Portable List Ranking: an Experimental Study

Isabelle Guérin Lassous
INRIA Rhône-Alpes, LIP, ENS Lyon
and
Jens Gustedt
LORIA and INRIA Lorraine

We present and analyze two portable algorithms for the List Ranking Problem in the Coarse Grained Multicomputer model (CGM). We report on implementations of these algorithms and experiments that were done with these on a variety of parallel and distributed architectures, ranging from PC clusters to a mainframe parallel machine. With these experiments, we validate the chosen CGM model, and also show the possible gains and limits of such algorithms.

---

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 and to compute the distance (or other parameters) to the end of such a chain of references. List Ranking problems are problems concerning a linked list, *i.e* is a set of nodes such that each node (but the tail of the list) points to another node called its successor and there is no cycle in such a list. The *restricted LRP* consists in determining the rank for all nodes, that is the distance to the last node of the list.

To be able to use an LR algorithm as a subroutine for other algorithms, it must also be able to handle collections of lists, that are not necessarily connected. Therefore we work in a more general setting where the list is cut into sublists and then the *general LRP* consists in identifying for all nodes the head of the particular sublist and determining their individual distance to that head. Figure 1 gives an example of the general LRP (called only LRP henceforth). The circled nodes are the last nodes of sublists.

LRP has obvious solutions in sequential contexts of computation, but the performance of implementations of such sequential algorithms is already surprisingly bad: up to our knowledge the best ones need between 150 to 450 CPU cycles per element (depending on the platform), whereas the related problem of computing the prefix sum of an array only requires about 10. This can by explain by The non locality of the data in this algorithm that implies numerous cachemisses.

Already this restricted performance in a sequential setting would well motivate any attempt to seek parallel solutions for this problems. In addition for many parallel graph algorithms List Ranking is crucial as a subroutine.

Whereas this problem seems (at a first glance) to have straightforward solutions in a se-

---

INRIA Rhône-Alpes, LIP, ENS Lyon, 46, allée d'Italie, 69007 Lyon, France. Email: `Isabelle.Guerin-Lassous@ens-lyon.fr` LORIA and INRIA Lorraine, Campus scientifique, BP 239, 54506 Vandœuvre-lès-Nancy, France. Email: `Jens.Gustedt@loria.fr`

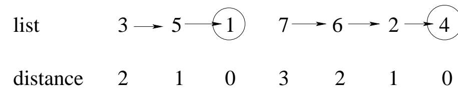

Fig. 1. Example of the LRP

quential 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 parallel algorithms for the LRP, see Karp and Ramachandran 1990 for an overview, are *fine grained* in nature, and written in the PRAM model; usually in algorithms written in 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 mainframe parallel computers as well as distributed systems, 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. See Guérin Lassous et al. 2000 for a broader description of the CGM model and for a discussion of different types of algorithms designed therefore.

Previous algorithms in the coarse grained models.

The first proposed algorithm in the coarse grained models is a randomized algorithm by Dehne and Song 1996 that performs in $O(\log p \log^* n)$ rounds (p is the number of processors and n the size of the linked list) with a workload (total number of local steps) and total communication size of $O(n \log^* n)$ ($log^* n = min\{i \mid log^{(i)} n \leq 1\}$). Then, Caceres et al. 1997 gave a deterministic algorithm that needs $O(\log p)$ rounds and a workload/total communication of $O(n \log p)$. These two algorithms were designed in the CGM model. As far as we know, no implementations of these algorithms have been carried out.

reference	comm. rounds		CPU time & communication		
Dehne and Song 1996	$\log p$	$\log^* n$	n	$\log^* n$	rand
Caceres et al. 1997	$\log p$		n	$\log p$	det
Sibeyn 1997			n		aver
Sibeyn 1999	$\log\log n$		n	$\log p$	aver
we	$\log p$	$\log^* p$	n	$\log^* p$	det
we	$\log p$		n		rand

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

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 (vectorized architecture) of different PRAM algorithms that gives good results. In Dehne and Song 1996, some simulations have been done, but they only give some results on the number of communication rounds. In Patel et al. 1997, fine-grained implementations of List Ranking are studied for applications such as computer vision and image processing. Their approach takes advantage of the locality and connectivity properties of images. They compare different algorithms, but no speedup results are given.

Sibeyn 1997 and Sibeyn et al. 1999 give several algorithms for the LRP with some algorithms derived from known PRAM techniques and some new ones. They fine-tune their algorithms according to the features of the interconnection network of the Intel Paragon and their code uses the communication library dedicated to the Paragon. The authors do not mention the portability aspects of their implementations.

Sibeyn 1999 and Lambert and Sibeyn 1999 propose an algorithm that has a total communication size of $\left(6+\frac{3\ln d+6\ln p}{d+1}\right)n$ where d is the number of recursion steps. It requires $6+2d\lceil\log\log n\rceil$ communication rounds (their cost measure is a simplification of BSP). These algorithms have been implemented on an Intel Paragon (with the NX communication library) and a cluster of workstations. On the Intel Paragon, speedups are obtained for p greater than 16 and n greater 1 million. On the workstations cluster, only restricted speedups are obtained for 16 workstations and lists of size greater than 1 million.

	machine				times		processors	
reference	CPU	speed	architecture	data range	seq	para	min	max
Lambert and Sibeyn 1999	i686	300 MHz	16×2-proc, 100 Mbs	1M–10M	350–385	220	16?	16
	i680 XP		Paragon	1000–1M			16?	64
this paper	RS1000	195 MHz	32×2-proc, shared mem	1M–10M	160	80	16	50
	i686	200 MHz	12 PC, Myrinet	1M–200M	365	290	9	12

Table 2. Running times (CPU cycles per item) in different implementations

This paper.

In this paper, we address ourselves to the problem of designing algorithms that give portable, efficient and predictive code for LRP. We do not pretend to have the best implementation of the LRP, but we tried to achieve three different goals, namely to be portable, efficient, predicitive at the same time. Especially, the implementation was done carefully to have the best possible results without loosing at portability level.

We propose two algorithms designed in the CGM model. First, we propose a deterministic algorithm that has better workload and total communication size than the previously known algorithms in the coarse grained models. Nevertheless, it only seems to be of theoretical interest, because it does no lead to efficient practical results as shown in Section 4. Then we propose a randomized algorithm that has better theoretical complexities on average and that gives better performances. We give the experimental results of their implementations. The code of the different implementations and the documentation of the experiments can be found at the following address: http://www.loria.fr/~gustedt/cgm. Our code runs on PC-clusters with different interconnection networks, an SGI Origin 2000, an SGI PCA cluster of 4 shared memory multiprocessors, a Cray T3E and SUN workstations. We preferred to focus on the results obtained on a specific PC cluster and the Origin 2000 because we think that they are representative.

Table 1 compares the existing coarse grained algorithms and the algorithms we propose concerning the number of communication rounds and the total workload and the total size of exchanged data. *rand* stands for randomized and *det* for deterministic. We mention the algorithms of Sibeyn 1997 and Sibeyn 1999, not designed in the CGM model, for their practical interest.

Table 2 compares experimental results of Lambert and Sibeyn 1999 and those presented in this paper. Our sequential running times are obtained with a straightforward solution of *general LRP*. The *min* number of processors is the break-even point to obtain the same running times as sequential. The parallel times are the times obtained with the *max* number of processors. We do not give the times obtained on the Paragon because we are not sure about the processor speed, since it was not given by the authors. We based our comparison on the clock cycles because this parameter allows a natural comparison with the sequential algorithms.

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 3.1, and a randomized algorithm in Section 3.2. Section 4 concerns the results of the implementations.

2. THE CGM MODEL FOR PARALLEL/DISTRIBUTED COMPUTATION

The CGM model initiated by Dehne et al. 1996 is a simplification of BSP proposed by Valiant 1990. These models 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).

The CGM model describes the number of data per processor explicitly: for a problem of size n, it assumes that the processors can hold $O(\frac{n}{p})$ data in their local memory and that $1 \ll \frac{n}{p}$. Usually the later requirement is put in concrete terms by assuming that $p \leq \frac{n}{p}$ because each processor has to store information about the other processors.

Algorithm 1: `Jump`

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

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

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

The algorithms are 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 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 CGM we have to ensure that the number R of supersteps is small compared to the size of the input. It can be shown that the *interconnection latency* which is one of the major bottlenecks for efficient parallel computation can be neglected if R is a function that only depends on p (and not on n the size of the input), see Guérin Lassous et al. 2000.

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.

3. TWO COARSE GRAINED ALGORITHMS FOR LIST RANKING

The two proposed algorithms are based on PRAM techniques used to solve the LRP. Directly translating the PRAM algorithms into CGM algorithms would lead to algorithms with $O(\log n)$ supersteps, what the CGM model does not recommend. Some efforts to bound the number supersteps have to be added. For instance, we can reduce the size of the problem to ensure that after $O(\log p)$ supersteps the problem can be solved sequentially on one processor. At the same time, we have to pay attention to the workload, as well as to the total communication bandwidth.

3.1 A deterministic algorithm

The deterministic 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. The second used PRAM technique is a *k-ruling set*, see Cole and Vishkin 1989. We use *deterministic symmetry breaking* to obtain a k-ruling set, see Jájá 1992. Such a k-ruling set S is a subset of the items in the list L such that

(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: `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. $S =$ `Ruling`$_k(p-1, n, L)$;
5. `Jump`(L, S);
6. **for** *all* $e \in S$ **do** set $e.succ$ to the next element in S ;
7. `ListRanking`$_k(S)$;
8. `Jump`$(L, \{t\})$;

Concerning the translation in the CGM model, the problems are to ensure that the size of the k-ruling sets decreases quickly at each recursive call to limit the number of recursive calls (and then of supersteps) and to ensure that the distance between two elements in the k-ruling set is not too long otherwise it would imply a consequent number of supersteps to link the elements of the k-ruling set.

Algorithm 1 implements the well known pointer jumping technique.

PROPOSITION 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* `Jump`(R,S) *requires $O(\lceil \log_2 l \rceil)$ supersteps and $O(l \lceil \log_2 l \rceil)$ workload/total communications size.* □

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.

It shows that the CGM pointer jumping technique algorithm performs in $O(\lceil \log_2 n \rceil)$ supersteps and $O(n \lceil \log_2 n \rceil)$ workload/total communications size. Due to the implied workload and total communications size, this algorithm is unlikely to lead to efficient code. Therefore, we can not just use this technique to solve the LRP, but we can use it as a subroutine of our algorithm.

Algorithm 2 solves the LRP in the CGM model. It implements the technique of the k-ruling set in CGM: the goal is to reduce the size of the list with the build of a k-ruling set; when 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 $O(\lceil \log_2 k \rceil)$*

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

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.*

PROOF. The only critical parts for these statements are lines 5, 6 and 8. Proposition 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 □ □

The inner (and interesting) part to compute a k-ruling set is given in Algorithm 3.

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 $e.lot$ 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 3. *$\mathtt{Ruling}_k$ 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 $e.lot$. 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.

range is the maximum $e.lot$ value that an element of R may have. Let b be the number of bits used to represent the value $e.lot$. When considering only non-winner and non-looser elements, after line 4, the maximum possible value for $e.lot$ is $2(b-1)-1$ that is $2\lfloor \log_2 range \rfloor + 1$. Moreover, the number of bits used to represent $e.lot$ 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 $e.lot$ 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 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 $p \geq 17$, $\mathtt{ListRanking}_k$ 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 $\mathtt{ListRanking}_k$, $\mathtt{Ruling}_k$ 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, $\mathtt{ListRanking}_k$ 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 $\mathtt{ListRanking}_k$ 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 $O(n log_2^* p)$. □ □

Algorithm 4: `Ruling`$_k$

Constants: $k > 9$ integer threshold

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

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 $f.succ$ for some $f \in R$, denote it by $e.pred$.

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

1 $range := q$;

repeat

C **Invariant**: If $e.lot \neq -\infty$ then $e.lot \neq e.succ \rightarrow lot$.

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

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

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

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

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 $e.lot$ to $+\infty$ for *winners* and to $-\infty$ for *loosers*;

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

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

return *Set S of winners.*

3.2 A randomized algorithm with better performance

In this section, we will describe a randomized algorithm for which we will have a better performance than for the deterministic one, as shown in Section 4, and that is easier to implement. 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 such a set I only contains *internal items* i.e. items that are not a head or a 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 to each other directly. The advantage of this technique compared to Algorithm 2 is that the construction of the set that goes into recursion requires only one communication round in each recursive call. To limit the number of supersteps, the depth of the recursion has to be relatively small and this can be ensured if the size of the independent set is sufficiently large in each recursive call. 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" such 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 workload is in* $O(|L|)$.

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 \ I;
  → D   | foreach i ∈ I do  Send l[v] to r[v] ;
  → D   | foreach i ∈ I do  Send r[v] and dist[v] to l[v] ;
  I →   | foreach v ∈ D with l[v] ∈ I do
        |     Let nl[v] be the value received from l[v];
        |     Set ol[v] = l[v] and l[v] = nl[v];
  I →   | foreach v ∈ D with r[v] ∈ 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);
  → I   | foreach v ∈ D with ol[v] ∈ I do  Send t[v] and d[v] to ol[v] ;
  D →   | foreach i ∈ 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];
```

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 $O(\log_{1/(1-\varepsilon)} p)$ supersteps. Also the total bound on the work depends by a factor of $1/(1-\varepsilon)$ from ε. The communication on the other hand does not depend on ε. 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 (i.e. not too small) independent set.

LEMMA 2. *Suppose every item v in list L has an integer value $A[v]$ that is randomly chosen in the interval $[1,K]$, for some value K. Let I the set of items v that have values smaller or equal than their left neighbor l, $A[v] \leq A[l]$ and strictly smaller than the one of their right neighbor r, $A[v] < A[r]$. Then I is an independent set of L and the expected size of I is*

$$E(|I|) = \frac{1}{3}\left(1 - \frac{1}{K^2}\right)|L|. \tag{1}$$

PROOF. Clearly I is an independent set. For the probability observe that if we choose $A[v]$ at random the probability that a neighbor w has a random value $A[w]$ that is strictly less than $A[v]$ is $\frac{A[v]-1}{K}$. So the probability that the neighbor value on the left is strictly less

and the neighbor value on the right is less or equal is

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

So the total probability summarized over all values of $A[v]$ for such an event is

$$\frac{1}{K}\sum_{i=1}^{K}\left(\frac{i^2}{K^2} - \frac{i}{K^2}\right) = \frac{1}{K^3}\sum_{i=1}^{K}\left(i^2 - i\right) = \frac{1}{3} - \frac{1}{3K^2} \tag{3}$$

□ □

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

(a) initializing the recursion, and
(b) choosing (pseudo) random numbers $A[v]$.

In fact, we ensure (a) by an array that always holds the active elements, i.e. 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 (b), observe that for the proof it was not necessary that the whole sequence was independent but that it is sufficient that each consecutive triplet of values is independent. We made our experiments with the following technique. Every item v basically uses its own (storage) number to compute $A[v]$. To ensure independence of the the neighboring values we use $A[v] = N \cdot v \bmod K$ for some values N and K that are chosen the same on all processors. Thereby we avoid to communicate the values that are chosen, since when necessary any processor can compute the values for its neighbors from their name.

To ensure that the values obtained by this computation are still sufficiently distinct in lower levels of recursion we choose for each such level R a different large number N_R and set $A[v] = N_R \cdot v \bmod K$.

4. IMPLEMENTATION

Our code runs on PC-clusters with different interconnection networks, an SGI Origin 2000, an SGI PCA cluster of 4 shared memory multiprocessors, a Cray T3E and SUN workstations. We preferred to focus on the results obtained on a specific PC cluster and the Origin 2000 because we think that they are representative.

For the PCs, the tests were run on the POPC[i] cluster of the ENS Lyon[ii]. It has 12 PC at 200 MHz running under linux. Each PC has a memory of 64 MB. The PC are interconnected with a high speed 1.28Gb/s MYRINET[iii] network.

The other series of tests were run on the Origin 2000[iv] of the Centre Charles Hermite[v]. It has 64 processors R10000 at 195 MHz. The memory is directly shared between pairs of processors. Beyond that physical coupling, by hardware caching the entire memory can be accessed by any processor.

---

[i] http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html/
[ii] http://www.ens-lyon.fr/
[iii] http://www.myri.com/
[iv] http://cch.loria.fr/presentation/materiel.html#origin
[v] http://cch.loria.fr/

The implementation of the algorithms was done –as we think– quite carefully in C++ and based on MPI, one well-known interface of message passing libraries between processes. 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 the specific algorithms for the LRP.

One of our intentions for this hierarchical design is to replace message passing in (1) by shared memory later on.

On both architectures we used the native C++ compilers with maximum level of optimization turned on.

For the experiments on the PC cluster, we made tests with two different implementations of MPI, MPI-BIP[vi] that uses specific features of that network and lam[vii] that is build upon standard unix sockets. Since there was no substantial difference in the performance of these two, we only document the experiments done with lam.

On the Origin 2000 we used the proprietary MPI implementation of SGI. This implementation has the advantage to use the shared memory hardware of the platform. On the other hand, it had the disadvantage that it seems to be quite greedy in memory consumption, which inhibited any performance gain for more than 50 processors.

4.1 Execution times

We report on execution times, measured in wall clock time which passed when executing on machines that where reserved for our programs during the tests. We found this the only reliable measure that allows at least some comparison between different architectures.

Besides the running times we also measured cpu times as reported by the unix kernels, times passed in our library, times passed for communication, bandwidth and number of messages that were exchanged. These don't deviate much from the predictions of the model and we refer the curious reader to the web page of the *opération CGM*[viii] for these additional statistics.

First we will briefly discuss the sequential algorithm that we use to compare for comparison. Then, we will present the execution times for Algorithm 2 on the PC cluster. Since the results are not competitive, we restrict the discussion to Algorithm 5 thereafter.

4.1.1 *The sequential algorithm.*. To solve the general LRP an algorithm can not simply run through each component of the collection of linked lists because it has no apriori knowledge about their starting points. A direct algorithm would usually compute these starting points first. Therefore it has to touch each element at least twice in a random order and thus producing in general two (time consuming) cache misses per item.

Our algorithm works as follows, see Algorithms 6 and 7. It visits the items one after the other and keeps track of whether or not a item of the list has already been visited. If the actual item v has already been visited, it does nothing and proceeds to the element $v+1$. If the actual item has not yet been visited, we mark it as visited and recursively visit the next

[vi] http://lhpca.univ-lyon1.fr/index_bip.html
[vii] http://www.mpi.nd.edu/lam/
[viii] http://www.loria.fr/~gustedt/cgm/

Algorithm 6: `SeqListRanking`

Input: Table R of n linked items e with pointer $e.succ$ and distance value $dist$.

Task: Modify $e.succ$ and $e.dist$ for all $e \in R$ s.t. $e.succ$ points to the nearest element $s \in R$ according to the list having no successor ($e.succ = nil$) and s.t. $e.dist$ holds the sum of the original $dist$ values along the list up to s.

```
for i = 1,...,n e = R[i] do Mark e as being unvisited;
for i = 1,...,n e = R[i] do
    if e is unvisited then
        visit (e);
```

Algorithm 7: `visit`

Input: Linked item e with pointer $e.succ$ and distance value $dist$.

Output: Item s with $s.succ = nil$ the endpoint of the list starting at e, and d the distance from e to s

```
if e.Succ = nil then return e and e.Dist;
if e is visited then return e.Succ and e.Dist;
Otherwise let s and d be the return values of visit(e.Succ);
Mark e as being visited;
Store s and d + e.Dist as e.Succ and e.Dist respectively;
return s and d + e.Dist;
```

item *v.succ* in the linked list. If by following these links we finally find the end of the list we track back.

By this simple technique we ensure that the link *v.succ* from any item v is only followed at most once and thus that the number of pure *random accesses* is minimized to one access per link. All other memory access is either done

—sequentially by stepping through arrays of items in order or backing tracking on the stack, or

—by write throughs directly to memory (lines “Mark” and “Store”) that must not pass through the cache.

and thus are handled more effecient by the platform than the direct algorithm mentioned above. We observed that our sequential algorithms have the same performance as described by Lambert and Sibeyn 1999 which to us looks much more sophisticated. See Table 2 for a comparison.

Our running times on the two different architectures are already quite different. On the PC we have about 365 CPU cycles per item, whereas the Origin outperforms this with the value of about 160. Since memory access is the main bottleneck of this algorithm, this difference is probably due to the substantial difference in bus speed, cache and memory sizes of the platforms.

4.1.2 *PC cluster.*. Figure 2 gives the execution times *per element* in function of the list size for Algorithm 2 on the PC cluster, whereas Figures 3 and 4 are for Algorithm 5. To cover the different orders of magnitude better, all axis are given in a *logarithmic* scale.

The lists were generated randomly with the use of random permutations. For each list size, the program was run (at least) 5 times and the result is the average of these results.

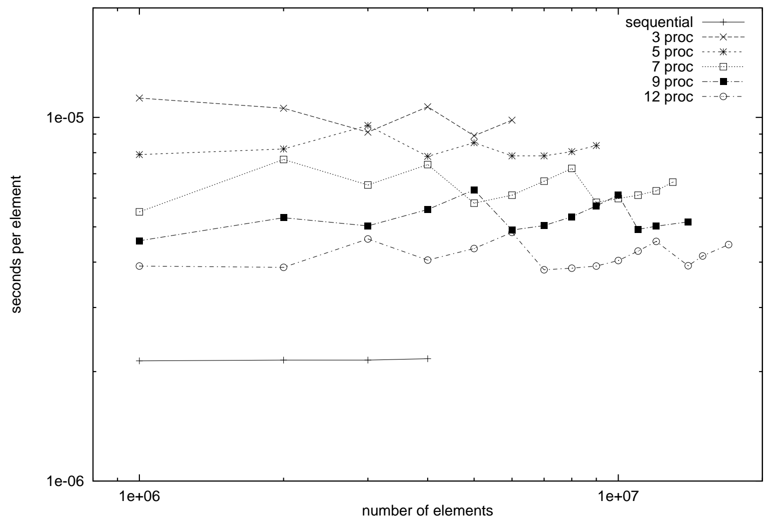

Fig. 2. Execution times per element for Algorithm 2

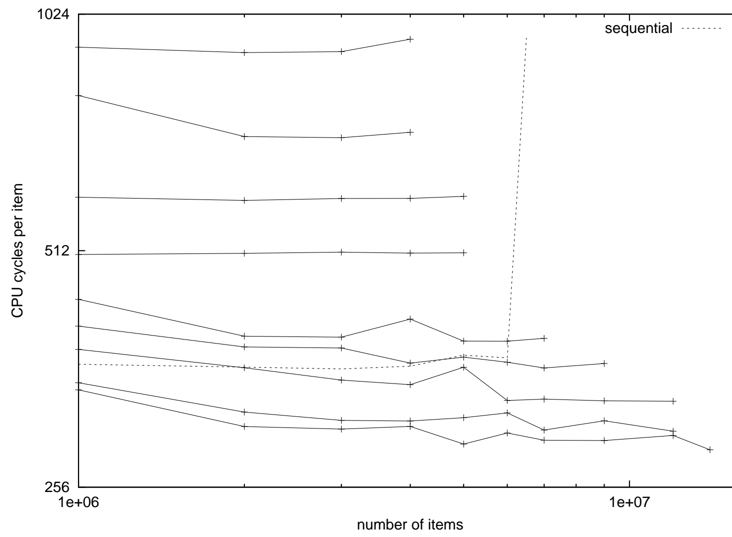

Fig. 3. Execution times per element for Algorithm 5 for 12 PC in a cluster, starting on top with the times for 2 PC going down step by step to the times for 12 PC.

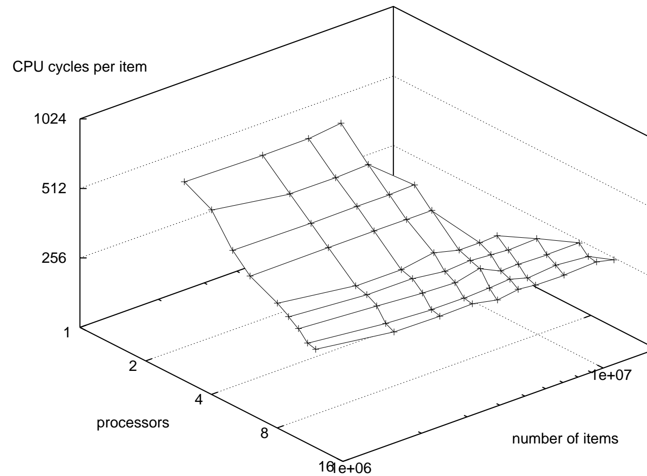

Fig. 4. Execution times per element for Algorithm 5 on a PC cluster. The results, given in three dimensions, are the same as those given in Figure 3.

For a fixed list size, very small variations in time could be noted. Figure 4 demonstrates well the scalability of our program.

p varies from 2 to 12 for both Algorithms. All the curves stop before the memory saturation of the processors. We start the measures for lists with 1 million elements.

Algorithm 2 is always slower than the sequential one. Nevertheless, the parallel execution time decreases with the number of used PC. One might expect that the parallel algorithm becomes faster than the sequential one with the use of more PC, but no cluster of more than 12 PC was at our disposal. 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.

4.1.3 *Origin 2000.*. The presentation of the results in Figures 5 and 6 for Algorithm 5 on the Origin 2000 is equal to the one given for the PC cluster. Observe that due to the much better performance of the sequential algorithm on this architecture the break-even point for achieving the same running times as in sequential is only at 16 processors.

On the other hand observe that the speed computed in number of clock cycles for the parallel algorithm don't deviate too much when using the same amount of processors. So, this gives an example of an algorithm were the the relative expensive architecture of the Origin 2000 doesn't pay off when compared to the relatively cheap PC architecture.

4.1.4 *Comparison.* If we compare the two algorithms, we can note that:

(1) The use of a larger amount of processors (for the Origin only valid up to 50 processors) lead to an improvement on the real execution times of the parallel program, which proves the scalability of our approach.

(2) Algorithm 5 is more efficient than Algorithm 2. This easily can be explained by the

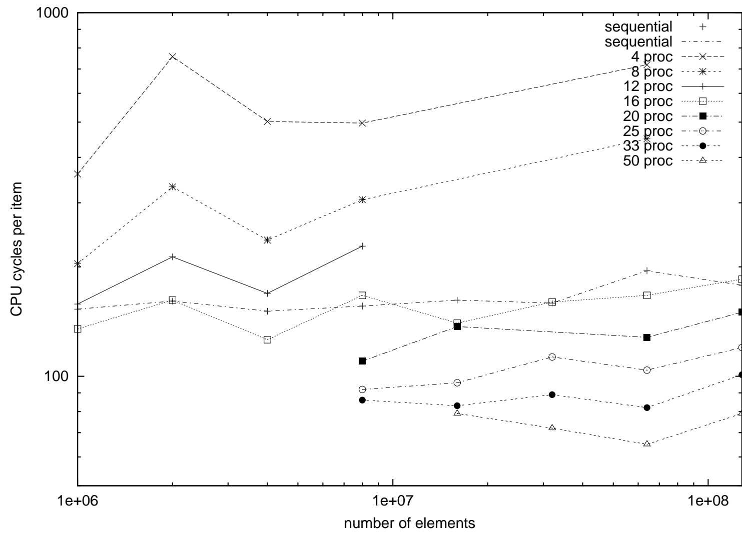

Fig. 5. Execution times per element for Algorithm 5 for processors on an Origin 2000

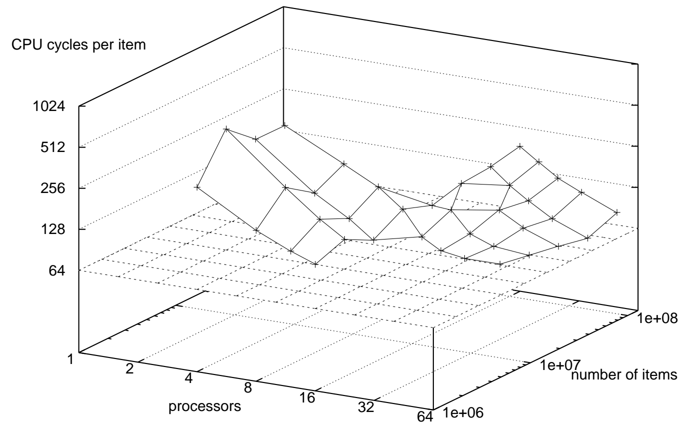

Fig. 6. Execution times per element for Algorithm 5 on an Origin 2000

fact that Algorithm 5 requires less communication rounds and smaller workload and communication costs. We also noted that for Algorithm 5, the size of I is about one third of L as proven.

(3) Our portable code does not lead to effective speedups.

4.2 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).

For increasing amount of data and fixed p the number of supersteps 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 3 and 5, 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), 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.

4.3 Taking memory saturation into account

This picture brightens if we take the well known effects of memory saturation into account. In 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 6 million elements. For 6.7 millions elements, the execution time is over 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 10.41 seconds with 12 PC. To handle lists with 18 millions elements with Algorithm 2 we need 71 seconds and with 17 millions elements with Algorithm 5 27.09 seconds. So our algorithms perform well on lists that can be considered large for a PC architecture.

For a shared memory architecture as the Origin 2000 this advantage can't play. The single processor that executes the sequential code has access to the same total amount of (expensive) memory as the set of p processors that execute a parallel algorithm.

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 Prylli from the ENS Lyon.

REFERENCES

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 P. DEGANO, R. GORRIERI, AND A. MARCHETTI-SPACCAMELA Eds., *Automata, Languages and Programming*, Volume 1256 of *Lecture Notes in Comp. Sci.* (1997), pp. 390–400. Springer-Verlag. Proceedings of the 24th International Colloquium ICALP'97.

COLE, R. AND VISHKIN, U. 1989. Faster optimal prefix sums and list ranking. *Information and Computation 81*, 3, 128–142.

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, F. AND SONG, S. W. 1996. Randomized parallel list ranking for distributed memory multiprocessors. In J. JAFFAR AND R. H. C. YAP Eds., *Concurrency and Parallelism, Programming, Networking, and Security*, Volume 1179 of *Lecture Notes in Comp. Sci.* (1996), pp. 1–10. Springer-Verlag. Proceedings of the Asian Computer Science Conference (ASIAN '96).

GUÉRIN LASSOUS, I., GUSTEDT, J., AND MORVAN, M. 2000. Feasibility, portability, predictability and efficiency: Four ambitious goals for the design and implementation of parallel coarse grained graph algorithms. Technical Report 3885, INRIA.

JÁJÁ, J. 1992. *An Introduction to Parallel Algorithms*. Addison Wesley.

KARP, R. M. AND RAMACHANDRAN, V. 1990. Parallel Algorithms for Shared-Memory Machines. In J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science*, Volume A, Algorithms and Complexity (1990), pp. 869–941. Elsevier Science Publishers B.V., Amsterdam.

LAMBERT, O. AND SIBEYN, J. F. 1999. Parallel and External List Ranking and Connected Components. In *Proceedings of the IASTED International Conference of Parallel and Distributed Computing and Systems* (1999), pp. 454 – 460.

PATEL, J., JAMIESON, L., AND KHOKHAR, A. A. 1997. Scalable Parallel Implementation of List Ranking on Fine-Grained Machines. *IEEE Transactions on Parallel and Distributed Systems 8*, 10, 1006–1018.

REID-MILLER, M. 1994. List ranking and list scan on the Cray C-90. In *Proc. ACM Symp. on Parallel Algorithms and Architectures* (1994), pp. 104–113.

SIBEYN, J. F. 1997. Better trade-offs for parallel list ranking. In *Proc. of 9th ACM Symposium on Parallel Algorithms and Architectures* (1997), pp. 221–230.

SIBEYN, J. F. 1999. Ultimate Parallel List Ranking? In *Proceedings of the 6th Conference on High Performance Computing* (1999), pp. 191–201.

SIBEYN, J. F., GUILLAUME, F., AND SEIDEL, T. 1999. Practical Parallel List Ranking. *Journal of Parallel and Distributed Computing 56*, 156–180.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Communications of the ACM 33*, 8, 103–111.

WYLLIE, J. C. 1979. *The Complexity of Parallel Computations*. Ph. D. thesis, Computer Science Department, Cornell University.