List Ranking on a Coarse Grained Multiprocessor

Isabelle Guérin Lassous, Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

List Ranking on a Coarse Grained Multiprocessor

Isabelle Guérin Lassous and Jens Gustedt

No 3640

Mars 1999

THÈME 1

ISSN 0249-6399

List Ranking on a Coarse Grained Multiprocessor

Isabelle Guérin Lassous* and Jens Gustedt†

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n°3640 — Mars 1999 — 14 pages

Abstract: We present a *deterministic* algorithm for the List Ranking Problem on a Coarse Grained p-Multiprocessor (CGM) that is only a factor of $\log^*(p)$ away from optimality. This statement holds as well for counting communication rounds where it achieves $O(\log(p)\log^*(p))$ and for the required communication cost and total computation time where it achieves $O(n\log^*(p))$. We report on experimental studies of that algorithm on a variety of platforms that show the validity of the chosen CGM-model, and also show the possible gains and limits of such an algorithm. Finally, we suggest to extend CGM model by the *communication blow up* to allow better a priori predictions of communication costs of algorithms.

Key-words: list ranking, coarse grained multiprocessor, parallel algorithms, models of paralellism

(Résumé : tsvp)

Part of this work has been realized by a visiting grant for Jens Gustedt at the Université Paris VII

* LIAFA – Université Paris VII Denis Diderot, place Jussieu, F-75000 Paris Cedex – France. Email: guerin@liafa.jussieu.fr
† TU Berlin, Germany and INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr

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
Téléphone : 03 87 20 35 00 - International: +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Classement de listes sur un multi-processeur à gros grain

Résumé : Nous présentons un algorithme déterministique pour le problème du classement de listes sur un multiprocesseur à gros grain (CGM). Sa complexité est d'un facteur de $\log^*(p)$ de l'optimalité, aussi bien pour le nombre de phase de communication, où il arrive à $O(\log(p)\log^*(p))$, que pour le coût de la communication et le temps total pour le calcul, où il arrive à $O(n\log^*(p))$. Nous exposons les études expérimentales de cet algorithme effectuées sur une varieté de plate-formes. Les expériments montrent la validité du modèle CGM choisi, les gains possibles et les limites d'un tel algorithme. Finalement, nous proposons une extension du modèle CGM avec l'invariant de *« blow-up » de communication* pour mieux permettre des prédictions à priori des coûts de communication.

Mots-clé : classement de listes, multiprocesseur à gros grain, algorithmes parallèles, modèles du parallèlisme

1 Introduction and Overview

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. 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 and portable model of parallel computation was available that allowed easy implementations.

The well studied variants of the PRAM model, see Karp and Ramachandran (1990) for an overview, are *fine grained* in nature; 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.

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 that combines theoretical abstraction with applicability to a wide range of architectures, 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, so-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.

In the case of LRP, such problems easily kick any algorithm that hides some log-factor in the over all communication or CPU costs out of the game: the obvious sequential algorithm performs so well, that the log-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. In this paper we give a deterministic algorithm for the LRP that for the first time bounds the proportional overhead for these measures to a reasonable small factor that only depends on p, the number of processors, and not on n, the size of the data. In particular we show that our algorithm performs in $O(\log p \log^* p)$ communication rounds and that the overall work load and communication cost are in $O(n \log^* p)$. This improves over the previously known *randomized* algorithm by Dehne and Song (1996) that performs in $O(\log p \log^* n)$ rounds with a workload/communication of $O(n \log^* n)$, and over a deterministic algorithm of Caceres et al. (1997) that needs $O(\log p)$ rounds and a workload/communication of $O(n \log p)$. Consider Table 1 for an overview of these different measures for the algorithms mentioned. The algorithms of Sibeyn (1997), not expressed in the CGM model, have different measures. We mention them for their practical interest.

Indeed, 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. Sibeyn (1997) gives good and promising results of different implementations of his randomized algorithms. The implementations are specific to the Intel Paragon and not portable (the author gives no details concerning the chosen language for instance). In this article, we address ourselves to the problem of writing portable, efficient and predictive deterministic code for LRP.

The paper is organized as follow : we give the main features of the CGM model in Section 2. After having formulated the basics of LRP in Section 3, we present our algorithm for solving the LRP in Section 4. Section 5 concerns the results of the implementation of our algorithm on a T3E, on an Origin2000 and on two PC clusters

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

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

with two different interconnection networks. The analyses of these results lead us to give some precisions to the CGM model in Section 6.

2 The CGM model for parallel 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 simultaneously between phases with only computations local to the processors and communications between those processors, so-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. 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 only on p only is desired. But, as already discussed in the introduction, this may fall to short, since the main bottleneck for this kind of parallel computation is the overall communication between the processors. We will see examples for that in the implementation section below.

At the end of this paper we give an extension of the cost model, that is able to cover that difficulty without any additional parameter or platform dependent specification.

3 Basics of the List Ranking Problem

Algorithm 1 reflects one of the first and basic ideas to solve the LRP on a parallel machine. It is easy to see that `Jump` is correct and that Invariant A is always satisfied if it was true in 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.

PROPOSITION 3.1. *Let R and S be inputs for `Jump`, and let ℓ be the maximum length of an element $x \in R \setminus S$ to the next element $s \in S$. Then $\texttt{Jump}(R, S)$ 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 perform 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.

Algorithm 1: `Jump`

Input: Set R of n linked items e with pointer $e.\mathsf{succ}$ and distance value 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 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 $e.\mathsf{succ} = e.\mathsf{succ} \to \mathsf{succ}$

COROLLARY 3.1. *`Jump` can be implemented on a CGM such that it runs for R, L and ℓ as above in $O(\lceil \log_2 \ell \rceil)$ communication rounds and requires an overall bandwith and processing time of $O(n \lceil \log_2 \ell \rceil)$.*

4 The Algorithm

Algorithm 2 is the top-level view of our algorithm to solve the LRP on a CGM. It implements the well-known recursive approach to use a so-called k-**ruling set**. 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.

Algorithm 2: $\texttt{ListRanking}_k$

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 $e.\mathsf{succ}$ to the end of the list t and $e.\mathsf{dist}$ 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** $e.\mathsf{lot}$ = 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 $e.\mathsf{succ}$ to the next element in S ;
7 $\texttt{ListRanking}_k(S)$;
8 $\texttt{Jump}(L, \{t\})$;

PROPOSITION 4.1. *Suppose we have an implementation $\texttt{Ruling}_k$ of a k-ruling set algorithm then $\texttt{ListRanking}_k$ 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 $\texttt{Ruling}_k$ need.*

Proof. The only critical parts for these statements are lines 5, 6 and 8. Corollary 3.1 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 bandwidt hand 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. The inner (and interesting) part of such a k-ruling algorithm is given in Algorithm 3. Here an item e decides whether or not it belongs to the ruling

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 $:= 2 * b_0 + b_1$.

set by some local value e.lot according 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.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.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.lot.

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

PROPOSITION 4.2. *`Ruling`$_k$ can be implemented on a CGM such that it uses $O(\log^* q)$ communication rounds and requires and 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.lot. After this line only winners and loosers modify their value e.lot. Or according to the algorithm *RuleOut* and lines 6, 7 and 8, if e is a winner then e.succ is a looser therefore the two values e.lot are different. Because of Invariant C, we can see that two elements of S are not neighbors in R.

Algorithm 4: `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.*

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 $\log^* q + 1$.

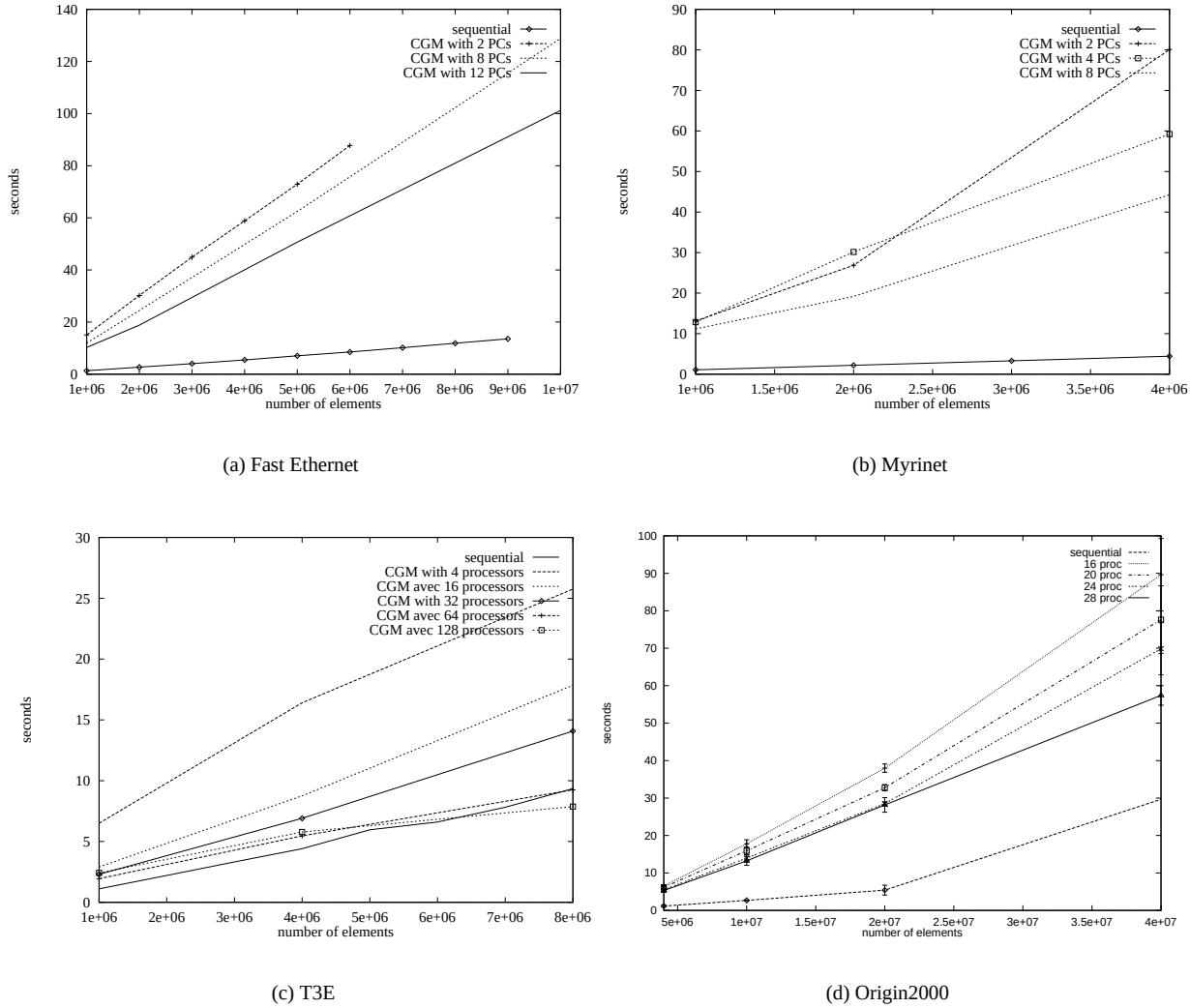
If the loop is exited when R contains only winner and looser elements then we claim that the distance between two winners in R is at most 3. Indeed, all the loosers have at least one neighbor that is a winner. An element e can become a looser in two ways: either it has a winner neighbor, either it is 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 $f.\mathsf{pred}$ is a looser (line 2 of `RuleOut`) and $f.\mathsf{succ} = e$ 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 is bounded by $O(\log^* q)$. □

PROPOSITION 4.3. *If* $p \geq 17$, `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 Proposition 4.1, 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 $O(nlog_2^* p)$. □



(a) Fast Ethernet

(b) Myrinet

(c) T3E

(d) Origin2000

Figure 1: Execution times on the four different architectures

5 Implementation

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. We will show in the following that here the CGM model allows a relatively good prediction of what we may expect. In particular we will see that even for relatively large amount of processors that are connected by a not too fast communication network there will be a loss or a very little gain compared to a sequential algorithm. This qualitative statement shows to be relatively stable with respect to an actual implementation and the chosen platform.

5.1 A platform independent implementation.

The implementation of the algorithm was done –as we think– quite carefully in C++ and based on PVM or MPI, two well-known libraries for message passing between processes. The use of C++ allowed us to actually do the implementation on different levels of abstraction:

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

For each list size, the program was run (at least) 10 times and the result is the average of these results. The implementations were carried out on quite different platforms such as PC clusters and main frames. We were using

MPI on

- a Cray T3E and
- a Silicon Graphics Origin2000,

both equipped with their native OS, proprietary C++ compilers and proprietary implementation of the MPI library,

PVM on PC clusters

- with a Myrinet network and
- with a 100MB/s Fast Ethernet network,

both equipped with the Linux OS, the g++ compiler from the EGCS project and the public domain implementation of PVM.

Not all machines were available at all times. As usual, it was difficult to run the tests under reliable circumstances. Thus only part of the tests what would have been desirable could be realized.

As far as possible, the code was the same for the four machines: for all four, the level implementing CGM and the list ranking algorithm has been the same. For each of the two subgroups using MPI and PVM respectively the code interfacing to these libraries also has been identical. This shows that our code is well portable on a variety of machines.

When doing such an implementation and then running it on the machines, the first disappointment comes when considering plots as in Fig. 1 that compares the running times of a sequential algorithm against the parallel one on p processors:

For nearly all values of p, the parallel algorithm is slower than the sequential one.

All the curves stop before the memory saturation of the sequential algorithm. For the two PC clusters, p varies from 2 to 12 (Fast Ethernet) and 8 (Myrinet). Note that for the Myrinet network the PC memory is of 64MB whereas it is of 128MB for the Fast Ethernet, which explains the differences in scale. With these machines, the parallel algorithm is always slower than the sequential one, but the parallel execution time decreases with the number of used processors.

For the T3E, p varies from 2 to 128 processors. The memory of each processor is of 64MB. A main problem on this machine was the suboptimal C++ compiler. The parallel algorithm becomes quicker than the sequential one from 64 processors on. But the reader may observe, that the main cause for this is not the good performance of the parallel implementation, but the poor performance of the sequential one, if we compare it to the results of the other platforms.

For the Origin2000 we show results from 16 to 28 processors. The results for higher numbers of processors show little improvement, so they are omitted. This is due to the fact that the Origin2000 in contrast to the other platforms has no clear borderline between “RAM behavior” and “swap behavior”.

In fact the virtual shared memory implementation in the OS of this machine together with *eg* memory page migration are some of the big advantages of that machine compared to the T3E: it allowed us to run the *sequential* algorithm for up to 3E+8 elements, whereas we had to stop at about 8E+6 for the T3E. Observe also that, although there is not big difference in the clock rates of the two machines, the same code for this sequential algorithm performs in all ranges by magnitudes faster on the Origin2000 than on the T3E.

We see 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. But we do also see that this improvement is not dramatic in neither case.

5.2 Verification of CGM paradigm.

The first positive fact that we can deduce from the plots given above is that the execution time for a fixed amount of processors p really shows a strongly linear behavior as expected (whatever the number of used processors may be). Since for increasing amount of data and fixed p the number of CGM-rounds and thus the number of messages remains constant we can deduce that, contrary to the size of messages, the number of messages is small enough such that it can be neglected for the running time.

> *Alternation of communication and computation rounds is a successful strategy to overcome network latency.*

5.3 Verification of the complexity.

Our algorithm should have a multiplicative factor of $a(p) = \alpha \cdot \log^*(p)/p$ for the effective running time where α is a constant depending only on the combination of hardware, software and implementation on a given platform. This constant reflects the average delay that the program uses for communicating data items per communication round on a specific platform. A regression on the data gives the times indicated in Fig. 2. Clearly the real factor α that we obtain can’t be a constant, but it also depends on p.

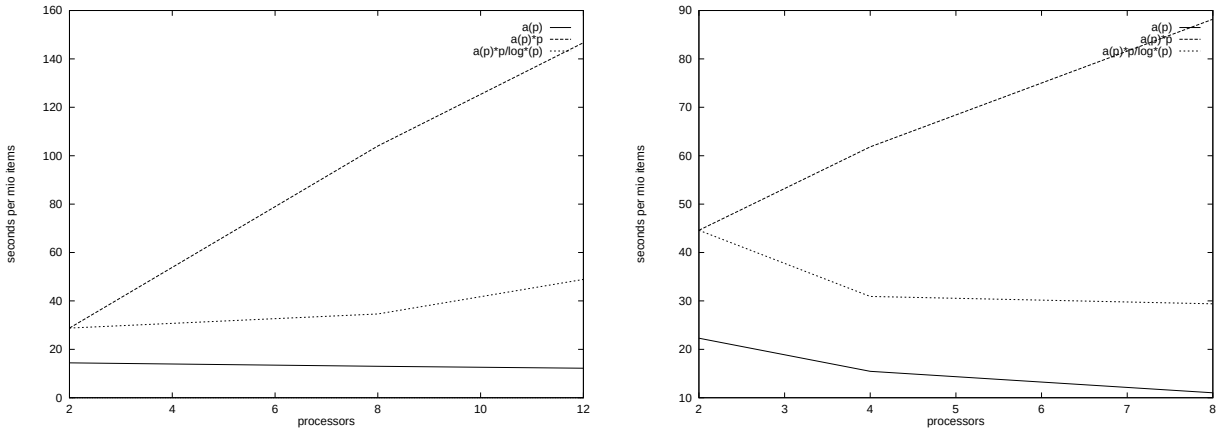
In all four figures the lower line is $a(p)$ as we found it for the data given in Figure 1. The upper line shows $a(p) * p$, the per processor cost per item, and the middle one shows $\alpha(p)$. Clearly that these curves only give a rough estimate of the behavior of the algorithm and its implementation. But we think that the general pattern is clear:

- The PC clusters, regardless of their network hardware have α around 30 seconds per mio items. We think that this is mainly due to the fact of using PVM as a library, but unfortunately lack constant availability and the instability of the machines made tests with MPI impossible up to now.
- The T3E doesn’t achieve a much better factor than the PC clusters. And this, although it has an inter-processor communication hardware that is by magnitudes faster than the faster of the PC networks.
- The Origin2000 has a much better constant in the shown range. But outside of that range the constant is in fact no constant any more, its value slowly increases to reach in fact about the same value as on the other platforms.

5.4 Taking memory saturation into account

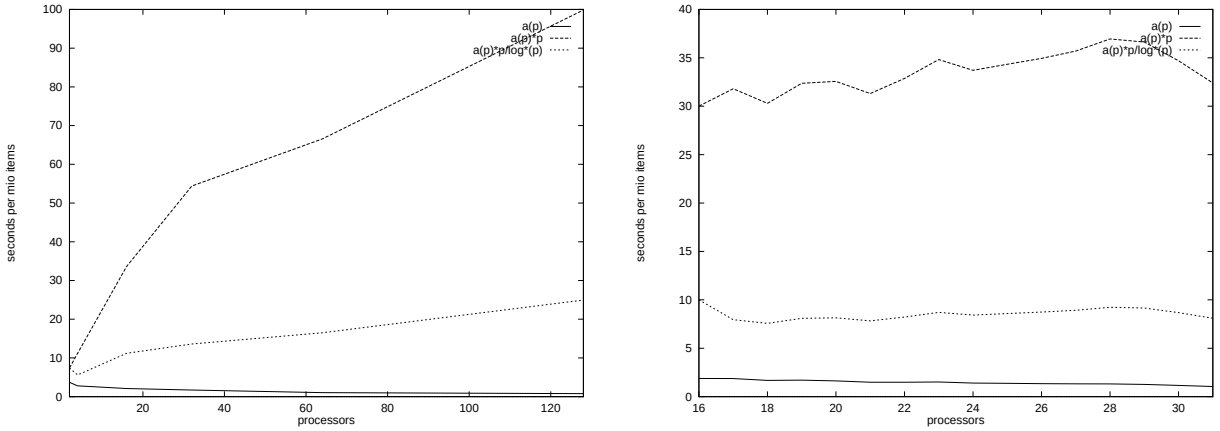
This picture brightens if we take the well known effects of memory saturation into account. Fig. 3 shows the execution times for the sequential and the parallel algorithms when exceeding the range in which the data can be hold in conventional memory. Since the scales of the different curves are quite distinct, we use a doubly logarithmic scale.

Subfigures 3(a) and 3(b) deal with the swapping effects on the two PC clusters. As we can see from the plot, due to these effects the sequential algorithm changes its behavior drastically when run with more than 6 million



(a) Fast Ethernet

(b) Myrinet



(c) T3E

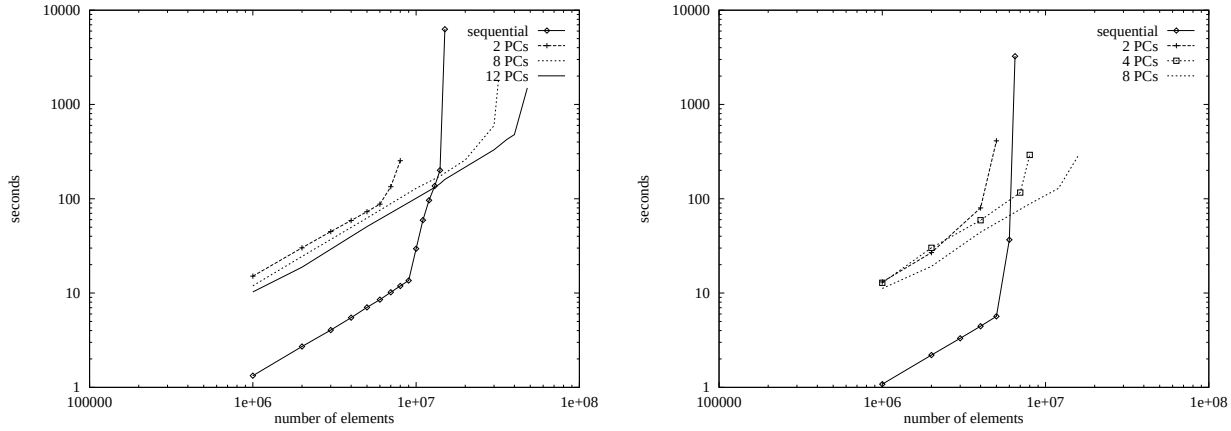
(d) Origin2000

Figure 2: The coefficients as a function of p

elements on the Myrinet network and with more than 14 million elements on the Fast Ethernet network. The factor of proportionality worsens drastically. Such a large factor would usually make this sequential algorithm impractical for a large amount of data. Here our implementation starts to be really competitive for 12 PCs on the Fast Ethernet network or for 8 PCs for the Myrinet network.

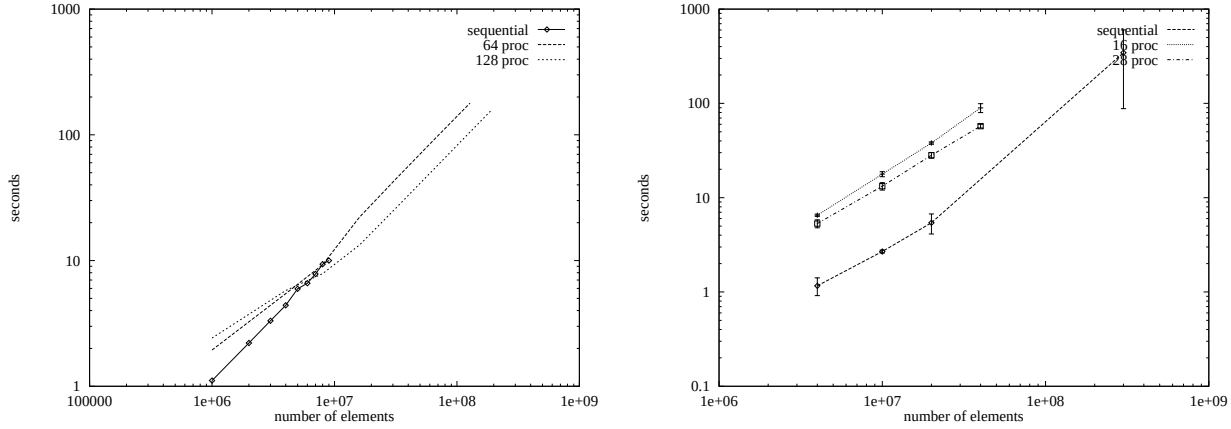
On the T3E, the memory of a processor is saturated with the sequential algorithm from 10 million elements. The OS doesn't implement swap at all, so execution just stops if a process tries to access more memory than it might. Fig. 3(c) shows that our CGM algorithm can solve quickly the LRP on lists of 128 million elements with 64 processors or on lists of 192 million elements with 128 processors.

As already mentioned above, the Origin2000 behaves quite differently. The memory organization of the OS performs surprisingly well. In fact, the OS uses the RAM of the other processors if the one of an individual processor is saturated and caches these pages locally only if necessary. So there is no hope to achieve competitive running times with our parallel algorithm, at least as long as we stick on using a message passing library.



(a) Fast Ethernet

(b) Myrinet



(c) T3E

(d) Origin2000

Figure 3: The effects of memory saturation

6 The communication blow up – an extension of the CGM paradigm

As we mentioned above, we think that one of the big advantages of the CGM model is, that it only uses two parameters to measure the theoretical performance of an algorithm. One, the number of processors p, depends on the platform and the other one n depends on the amount of input data. We have seen that with these parameters alone it was possible to predict the qualitative behavior of the asymptotics for a quite sophisticated algorithm. On the other hand, the prediction of the constants of proportionality was not satisfactory.

For that purpose, other models of parallel computation as BSP (Valiant (1990)) and LogP (Culler et al. (1993)) introduce new parameters that reflect certain hardware characteristics, such as communication latency or ... and are well suited to predict how switching from one platform to another will effect the performance of an algorithm after an implementation has been done. But they are unsatisfactory in the sense that they don't facilitate an *a priori* analysis of the algorithm.

The main problem that all these models like BSP, LogP and CGM attack is the communication bottleneck between processing units. Thus, it would be convenient to have a measure that allows to focus on the communication overhead that a parallel algorithm produces. Our proposition is to compare the overall communication $c(p,n)$ of a parallel algorithm to the corresponding value of the best known sequential algorithm. As such a corresponding value we chose the random memory access, $v(n)$, that is issued by a

sequential algorithm. In fact this measure corresponds to the main bottleneck of sequential computation on a RAM, as it has already been seen by von Neumann.

If both values are measured in machine words, their quotient is a unit-less invariant of the parallel algorithm. So for a given parallel algorithm $\mathcal{A}$ and an unspecified optimal sequential algorithm, we call this quotient

$$b_{\mathcal{A}}(p,n) = \frac{c_{\mathcal{A}}(p,n)}{v(n)} \tag{6.1}$$

the *communication blow up* of $\mathcal{A}$.

To be concrete let us check this value for our list ranking algorithm. Therefore we must first estimate $v(n)$ for this problem. The sequential algorithm that we implemented basically runs twice through the data: in a first scan it determines the tail(s) of the list(s) and in a second it finds the head from each sublist by doing a simple pointer jumping from these identified tails. This means that we basically do two random accesses per data item, one to tell its successor that it is not a tail and one when performing the actual pointer jumping. So we have

$$v(n) \leq 2 \cdot n \cdot words. \tag{6.2}$$

The communication needs of our parallel algorithm can must be distinguished into different parts:

1. The communication of the ruling set algorithm must be separated in the cost that must be accounted
   (a) to the different phases of the ruling set algorithm, $f(p,n)$, and
   (b) to the communication that is performed only once per data item, $i(n)$.
2. This ruling set algorithm is called recursively, so we must also estimate what this recursion adds to the total cost, $r(p,n)$.

So $c(p,n)$ can be estimated with

$$c(p,n) \geq r(p,n) \cdot (f(p,n) + i(n)) \cdot words. \tag{6.3}$$

For $f(p,n)$, observe that in each phase of the ruling set algorithm one address and two values of lot are communicated. So the total cost of that loop is $f(p,n) = 3\log^*(p) \cdot n$. Communication for individual items occurs when they are chosen to be winner or looser. This consist in communication of two words per item so $i(n) = 2n$. The lists that are passed to the recursive call are of a total length that is of size less than a half of the original one, so

$$r(p,n) \geq \frac{1}{1-\frac{1}{2}} = 2. \tag{6.4}$$

In total this leads us to an estimation of

$$c(p,n) \geq 2 \cdot (3\log^*(p) + 2) \cdot n \cdot words, \tag{6.5}$$

and

$$b(p,n) \geq 3\log^*(p) + 2 \tag{6.6}$$

For large values of p this evaluates to $3\log^*(p) + 2 = 3 \cdot 4 + 2 = 14$.

Observe that this value only gives a rough lower bound of the factor that is finally realized by a concrete implementation on a concrete platform. *Eg* the use of a message passing library that buffers the data blows up by another factor of 2, leading to a blow up of 28 in our case.

Then, the choice of $v(n)$ for a comparison is an underestimation, that is only justified if we have a system for which communication between processors is as expensive as the addressing of memory by an individual processor. This has not been the case for the Origin2000, where the efficient memory management of the OS guarantees much faster access to RAM than ever would be possible with a message passing library.

7 Conclusion

We showed that it is possible to solve the LRP on very large lists with our algorithm whatever the used platform may be. We confirmed the fact that the approach of coarse grained models as CGM or BSP to group communication as far as possible together pays off and allows a better predictability of concrete implementations of our algorithm.

We were able to compute concrete implementation depending factors, that allow a judgment and comparison of these different platforms, that is of an interplay between hardware, OS, compilers, the message passing libraries and last but not least our implementation.

On the other hand, we found it necessary to introduce a new invariant, the *communication blow up*, that permits an *a priori* evaluation of algorithms, and shows how this blow up may explain certain behavior that we observed during the implementation.

Acknowledgement

All the implementation part of this work only was possible because of the constant and competent support we received from the various system engineers at the different test sites.

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.

[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., 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).

[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.

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

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e 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