seconds with 12 PC.

We see also that to handle lists with 18 millions elements with Algorithm 2 we need 71 seconds and with 17 millions elements with Algorithm 5 18 seconds. Therefore, our algorithms perform well on huge lists.

## 6 Conclusion

To conclude, we are going to study the three goals given in the introduction:

- The proposed algorithms were implemented on a PC cluster, as shown in Section 5. We think that the CGM model (and more generally the coarsed grained models) is well adapted to design algorithms and code on clusters,

- The written code is independent of the architecture of the target machine (here a PC cluster). As explained above, the code was used on different machines. This shows that our code is portable. As fas as we know, this is the first portable code on List Ranking that runs on PC clusters as well as on mainframe parallel machines.

  Section 5 showed that Algorithm 5 is faster than the sequential algorithm if 9 or more PCs are used. Even if the obtained speedups are small, it is nevertheless possible to obtain speedups for such challenging problems like the LRP on a cluster with few PCs. We showed that it is possible to solve the LRP on very large lists with the two proposed algorithms. We think that this is a first step towards handling graphs in parallel on such low cost machines.

- Section 5 showed that it was possible to predict the behavior of the curves and that no dramatic deviations from the expectations were obtained. We found that the CGM model with the distinction into local computation steps and global communication rounds allows us to know whether or not the behavior of a code will be correct on clusters. Nevertheless, we saw that it was not sufficient to deduce the constants of proportionality. If we take the overall workload and communication costs into account, then we have a better estimation of the results, as was shown in Section 5.

## Acknowledgement

The implementation part of this work only was possible because of the constant and competent support we received from the various system engineers of the test sites. Especially, we want to thank Loïc Prilly from the ENS Lyon.

## References

[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 multicomputers and BSP. In Degano, P., Gorrieri, R., and Marchetti-Spaccamela, A., editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Comp. Sci.*, pages 390–400. Springer-Verlag. Proceedings of the 24th International Colloquium ICALP'97.

[Cole and Vishkin, 1989] Cole, R. and Vishkin, U. (1989). Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):128–142.

[Dehne et al., 1996] Dehne, F., Fabri, A., and Rau-Chaplin, A. (1996). Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400.

[Dehne and Song, 1996] Dehne, F. and Song, S. W. (1996). Randomized parallel list ranking for distributed memory multiprocessors. In Jaffar, J. and Yap, R. H. C., editors, *Concurrency and Parallelism, Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Comp. Sci.*, pages 1–10. Springer-Verlag. Proceedings of the Asian Computer Science Conference (ASIAN '96).

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

[Karp and Ramachandran, 1990] Karp, R. M. and Ramachandran, V. (1990). Parallel Algorithms for Shared-Memory Machines. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume A, Algorithms and Complexity, pages 869–941. Elsevier Science Publishers B.V., Amsterdam.

[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.

[Sibeyn, 1997] Sibeyn, J. (1997). Better trade-offs for parallel list ranking. In *Proc. of 9th ACM Symposium on Parallel Algorithms and Architectures*, 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.

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

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

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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