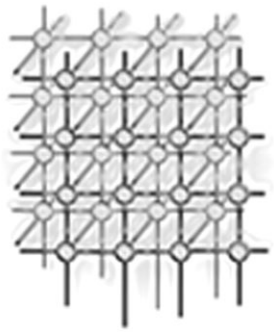
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2002; **14**:885–910 (DOI: 10.1002/cpe.700)



Parallel computation on interval graphs: algorithms and experiments

A. Ferreira[1,*,†], I. Guérin Lassous[2], K. Marcus[3] and A. Rau-Chaplin[4]

[1] *CNRS, Projet Mascotte, BP 93, F-06902 Sophia Antipolis, France*
[2] *INRIA - LIP, ENS Lyon, 46, allée d'Italie, F-69364 Lyon Cedex 07, France*
[3] *Eurecom, 2229, route des Cretes, BP 193, 06901 Sophia Antipolis Cedex, France*
[4] *Faculty of Computer Science, Dalhousie University, P.O. Box 1000, Halifax NS, Canada B3J 2X4*

SUMMARY

This paper describes efficient coarse-grained parallel algorithms and implementations for a suite of interval graph problems. Included are algorithms requiring only a constant number of communication rounds for connected components, maximum weighted clique, and breadth-first-search and depth-first-search trees, as well as $O(\log p)$ communication rounds algorithms for optimization problems such as minimum interval covering, maximum independent set and minimum dominating set, where p is the number of processors in the parallel system. This implies that the number of communication rounds is independent of the problem size. Implementations of these algorithms are evaluated on parallel clusters, using both Fast Ethernet and Myrinet interconnection networks, and on a CRAY T3E parallel multicomputer, with extensive experimental results being presented and analyzed. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: coarse grain; interval graphs; parallel algorithms; practical experiments

1. INTRODUCTION

In this paper we describe coarse-grained parallel algorithms and implementations for a suite of standard interval graph problems [1]. This suite includes solutions to fundamental interval graph problems such as connected components [2], maximum weighted clique [3], breadth-first-search (BFS) trees and depth-first-search (DFS) trees [4,5], as well as solutions for optimization problems such as minimum interval covering, maximum independent set [3] and minimum dominating set [6]. These problems on interval graphs have been shown to be important in a number of applications in scheduling, circuit design, traffic control, genetics and other problem domains [1,7,8].

*Correspondence to: A. Ferreira, CNRS, Projet Mascotte, BP 93, F-06902 Sophia Antipolis, France.
†E-mail: ferreira@sophia.inria.fr

Received 28 November 2000
Revised 3 March 2002

Table I. Interval graph problems and the required time for PRAM algorithms with n processors, unless stated otherwise. The sequential complexity is $\Theta(n \log n)$ in all cases. Note that for most of the PRAM algorithms, the number of processors is the same as the problem size.

Problem	Time
Maximum weighted clique [3]	$O(\log(n))$
Maximum independent set [3]	$O(\log(n))$
Minimum clique cover [3,6]	$O(\log(n))$
Minimum dominating set [3,6]	$O(\log(n))$
Depth-first spanning tree [4,5]	$O(\log(n))$
Breadth-first spanning tree [4,5]	$O(\log(n))$
Connected components (PRAM CRCW) [2] (using $\{[[m+n]\alpha(m,n)]/\log n\}$ processors, where $\alpha(m,n)$ is the inverse of the Ackerman function)	$O(\log(n))$

The implementations described in this work are portable over a range of types of coarse grained parallel machines, which is demonstrated via experimental results. Our primary focus here is on the use of PC clusters which offer impressive computing and communication power at a reasonable price when interconnected by high-performance local networks with raw throughput close to 1 Gb s^{-1} and latency smaller than 10 μs. Clusters, henceforth called PCCs, based on off-the-shelf hardware (e.g. Myrinet and Fast Ethernet) can yield effective parallel systems for a fraction of the price of machines using special purpose hardware. For the sake of comparisons, we also experimentally analyze some of our algorithms on the Cray T3E, whose implementation was straightforward because of the portability of our codes.

Hereinafter, let n denote the number of vertices (or intervals) of our graphs; m be the number of their edges; and p denote the number of computing elements in the parallel system.

1.1. Previous work

Interval graphs have several applications in different fields. Some of these applications, as in genetics [9,10,11], often require the use of very large databases. Handling these databases on a single processor can be very long and the use of parallel machines may drastically reduce processing time. Therefore, interval graphs have been studied extensively in the parallel setting and a number of fine-grained PRAM algorithms have been described [1], as shown in Table I. In particular, for the problems we investigated, $O(\log(n))$ time algorithms exist [2–6].

Whereas fine-grained PRAM algorithms are likely to be efficient on fine-grained shared memory architectures, they tend to be impractical on PCCs due to their failure to exploit locality. Therefore, there has been a growth of interest in coarse-grained computational models [12–15] and the design of coarse-grained algorithms [13, 16–26].

With respect to models, the 'bulk synchronous' processing (BSP), described by Valiant [15], uses slackness in the number of processors and memory mapping via hash functions to hide communication latency and provide for the efficient execution of fine grained PRAM algorithms on coarse-grained hardware. Culler *et al.* introduced the LogP model [12] which, using Valiant's BSP model as a starting point, focuses on the technological trend from fine-grained parallel machines towards coarse-grained systems and advocates portable parallel algorithm design. Other coarse-grained models focus more on utilizing local computation and minimizing global operations. These include the coarse-grained multicomputer (CGM) model [13] used in this paper. In this mixed sequential/parallel setting, there are three important measures of any coarse-grained algorithm, namely, the amount of local computation required, the number and type of global communication phases or rounds required and the scalability of the algorithm, that is, the range of values for the ratio (n/p) for which the algorithm is efficient and applicable.

Recently, Cáceres *et al.* [27] showed that many problems in general graphs, such as list ranking, connected components and others, can be solved in $O(\log p)$ communication rounds in BSP and CGM. Note that while this work is of significant theoretical interest, the proposed algorithms involve simulation of their corresponding PRAM algorithm during $\log p$ communication phases, which is both complex to implement and computationally expensive in practice. Hence, whereas in theory these results yield algorithms for interval graphs on coarse-grained machines, in practice a different approach is called for. Unlike general graphs, interval graphs can be more easily partitioned and treated in the distributed memory setting. Since each interval is given by its two extreme points, they can be sorted by left and/or right endpoints and distributed according to this ordering. This partitioning allows us to design less complex parallel algorithms; moreover, the derived algorithms are faster both in theory and in practice, and easier to implement (see evidence of this in Section 4.2 and in [28]).

1.2. Our work

We describe constant communication round coarse-grained parallel algorithms to solve a set of the standard problems arising in the context of interval graphs [1], namely connected components [2], maximum weighted clique [3] and breadth-first-search (BFS) and depth-first-search (DFS) trees [4,5]. We also propose $O(\log p)$ communication rounds algorithms for optimization problems such as minimum interval covering, maximum independent set [3] and minimum dominating set [6]. Note that the number of communication rounds is independent of n.

In order to demonstrate the practicability of our approach, we present the implementation results of one of these algorithms on two PCCs, one of which is interconnected by a Fast Ethernet backbone and the other interconnected by a Myrinet network, and on a CRAY T3E parallel multicomputer. Because of the paradigms used, the programs were easy to develop and are portable. The results presented in this paper show that high performance can be achieved with off-the-shelf PCCs along with the right model for algorithm design[‡]. We can also draw a comparison of performances between the PCCs and the T3E. Interestingly, super-linear speedups were observed in some cases due to memory swapping

[‡]A preliminary version of this paper was published in [29]. In the present paper, we provide all proofs omitted in [29]. Furthermore, we show through new experimental results on two other machines the portability and efficiency of the proposed algorithms.

effects in the PCCs. Indeed, using multiple processors allows us to effectively utilize more RAM and therefore allows computation on data sets that are simply too large to be effectively processed on single processor machines.

This paper is organized as follows. In Section 2 we review the coarse-grained model adopted in this paper. In Section 3 the basic operations are described. Then, in Section 4, solutions for interval graph problems are proposed, using the basic operations from Section 3. In Section 5, we describe experiments on a Fast Ethernet based PCC, on a Myrinet based PCC and on a CRAY T3E parallel multicomputer. We end the paper with some conclusions and directions for further research.

2. THE COARSE-GRAINED MODEL

In a BSP model, an input of size n is distributed evenly across a p-processors parallel computer. In a single computation round or superstep each processor may send and receive h messages and then perform an internal computation on its internal memory cells using the messages it has just received. To avoid conflicts that might be caused by asynchronies in the network (whose topology is left undefined) the messages sent out in a round t by some processor cannot depend upon any message that the processor receives in round t.

In this paper we use the Coarse-Grained Multicomputer model, or CGM(n, p) for short, introduced in [13]. The CGM(n, p) is a BSP model consisting of a set of p processors with $O(n/p)$ local memory each. The term 'coarse grained' refers to the fact that (as in practice) $O(n/p)$ is defined to be 'considerably larger' than $O(1)$. The definition of 'considerably larger' is $(n/p) \geq p^{\epsilon}$, where ϵ depends on the proposed algorithms; in this paper $\epsilon = 1$. In a superstep, each processor may send or receive $O(n/p)$ data.

From both theoretical and practical viewpoints, it is important to show that the proposed algorithms are independent of the parallel machines communication topology and that they use only a small, preferably constant, number of communication rounds. The challenge is thus to design correct algorithms which communicate only a few times, while taking into account the local memory restriction. The good point is that one such algorithm will be portable and efficient over a large range of multicomputers.

3. BASIC OPERATIONS

3.1. General operations

Sorting can be used as a fundamental data movement operation in the parallel setting. From a theoretical perspective it has been shown that sorting can be done in the CGM model in a constant number of communication rounds.

Theorem 1. [24] *Given a set S of n items, $O(n/p)$ items per processor on a* CGM(n, p), *$n/p \geq p$, sorting S takes a constant number of communication rounds, and $O(t_{\text{seq}}(n/p)/p)$ local time (where $t_{\text{seq}}(n)$ is the time of sequential sorting of n items).*

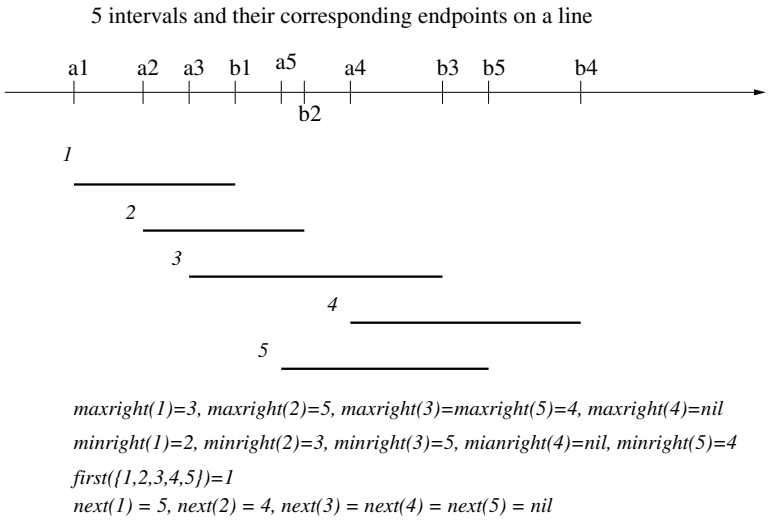


Figure 1. An example of the maxright, minright, first and next functions.

In practice, which sorting algorithm is most efficient depends on the parallel machines communication topology and other system specific features. Also the amount of data to be communicated may affect the selection of a basic sort procedure. In the experimental results reported in this paper we use a number of different sorting algorithms [30,31]. In the remainder, let $T_S(n, p)$ denote the time complexity of a global sort in the CGM.

In the CGM model, **parallel prefix operations** for associative function to be performed in an array of elements can be done in $O(1)$ communication steps, since each processor can locally compute the function, and then with a total exchange operation all the processors get to know the partial result of all the other processors and can compute the final result for each element in the array.

We will also use the **pointer-jump operation**, to identify the elements in a linked list. This operation can be done in $O(\log p)$ communication steps [27]: at each step each processor keeps track of the pointers of its elements.

3.2. Interval operations

We consider a set of n intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ on a line.

In the following, two functions will be widely used, the minright and maxright [7]. Given an interval I, maxright(I) (minright(I)) denotes, among all intervals that intersect the right endpoint of I, the one whose right endpoint is the furthest right (left) (see Figure 1). The formal definition is as follows:

$$\text{maxright}(I_i) = \begin{cases} I_j, & \text{if } b_j = \max\{b_k | a_k \leq b_i < b_k\} \\ \text{nil}, & \text{otherwise} \end{cases}$$

Input: n intervals I_i $(1 \leq i \leq n)$ (n/p intervals on each processor)
Output: maxright(I_i) $(1 \leq i \leq n)$

1 Global sort of the endpoints of the intervals in ascending order
2 **foreach** $i \in [1, 2n]$ **do**
assign to endpoint c_i the value d_i defined by

$$d_i = \begin{cases} b_j, & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n \\ 0, & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n \end{cases}$$

3 **foreach** $i \in [1, 2n]$ **do**
assign to endpoint c_i the value num_i defined by

$$\text{num}_i = \begin{cases} -j, & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n \\ j, & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n \end{cases}$$

4 Compute the prefix maximum on the d_i, and let the result in $e_1, e_2, \ldots, e_{2n}$ and in the same time update the value num_i according to the following rule: if $e_i \neq d_i$ and $i > 1$ set

$$\text{num}_i = \begin{cases} -|\text{num}_{(i-1)}|, & \text{if } c_i = a_j \text{ for some } 1 \leq j \leq n \\ |\text{num}_{(i-1)}|, & \text{if } c_i = b_j \text{ for some } 1 \leq j \leq n \end{cases}$$

5 **foreach** $i \in [1, 2n]$ **do**
set maxright(I_k) $= I_j$, if $c_i = b_k$ and $\text{num}_i = j$ and $k \neq j$
set maxright(I_k) $= nil$, if $c_i = b_k$ and $\text{num}_i = j$ and $k = j$

Algorithm 1. maxright.

One way to compute the function maxright (and minright with the appropriate variations) is given in Algorithm 1.

After step 1 of Algorithm 1, we know that all the left endpoints of the intervals intersecting I_i $(1 \leq i \leq n)$ are on the left of its right endpoint b_i. Due to the definition of d_i $(1 \leq i \leq n)$ and of the prefix maximum on d_i at step 4, we are sure that for all the right endpoints b_i $(1 \leq i \leq n)$, e_i gives the right endpoint the furthest right of the intervals which intersect I_i and that num_i gives the number of the associated interval, that is to say maxright(I_i). We keep negative values for num_i $(1 \leq i \leq 2n)$ for left endpoints in order to be able at step 2 to distinguish the left endpoints from the right endpoints.

Step 1 requires $O(T_S(n, p))$ time, whereas all the other steps require $O(n/p)$ local computations. Step 1 and step 4 use a constant number of communications rounds. Then, maxright (and minright with the appropriate modifications) can be computed with time complexity $O(T_S(n, p))$ and a constant number of communications rounds.

Input: n intervals I_i $(1 \leq i \leq n)$ (n/p intervals on each processor)
Output: next(I_i) $(1 \leq i \leq n)$

1 Global sort of the endpoints of the intervals in ascending order
2 **foreach** $i \in [1, 2n]$ **do**
  assign to endpoint c_i the value d_i defined by

$$d_i = \begin{cases} b_j, & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n \\ 0, & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n \end{cases}$$

3 **foreach** $i \in [1, 2n]$ **do**
  assign to endpoint c_i the value num_i defined by

$$\text{num}_i = \begin{cases} -j, & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n \\ j, & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n \end{cases}$$

4 Compute the suffix minimum on the d_i, and let the result in $e_1, e_2, \ldots, e_{2n}$ and in the same time update the value num_i according the following rule: if $e_i \neq d_i$ and $i > 1$ set

$$\text{num}_i = \begin{cases} -|\text{num}_{(i+1)}|, & \text{if } c_i = a_j \text{ for some } 1 \leq j \leq n \\ |\text{num}_{(i+1)}|, & \text{if } c_i = b_j \text{ for some } 1 \leq j \leq n \end{cases}$$

5 **foreach** $i \in [1, 2n]$ **do**
  set $\text{next}(I_k) = I_j$, if $c_i = b_k$ and $\text{num}_i = j$ and $k \neq j$
  set $\text{next}(I_k) = \text{nil}$, if $c_i = b_k$ and $\text{num}_i = j$ and $k = j$

Algorithm 2. next.

We define also the parameter first($\mathcal{I}$) as the segment I which 'ends first', that is, whose right endpoint is the furthest left (see Figure 1):

$$\text{first}(\mathcal{I}) = I_j, \quad \text{with } b_j = \min\{b_i | 1 \leq i \leq n\}$$

To compute it, we need only to compute the minimum of the sequence of right endpoints of intervals in the family $\mathcal{I}$.

We will also use the function $\text{next}(I) : \mathcal{I} \to \mathcal{I}$ defined as

$$\text{next}(I_i) = \begin{cases} I_j, & \text{if } b_j = \min\{b_k | b_i < a_k\} \\ \text{nil}, & \text{otherwise} \end{cases}$$

That is, $\text{next}(I_i)$ is the interval that ends farthest to the left among all the intervals beginning after the end of I_i (see Figure 1). To compute $\text{next}(I_i)$, $1 \leq i \leq n$, we use the same algorithm used for $\text{maxright}(I_i)$ (Algorithm 1), with a new step 4. Algorithm 2 computes the function next.

It is easy to see that the given procedure implements the definition of $\text{next}(I_i)$, with the same complexity as for computing $\text{maxright}(I_i)$.

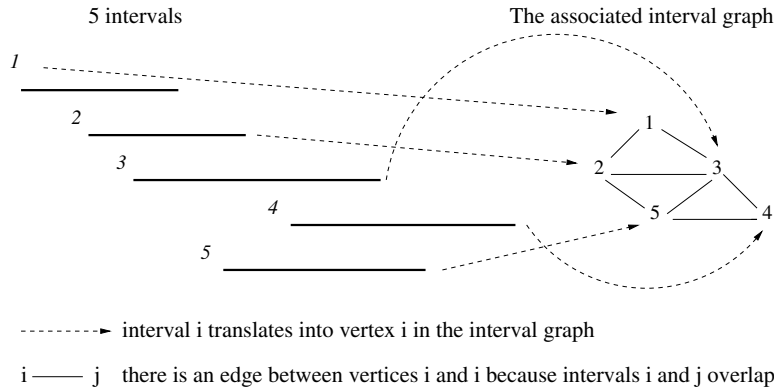


Figure 2. A set of five intervals and the corresponding interval graph.

4. INTERVAL GRAPH PROBLEMS AND ALGORITHMS

Formally, given a set n of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ on a line, the corresponding interval graph $G = (V, E)$ has the set of nodes $V = \{v_1, \ldots, v_n\}$, and there is an edge in E linking nodes v_i, v_j if and only if $I_i \cap I_j \neq \emptyset$. See Figure 2 for an example.

In this section, solutions for some important problems in interval graphs are proposed for the CGM model. Some of these algorithms use techniques derived from their corresponding PRAM algorithms, while others require different methods, e.g. to compute the connected components, as now shown.

4.1. Maximum weighted clique

A clique is a set of nodes that are mutually adjacent. Usually, finding cliques in general graphs is non-deterministic polynomial (NP)-hard. However, the properties of interval graphs make the problem polynomially solvable. In the maximum weighted clique problem for an interval graph, we are given weights $p(I_i) \geq 0$ on the intervals and we want to find a clique for which the sum of its nodes weights is maximum among all cliques. Finally, the nodes composing such a maximum clique should be identified.

In Figure 3, each interval has an associated weight (above the interval) and you can see the maximum weighted clique of the example. Algorithm 3 computes a maximum weighted clique of an interval graph in the CGM model.

Theorem 2. *The maximum weighted clique problem in an interval graph of size n can be solved on a* CGM(n, p) *in* $O(T_S(n, p) + n/p)$ *time, with a constant number of communication rounds.*

Proof. Algorithm 3 uses techniques derived from the algorithm in [32]. After step 3, if $c_i = a_j$, for some $1 \leq j \leq n$, then d_i gives the sum of the weights of all intervals containing the point c_i (the weights of all intervals on the left of c_i not containing the point c_i are not counted in d_i due to the

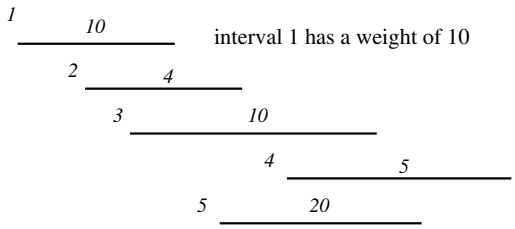


Figure 3. An example of the maximum weighted clique problem.

Input: n intervals I_i $(1 \leq i \leq n)$ $(n/p$ intervals on each processor)

Output: The intervals belonging to the maximum weighted clique are marked, the others are not

1 Global sort of the endpoints of the segments that each processor receives

2 **foreach** $i \in [1, 2n]$ **do**

assign to endpoint c_i a weight w_i defined by

$$w_i = \begin{cases} p(I_j), & \text{if } c_i = a_j, \text{ for some } 1 \leq j \leq n \\ -p(I_j), & \text{if } c_i = b_j, \text{ for some } 1 \leq j \leq n \end{cases}$$

3 Compute the prefix sum of the resulting weighted sequence; let $d_1, \ldots, d_{2n}$ denote the resulting sequence

4 Consider the sequence $e_1, \ldots, e_{2n}$ obtained by replacing every d_j corresponding to a right endpoint of an interval with -1 and compute the rightmost maximum of the resulting sequence; this occurs at a_k

5 Broadcast a_k. Every interval I_u such that $a_u \leq a_k < b_u$ is marked to be in the final maximum weighted clique

Algorithm 3. Maximum weighted clique.

negative weight associated to the right endpoint of an interval). The set of all these intervals containing c_i with the interval I_j is a clique, and d_i is the weight of this clique.

After step 4, the rightmost maximum d_i corresponds to the weight of a maximum weighted clique. The associated c_i is equal to an a_k $(1 \leq k \leq n)$, because if it was a b_j $(1 \leq j \leq n)$, d_i would not be a maximum. This a_k is the furthest right left-endpoint in this maximum weighted clique, because otherwise d_i would not be a maximum. Thus, we are sure that all intervals belonging to the maximum weighted clique have a left endpoint lower than a_k. This shows the correctness of Algorithm 3. □

Input: n intervals I_i $(1 \leq i \leq n)$ (n/p intervals on each processor)
Output: Each interval has the number of the connected component it belongs to (given by val)

1 Global sort of the endpoints in ascending order. Assign to each endpoint the value of its interval
2 Assign the value 1 to each left endpoint and the value -1 to each right endpoint
3 Compute the prefix sum of the endpoints. The result is stored in an array L
4 **foreach** Processor P_i **do**
  If there is a 0 in L then send your identification i to all processors
5 Let $j_1, j_2, \ldots, j_q; q \leq p$ be the list of identifications received.
counter $= 0$
foreach Processor P_i **do**
  for $j = 1$ *to* $2n/p$ **do**
    if $L[j] == 0$ **then**
      val$[j]$ = counter.i; counter = counter $+ 1$
    else
      val$[j]$ = counter.i
  $j = 2n/p$
  while $L[j] > 0$ **do** val$[k]$ = j_k.0, where $j_k \leq i < j_{k+1}; k = k - 1$

Algorithm 4. Connected components.

Step 1 requires time $O(T_S(n, p))$, step 2 requires time $O(n/p)$ and steps 3, 4 and 5 require $O(n/p)$ local computations and a constant number of communication rounds. To summarize we have a running time of $O(T_S(n, p) + n/p)$, with a constant number of communication rounds.

4.2. Connected components

The connected components of a graph G are the maximal connected subgraphs of G. The connected components problem consists of assigning to each node the label of the connected component that contains it.

Algorithm 4 solves this problem in the CGM model.

Theorem 3. *The connected components problem in interval graphs can be solved on a* CGM(n, p) *in* $O(T_S(n, p) + n/p)$ *time, with a constant number of communication rounds.*

Proof. The sort at step 1 of Algorithm 4 ensures that there cannot be an endpoint of an interval belonging to a connected component A between two endpoints of intervals belonging to a connected component B, with $A \neq B$. Therefore, the endpoints of intervals belonging to the same connected component are stored in a continuous way. The prefix sum at step 4 ensures that a 0 in L completes a connected component and that the following element belongs to a new connected component. Step 6 labels the components: the for-loop labels the local components, whereas the while-loop labels the components that are divided on several processors. This labelling ensures

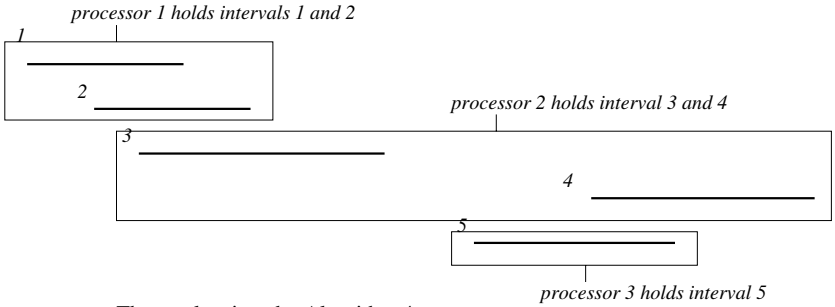


Figure 4. An example of the connected components problem.

that the computed connected components of the interval graph have different labels. At the end of Algorithm 4, all connected components have a sole label.

With respect to the time complexity, Algorithm 4 requires $O(T_S(n, p))$ in step 1, and $O(n/p)$ in steps 2, 4, 5 and 6. In total we have $O(T_S(n, p) + n/p)$, with a constant number of communication steps. □

Figure 4 gives an example of the connected components problem. Each interval belongs to a processor before the execution of Algorithm 4, as shown. After step 1, some endpoints may have changed processor, which explains the values of the connected components for intervals 4 and 5.

4.3. BFS and DFS tree

Figure 5 gives an example of BFS and DFS trees of an interval graph. Algorithm 5 solves the problem of finding a BFS tree in an interval graph.

Theorem 4. *Given an interval graph G, BFS and DFS trees can be found using a* CGM(n, p) *in* $O(T_S(n, p) + n/p)$ *time, with a constant number of communication rounds.*

Proof. We adapt the techniques given in [5], ensuring that we use only functions that require a small number of communication steps. The problem of finding a BFS tree in an interval graph reduces to the problem of computing the function maxright described earlier. The tree given by the edges $(I_i, \text{maxright}(I_i))$ is a BFS tree. Suppose we have a set of intervals S which have the same result I_k (for some $1 \leq k \leq n$) for *maxright* (and thus have the same father in the tree we construct). It may exist some edges between some intervals of S in the interval graph different from the edges with I_k. Choosing maxright for each interval as the father in the tree ensures that all these edges are eliminated in the tree we construct and that all intervals of S are connected to the same interval. Thus, we are sure that the constructed tree is a BFS tree.

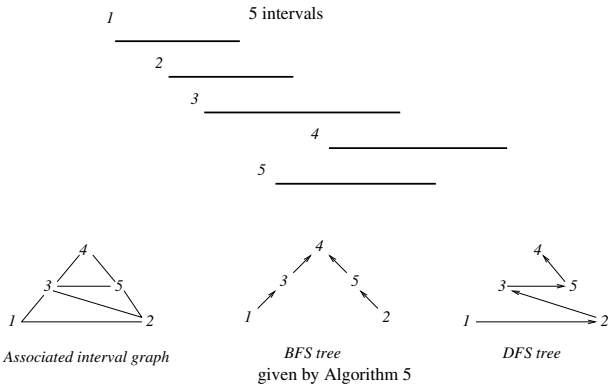


Figure 5. Examples of BFS and DFS trees.

Input: n intervals I_i $(1 \leq i \leq n)$ $(n/p$ intervals on each processor)
Output: a BFS tree

1 **foreach** $i \in [1, n]$ **do**
  compute maxright(I_i)
2 **foreach** $i \in [1, n]$ **do**
  let father(I_i) = maxright(I_i)
3 The edges $(I_i, \text{father}(I_i))$ $(1 \leq i \leq n)$ form a BFS tree

Algorithm 5. BFS tree.

The tree formed by the edges $(I_i, \text{minright}(I_i))$ is a DFS tree. Choosing minright for each interval as its father in the tree, ensures that all the intervals belonging to a clique will have a different father and thus the constructed tree is a DFS tree. With the appropriate modifications, Algorithm 5 may be used to find a DFS tree.

The obtained BFS and DFS trees have their roots in the segments ending farthest to the right in each connected component.

Given the complexity of maxright (or minright), Algorithm 5 takes a constant number of communication steps and requires a total running time of $O(T_S(n, p) + n/p)$. □

4.4. Minimum interval covering

Given a family $\mathcal{I}$ of intervals and a special interval $J = (J_a, J_b)$, the problem of the minimum interval covering is to find a subset $\mathcal{J} \subseteq \mathcal{I}$ such that $J \subseteq \cup(\mathcal{J})$, and $|\mathcal{J}|$ is minimum; i.e. to find the minimum number of intervals in $\mathcal{I}$ needed to cover J. To solve this problem we may only consider the intervals

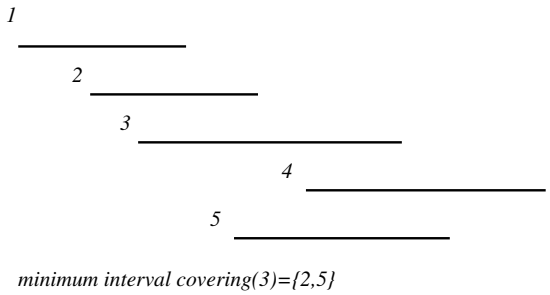


Figure 6. An example of the minimum interval covering problem.

Input: all the intervals belonging to $\mathcal{I}_J$ and the interval J to be covered
Output: only the intervals belonging to a minimum interval covering $\mathcal{J}$ of J are marked

1 **foreach** $i \in [1, n]$ **do**
  compute maxright(I)
2 Find the interval I_{init} such that $b_{\text{init}} = \max\{b_k | a_k \leq J_a\}$
3 Mark all the intervals in the list given by maxright and beginning at I_{init}

Algorithm 6. Minimum interval covering.

$I_i = (a_i, b_i) \in (\mathcal{I})$ such that $b_i \geq J_a$ and $a_i \leq J_b$. Let $\mathcal{I}_J$ be the family of the intervals in $\mathcal{I}$ satisfying this condition.

Figure 6 gives an example of the minimum interval covering problem. Algorithm 6 solves this problem in the CGM model.

Theorem 5. *The minimum interval covering problem in interval graphs can be solved using a* CGM(n, p) *in* $O(T_S(n, p) + \log p)$ *time, with* $O(\log p)$ *communication rounds.*

Proof. To confirm the correctness of Algorithm 6, suppose that there is another family $\mathcal{J}' \subseteq \mathcal{I}_J$ such that $J \subseteq \cup(\mathcal{J}')$, and $|\mathcal{J}'| < |\mathcal{J}|$. Choose such a family $\mathcal{J}'$ where $\mathcal{J} \cap \mathcal{J}'$ is maximum. Let $\mathcal{J} = \{J_1, \ldots, J_m\}$ and $\mathcal{J}' = \{J'_1, \ldots, J'_{m'}\}$, $m > m'$.

Let J_k be the first interval in $\mathcal{J}$ which is not in $\mathcal{J}'$, i.e. k is such that

$$k = \min\{i | J_l = J'_l, \text{ for all } l \text{ such that } 1 \leq l < i\}.$$

By definition, $J_k \cap J_{k-1} \neq \emptyset$ and $J'_k \cap J_{k-1} \neq \emptyset$. Hence, since $O\text{maxright}(J_{k-1}) = J_k$, the right-endpoint of J'_k is to the left of the right-endpoint of J_k (if $k = 1$, then by the definition of I_{init} we get the same conclusion). We can now replace J'_k with J_k in $\mathcal{J}'$, which is a contradiction to the hypothesis that $|\mathcal{J} \cap \mathcal{J}'|$ is maximum.

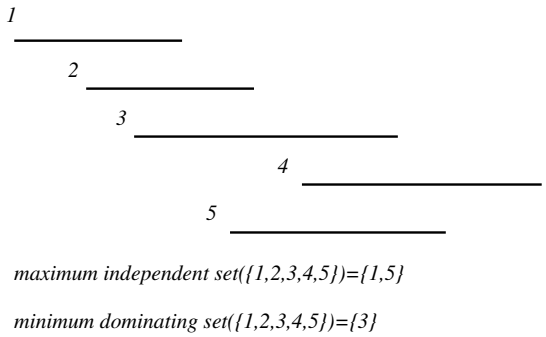


Figure 7. Examples of maximum independent set and minimum dominating set problems.

Input: n intervals I_i $(1 \leq i \leq n)$ $(n/p$ intervals on each processor)
Output: only the intervals belonging to a maximum independent set are marked

1 Compute first($\mathcal{I}$)
2 **foreach** $i \in [1, n]$ **do**
  compute next(I_i)
3 **foreach** $i \in [1, n]$ **do**
  Let father(I_i) = next(I_i)
4 Using the pointer-jump operation, mark all the intervals in the linked list given by father and beginning at first($\mathcal{I}$)

Algorithm 7. Maximum independent set.

Note that step 1 requires $O(T_s(n, p))$ time, step 2 requires $O(T_S(n, p))$ time and step 3 uses pointer-jump and therefore $O(\log p)$ communication rounds. The total time is thus $O(T_s(N, p) + \log p)$ with $O(\log p)$ communication rounds. □

4.5. Maximum independent set and minimum dominating set

The first problem consists of finding a largest set of mutually non-overlapping intervals in the family $\mathcal{I}$, called the maximum independent set. The second problem consists of finding a minimum dominating set, i.e. a minimum set of intervals which are adjacent to all remaining intervals in the family $\mathcal{I}$. As in the case of cliques, these problems are in general NP-hard, but polynomial in interval graphs. To solve these problems, we give a coarse-grained implementation of the algorithms proposed in [6]. Both problems are based on first($\mathcal{I}$) from which a convenient linked list is built.

Figure 7 gives an example of maximum independent set and minimum dominating set problems. Algorithm 7 solves the maximum independent set problem, while Algorithm 8 solves the minimum dominating set problem in the CGM model.

Input: n intervals I_i $(1 \leq i \leq n)$ $(n/p$ intervals on each processor)
Output: only the intervals belonging to a minimum dominating set are marked

1 Compute first($\mathcal{I}$)
2 **foreach** $i \in [1, n]$ **do**
  compute maxright(I_i)
3 **foreach** $i \in [1, n]$ **do**
  compute next(I_i)
4 **foreach** $i \in [1, n]$ **do**
  Let father(I_i) = maxright(next(I_i))
5 Using the pointer-jump operation, mark all the intervals in the linked list given by father and beginning at maxright(first($\mathcal{I}$))

Algorithm 8. Minimum dominating set.

Theorem 6. *The maximum independent set problem and the minimum dominating set in interval graphs can be solved using a* CGM(n, p) *in* $O(T_S(n, p) + \log p)$ *time, with* $O(\log p)$ *communication rounds.*

Proof. The correctness of Algorithm 7 stems from a result in [33] that shows that there exists a maximum size independent set S in $\mathcal{I}$ such that first($\mathcal{I}$) $\in \mathcal{S}$ and for every I_i belonging to S, next(I_i) also belongs to S.

For the complexity of Algorithm 7, step 1 requires $O(n/p)$ time, steps 2 and 3 require $O(T_S(n, p))$ time, whereas step 3 uses pointer-jump and therefore $O(\log p)$ communication rounds, giving us a total time complexity of $O(T_S(n, p) + \log p)$ and $O(\log p)$ communication rounds.

The correctness of Algorithm 8 stems from the arguments in [6]. For the complexity of Algorithm 8, step 1 requires $O(n/p)$ time, step 2 and step 3 require $O(T_S(n, p))$ time, step 4 requires $O(n/p)$ time and only one communication round, and step 5 uses pointer jumping and therefore $O(\log p)$ communication rounds. So, the total time complexity is $O(T_S(n, p) + \log p)$ and there are $O(\log p)$ communication rounds. □

5. EXPERIMENTAL RESULTS

Our aim here is to demonstrate that these algorithms are not only theoretically efficient but that they lead to simple fast codes in practice.

We implemented our connected components, maximal weighted clique and BFS algorithms. Their portability is witnessed by their easy implementation on three different multicomputer systems, with different architectures as explained later. For the three algorithms, the implementations behave as expected. In order to compute the experimental speedup, we compare the parallel algorithms with the usual sequential algorithm for each problem. We obtained efficient results for the three implementations: the connected components and maximal weighted clique algorithms have a speedup

of around $p/1.6$, whereas the BFS algorithm has a speedup of around $p/2$. The speedups are an average on p and n. Actually, the speedups are a little larger for small p and a little smaller for large p. Note that it was also possible to handle very large data sets with the three algorithms.

In the following we decided to explore the results of one of the algorithms. We describe and analyze in detail the implementation of the maximal weighted clique algorithm (Algorithm 3) for interval graphs. It was implemented on two PCC platforms and a parallel multicomputer. The first PCC ([34]) consisted of a set of eight Pentium Pro 200 MHz processors each with 64 M of RAM that were linked by a fast Myrinet network. We refer to it as POPC in the following. The second PCC ([35]) consisted of a set of 12 Pentium Pro 200 MHz processors each with 128 M of RAM that were linked by a 100 Mb s^{-1} Fast Ethernet network. Henceforth, it is called PF. The processors of the two platforms ran the Linux operating system. The parallel multicomputer used was a Cray T3E with 32 DEC Alpha processors running at 300 MHz [36]. The programs were written in C++ utilizing the portable PVM communication library [37] for all interprocessor communications. The use of three different machines helped us explore portability issues over a range of architectures and interconnection networks.

All the tests were carried out ten times for each input of the same size. The given results are an average of the ten tests. All the execution times reported are given in seconds. The times have been measured by the system function *gettimeofday*, which we have found to be the most reliable. The communication time corresponds to the time required for the complete communication rounds. We do not only measure the time for data to be communicated, but also all the time required supporting operations (e.g. buffer management or rearranging data, etc.). For this measure, we have added a synchronization barrier (which is not necessary for the correct execution of the code) just before the starting of the processors' chronometer, and thus we are sure that all the processors have finished their local computation round before entering the communication round.

In all of the figures, the x-axis represents the number n of elements in the input (the number of intervals), and the y-axis represents the execution time in seconds (except for the speedup). The handled elements are integers.

Since Algorithm 3 relies on sorting, the choice of the sorting method was critical. In the following we first present the implemented sort and its performance before describing the implementation and performance of our algorithm.

5.1. Global sort

The sorting algorithm implemented is described in [31]. It was selected due to its excellent communications properties and for the ease with which it can be implemented. The algorithm requires a constant number of communication steps and its single drawback is that data may not be equally distributed at the end of the sort. Nevertheless, a partial sum procedure and a routing can be used to redistribute the data with a constant number of communication rounds so that each processor stores n/p data in its memory. The sequential algorithm used is the classical counting sort [30]. We give the results of this sequential algorithm to give a baseline against which to evaluate the performance of our parallel sort.

Figure 8 shows the execution time for the global sort on an array of integers with one, two, four and eight PCs for POPC and PF and for 12 PCs for PF compared with the performance of the sequential sort. Each curve stops before the PCs' swap limit. The measures begin with 1 million elements in order

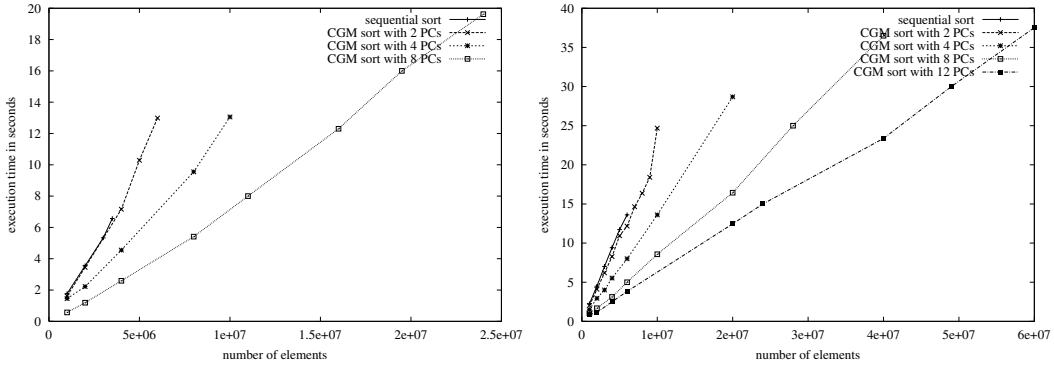


Figure 8. Sorting on the PCCs (POPC and PF).

to satisfy the constraints of the algorithm given in [31]. This bound on the number of elements is not really a drawback because the sort of less than 1 million elements is done very quickly sequentially and does not require a parallel sort.

PF can handle twice as many elements as POPC because it has twice the memory. Thus, POPC can sort a bit more than 20 million elements with eight PCs, whereas PF can sort 40 million elements.

The execution times decrease when two, four, eight and 12 PCs are used. The obtained speedup is around 1.75 with four PCs, 3 with eight PCs and 3.75 with 12 PCs. Note that it is easy to show that the theoretical speedup cannot be larger than $(3/4)p$. There are $3n$ local computations in the sequential sort and the parallel sorting algorithm requires at least $4(n/p)$ local computations. Moreover, a local $\log p$ factor that appears in one step of this parallel sorting algorithm reduces the speedup when p increases.

The memory swapping effects significantly increase the execution time. On the PF, one can reasonably sort 7 million elements sequentially, whereas two PCs can sort 12 million elements, four PCs 23 million elements, eight PCs 44 million elements and 12 PCs 60 million elements. The sort on 60 million elements is done in less than 40 s. POPC has the same behavior except it can handle half the data due to its memory size.

Figure 9 gives the execution times for one, 12 and 32 processors of the T3E. It is exactly the same code on the T3E and on the PCCs. Each curve stops before the memory becomes saturated. To satisfy the constraints given by the algorithm of [31], the measures begin with 1 million elements for 12 processors and with 8 million elements for 32 processors.

As for the PCC platforms, the more processors we have, the lower the execution times. A speedup of about 3.5 is obtained with 12 processors. We cannot compare the sequential execution time and the execution time of 32 processors because the processor's memory is saturated after 4 million elements. Twelve processors can sort 25 million elements whereas 32 processors can sort 110 million elements.

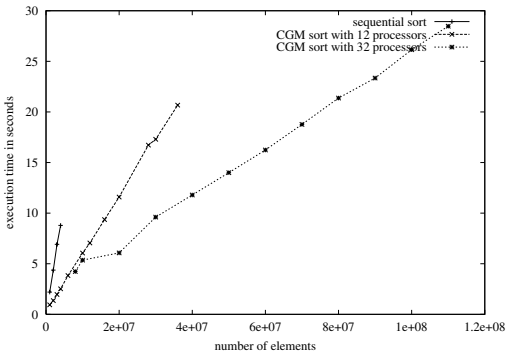


Figure 9. Sorting on the T3E.

5.2. Maximum weighted clique

The intervals encoding is done with an array of structures with two fields, the first field corresponding to the left-endpoint and the second one to the right-endpoint. Each processor owns an array of size n/p which contains a part of intervals.

We used random permutations to generate inputs. For each test, we generated a permutation of size $2n$ and the endpoints of intervals are the elements of this permutation. The elements are unsigned long integers. The intervals' weights were chosen randomly. Quicksort was used for the sequential sorting rather than counting sort due to the large key ranges involved.

5.2.1. PCC platforms

We give in parallel the results obtained on POPC and PF. The left figure corresponds to POPC and the right one to PF.

Influence of p. Figure 10 gives the execution times on one, four and eight PCs of POPC and on one, four and 12 PCs of PF. For POPC, n ranges from 500 000 to 6 million intervals, whereas for PF n ranges from 500 000 to 20 million. Each curve begins at 500 000 intervals because the sort is done on the two endpoints of intervals, that is to say on 1 million data, which is one of the constraints of the chosen sorting algorithm. Each curve stops before the swap limit of the PCs.

First, we can see that the curves are almost linear in n, which is not surprising because all the steps require only $O(n/p)$ operations except for the sequential quicksort. Moreover, the more PCs we have, the faster the program is, which is as expected.

Table II shows the different speedups obtained for a problem with 1 million intervals in relation to the number of processors used. For this problem, the speedups are good (larger than $p/2$). Table III shows the speedups obtained with four processors with n varying (from 500 000 to 1 million). They increase with the input size. We can note that these speedups are better than those presented for the parallel

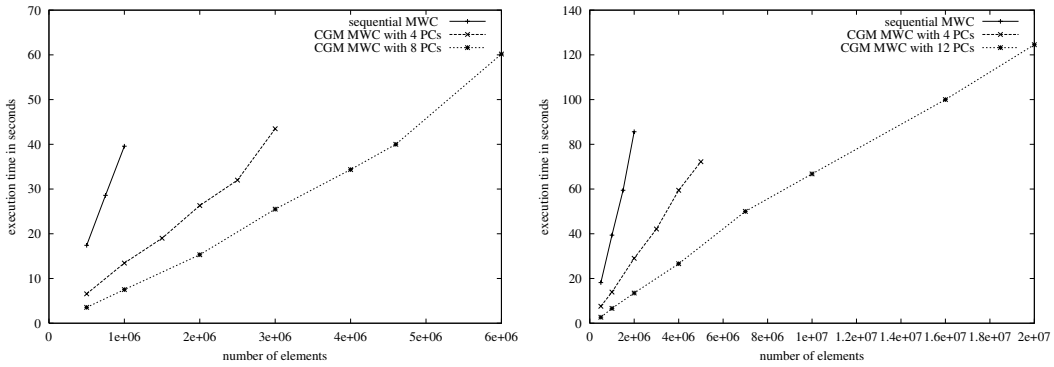


Figure 10. Maximum weighted clique: execution time on the PCCs (POPC and PF).

Table II. Maximum weighted clique: speedups on the PCCs (POPC and PF) for $n = 1$ million.

p ($n = 1$ million)	Speedup POPC	p ($n = 1$ million)	Speedup PF
4	2.94	4	2.85
8	5.26	12	6.98

Table III. Maximum weighted clique: speedups on the PCCs (POPC and PF) for $p = 4$ and n varying.

n ($p = 4$)	Speedup POPC	Speedup PF
500 000	2.65	2.65
750 000	2.86	2.86
1 000 000	2.94	2.85

sort. It seems that this comes from the fact that we used the quicksort as sequential sort and therefore the gain between the quicksort and the parallel sort is larger than the gain between the counting sort and the parallel sort. Moreover, the speedup drops away from the theoretical speedup when we use more processors. These results come from communication overhead, and not from local computations. If the time for local computations is decreased by a factor p compared to the sequential time when p processors are used, it is not the case with communications. Indeed, when we use p processors for the execution of the algorithm on one input, and then p' on the same input size with $p < p'$ (then

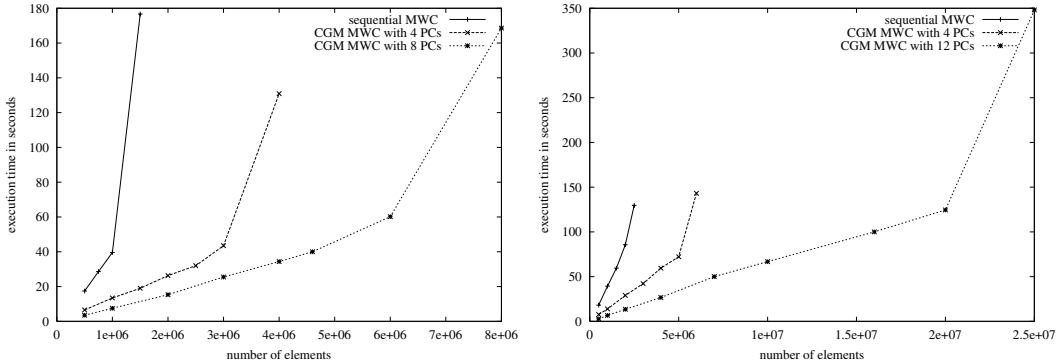


Figure 11. Maximum weighted clique: swap effects on the PCCs (POPC and PF).

there are less data per processor with p' than with p) the communication time is not decreased by a factor p/p', because even if each of the p' processors communicates p/p' less data than with the p processors, globally the same number of elements are communicated on the network. Note also that, as shown in Table III, the speedups are a little larger with POPC than with PF, especially for large values of n. This can be explained by routing conflicts that may be more frequent in the Fast Ethernet network than in the Myrinet network.

Swap effects. Figure 11 shows it is possible to handle very large data with this algorithm. On PF, one PC begins to swap with 2 million intervals sequentially, whereas four PCs can solve the problem on 5 million intervals before the swap limit and 12 PCs on 20 million intervals. POPC has the same behavior but begins to swap on data of half the size due to the PCs' memory size. If the sequential program runs on n intervals before the swap limit, the CGM program is able to reasonably solve the problem on a little less than $(p/2)n$ data. The factor $p/2$ comes from the fact that the algorithm needs not only the array of intervals as input, but also a buffer of the same size to communicate data.

We may note that for the counting sort one PC begins to swap for 7 million elements on PF, whereas for the maximum weighted clique problem it begins to swap before 2 million intervals. This is consistent because the counting sort does not sort in place and requires a memory space of size $2n$ to sort n data, whereas the quicksort is done in place. Since we sort $2n$ endpoints when we have n intervals we need $6n$ extra memory space to run Algorithm 3. This corresponds to a memory space of size $8n$ to solve the problem on n intervals. It is an algorithm greedy in memory because it does not work directly on intervals.

Communications–local computations. Figure 12 compares the communication times and the local computation times for four and eight PCs of POPC and for four and 12 PCs of PF. The local computations are predominant compared with communications. We note that we have only one large communication during the sort (exactly $4n$ data are communicated if we have n intervals: the endpoints and the associated weights), whereas there are essentially $(8(n/p) + 2(n/p)\log_2(2(n/p)))$ local operations. This behavior is the inverse of the sort where communications are predominant.

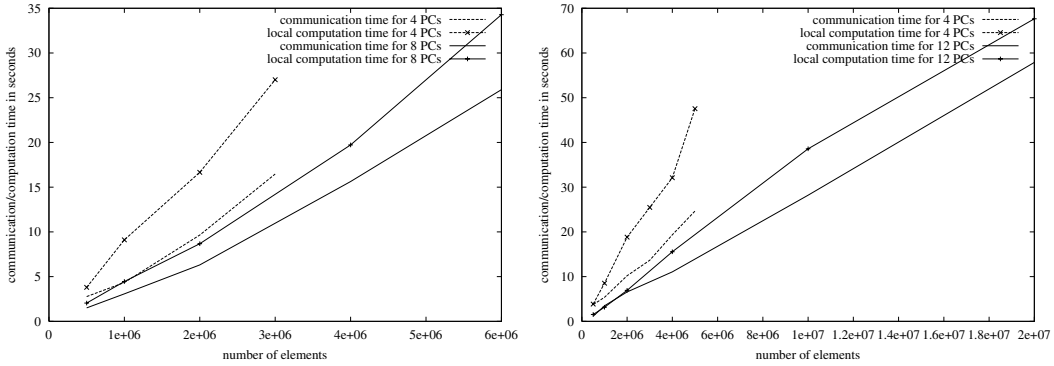


Figure 12. Maximum weighted clique: communications–local computations on the PCCs (POPC and PF).

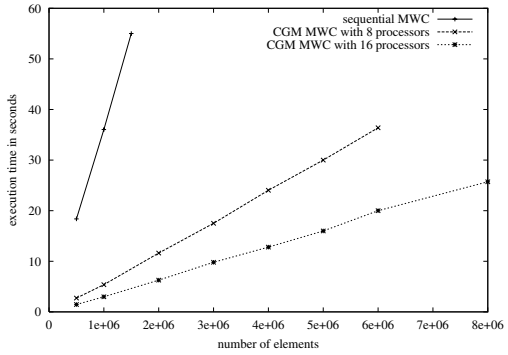


Figure 13. Maximum weighted clique: execution time on the T3E.

Moreover, this gap decreases when we use more processors, most probably due to the communications, as explained in the paragraph **Influence of p**.

Note that the results are similar between the two PCCs. Moreover, the PCs begin to swap before the interconnection networks are saturated.

5.2.2. T3E

We present the outcomes on the T3E. The code is the same as the one used on the PCCs.

Influence of p. Figure 13 shows the execution times on one, eight and 16 processors of the T3E. We will give the results with 32 processors in the paragraph **Handling of large data** because the curve

Table IV. Maximum weighted clique: speedups on the T3E for a problem with 1 million intervals.

p ($n = 1$ million)	Speedup
8	6.7
16	12.05

Table V. Maximum weighted clique: speedups on the T3E with eight processors and n varying.

n ($p = 8$ million)	Speedup
500 000	6.73
750 000	6.7
1 000 000	6.8

begins with 4 million elements and we cannot draw a comparison with the sequential. n ranges from 500 000 to 8 million. All the curves begin for $n = 500\,000$ because the constraints of the chosen sort algorithms are respected. Each curve stops before the memory of the processors becomes saturated.

The curves have the expected features and they are almost linear in n as for the PCCs. Moreover, the CGM program decreases the execution times, as expected.

Table IV gives the obtained speedup according to the number of processors used for a fixed problem size (1 million intervals). Table V gives the speedups obtained on eight processors for varying input size. As for the PCCs, these speedups are good. They are also better than those given for the parallel sort. For the same reasons as for the PCCs, it should come from the use of the quicksort. They are also better than those of the PCCs for large data. We think it is due to the fact that the communication of large data is faster on the T3E than on the PCCs, since local computations are more or less alike (see the paragraph **Communications–local computations**).

Handling of large data. Figure 14 shows it is possible to compute the maximum weighted clique with a lot of intervals using several processors. The curve with 32 processors begins with 4 million intervals, which implies a sort of 8 million data as required by the constraints of the parallel sort algorithm. In sequential, the processor's memory of the T3E saturates with a little more than 1.5 million intervals. Before memory saturation, eight processors can handle 6 million intervals, whereas 16 processors can handle 8 million intervals and 32 processors can handle 16 million intervals. If the problem can be solved sequentially on n intervals before memory saturation, then the CGM program can solve the problem in a little less than $(p/2)n$ intervals, as for the PCCs.

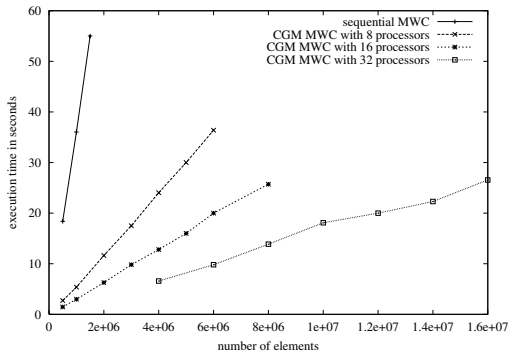


Figure 14. Maximum weighted clique: handling of large data on the T3E.

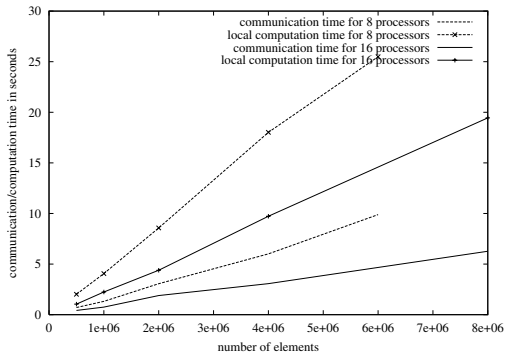


Figure 15. Maximum weighted clique: communications–local computations on the T3E.

Note that we can solve this problem on 16 million intervals in less than 30 s with 32 processors.

Communications–local computations. Figure 15 compares the communication times and the local computation times for eight and 16 processors of the T3E. As for the PCCs and for the same reasons, local computations are predominant and this gap decreases when more processors are used. This gap is larger on the T3E than on the PCCs because the communication of large data is faster on the T3E whereas the local computations are almost the same.

5.2.3. Conclusion on the maximum weighted clique problem

According to the outcomes, we can say that Algorithm 3 is portable and efficient. The execution times are always shorter when more processors are used. Moreover, it is possible to compute a maximum

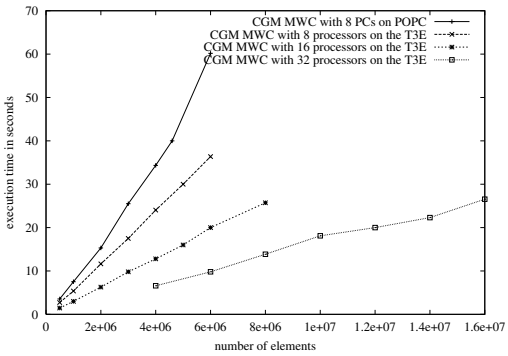


Figure 16. Maximum weighted clique: comparison between POPC and the T3E.

weighted clique on a great number of intervals, and the scalable aspect of our algorithm should allow the processing of larger data sets with more processors.

We think that this implementation is efficient because we have an efficient sort. When the endpoints of intervals are sorted, the other operations carried out on the intervals, like the prefix maximum, are simple and efficient. We observed exactly the same kind of results with the two other implemented algorithms (e.g. connected components and BFS).

To conclude, Figure 16 compares execution times on eight PCs of POPC and on eight, 16 and 32 processors of the T3E. The time on the eight processors of the T3E is shorter than on the eight PCs. It comes from the communication time which is less on the T3E than on POPC as previously explained. Nevertheless, the gap between these two curves is small compared with the performances of the two machines (the speed of the interconnection network of the T3E, like one of the processors, is larger than on POPC) and compared with their respective cost. Even if this comparison has to be drawn carefully because the C++ compilers are different between the T3E and POPC, such clusters of PCs with few processors can lead to efficient parallel executions for a relatively cheap price.

6. CONCLUSION

In this paper we have shown how to solve many important problems on interval graphs using a coarse-grained parallel computer such as a cluster of PCs. The proposed algorithms were shown to be theoretically efficient, straightforward to implement, portable and fast in practice on different parallel platforms. We believe this can largely be attributed to the use of the CGM model which accounts for distributed memory effects, mixes sequential and parallel coding, and encourages the use of a constant or very small number of communication rounds.

Note that the use of the CGM model, which was primarily developed for algorithm design in the context of interconnection networks, has led to efficient implementations even in the context of a bus-based network like Ethernet. We speculate that this is due to several factors including: (i) the

model focuses on sending a small number of large messages rather than a large number of small ones; (ii) it relies on standard, and typically well optimized, communications operations; and (iii) it focuses on reducing the number of communication rounds and therefore the number of interdependencies between rounds. Evidently, at some point such bus-based networks always become saturated and more attention must be paid to bandwidth and broadcast conflict concerns, particularly as one scales up. We are currently exploring how such concerns can best be dealt with within the context of a CGM-like model.

An interesting way forward for further research would be to compare larger PCCs (with up to 50 or 100 PCs for instance) with large parallel computers, like the Cray T3E.

ACKNOWLEDGEMENTS

We are grateful to the anonymous referees for their thorough reading of the manuscript and very helpful comments.

REFERENCES

1. Olariu S. Parallel graph algorithms. *Handbook of Parallel and Distributed Computing*, Zomaya A (ed.). McGraw-Hill: New York, 1996; 355–403.
2. Cole R, Vishkin U. The accelerated centroid decomposition technique for optimal tree evaluation in logarithmic time. *Algorithmica* 1988; **3**:329–346.
3. Moitra A, Johnson R. PT-Optimal algorithms for interval graphs. *Proceedings of the 26th Annual Allerton Conference Communication, Control and Computing*, University of Illinois, Monticello, USA, vol. 1, 1998; 274–282.
4. Das SK, Chen CC-Y. Efficient parallel algorithms on interval graphs. *Proceedings of the 4th International PARLE Conference*. Springer: Heidelberg, 1992; 131–143.
5. Kim SK. Optimal parallel algorithms on sorted intervals. *Proceedings of the 27th Annual Allerton Conference Communication, Control and Computing*, University of Illinois, Monticello, USA, vol. 1, 1990; 766–775.
6. Olariu S, Schwing JL, Zhang J. Optimal parallel algorithms for problems modelled by a family of intervals. *IEEE Transactions on Parallel and Distributed Systems* 1992; **3**(3):364–374.
7. Bertossi AA, Bonuccelli MA. Some parallel algorithms on interval graphs. *Discrete Applied Mathematics* 1987; **16**:101–111.
8. Gupta UI, Lee DT, Leung JY-T. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers* 1979; **C-28**:807–810.
9. Carrano AV. Establishing the order of human chromosome-specific DNA fragments. *Biotechnology and the Human Genome*. Plenum Press: New York, 1988.
10. Jungck JR, Streif V. Benzer: Deletion mapping of genetic 'fine structure'. http://www.beloit.edu/ biology/benzer.html [1997]. Software for teaching and learning biology.
11. Karf RM. Mapping the genome: Some combinatorial problems arising in molecular biology. *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*. ACM Press: New York, 1993; 278–285.
12. Culler DE, Karp RM, Patterson DA, Sahay A, Shacuser KE, Santos E, Subramonian R, von Eicken T. LogP: Towards a realistic model of parallel computation. *Proceedings of the 4th ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. ACM Press: New York, 1993; 1–12.
13. Dehne F, Fabri A, Rau-Chaplin A. Scalable parallel geometric algorithms for coarse grained multicomputers. *Proceedings of the 9th ACM Symposium on Computational Geometry*. ACM Press: New York, 1993; 298–307.
14. Hambrusch S, Khokhar A. C^3: An architecture-independent model for coarse-grained parallel machines. *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, USA, October 1994. IEEE Computer Society Press: Los Alamitos, 1994.
15. Valiant LG. A bridging model for parallel computation. *Communications of the ACM* 1990; **33**:103–111.
16. Caceres E, Chan A, Dehne F, Prencipe G. Coarse grained parallel algorithms for detecting convex bipartite graphs. *Proceedings of the 26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000)* (*Lecture Notes in Computer Science*, vol. 1928). Springer: Berlin, 2000; 83–94.

17. Dehne F, Deng X, Dymond P, Fabri A, Khokhar A. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, ACM Press: New York, 1995; 27–33.
18. Dehne F, Eavis T, Rau-Chaplin A. Coarse grained parallel on-line analytical processing (OLAP) for data mining. *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)* (*Lecture Notes in Computer Science*, vol. 2074). Springer: Berlin, 2001; 589–598.
19. Dehne F, Fabri A, Kenyon C. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. *Proceedings 6th IEEE SPDP*. IEEE Press: San Antonio, CA, 1994; 586–593.
20. Deng X, Gu N. Good algorithm design style for multiprocessors. *Proceedings 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, October 1994. IEEE Computer Society Press: Los Alamitos, 1994; 538–543.
21. Ferreira A, Rau-Chaplin A, Ubéda S. Scalable 2d convex hull and triangulation for coarse grained multicomputers. *Proceedings 6th IEEE Symposium on Parallel and Distributed Processing*, October 1995. IEEE Press: San Antonio, CA, 1995; 561–569.
22. Ferreira A, Schabanel N. A randomized BSP/CGM algorithm for the maximal independent set. *Parallel Processing Letters* 2000; **9**(3):411–422.
23. Gebremedhin A, Guérin Lassous I, Gustedt J, Telle JA. Graph coloring for a coarse grained multiprocessor. *Proceedings of the Workshop on Graph-Theoretical Concepts (WG 2000)* (*Lecture Notes in Computer Science*, vol. 1928), Brandes U, Wagner D (eds.). Springer: Berlin, 2000; 184–195.
24. Goodrich MT. Communication-efficient parallel sorting. *Proceedings of the 28th Symposium on Theory of Computing*. ACM Press: New York, 1996.
25. Guérin Lassous I, Gustedt J. Portable list ranking: An experimental study. *Proceedings of the Workshop on Algorithm Engineering (WAE 2000)*, September 2000 (*Lecture Notes in Computer Science*, vol. 1982). Springer: Berlin, 2000; 111–123.
26. Li H, Sevick K. Parallel sorting by overpartitioning. *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. ACM Press: New York, 1994; 46–56.
27. Caceres E, Dehne F, Ferreira A, Flocchini P, Rieping I, Roncato A, Santoro N, Song S. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Proceedings of ICALP'97* (*Lecture Notes in Computer Science*, vol. 1256). Springer: Berlin, 1997; 131–143.
28. Guérin Lassous I, Gustedt J, Morvan M. Feasibility, portability, predictability and efficiency: four ambitious goals for the design and implementation of parallel coarse grained graph algorithms. *Technical Report 3885*, INRIA 2000.
29. Ferreira A, Guérin Lassous I, Marcus K, Rau-Chaplin A. Parallel computations on interval graphs using pc clusters: Algorithms and experiments. *Proceedings of EuroPar'98* (*Lecture Notes in Computer Science*, vol. 1470), Pritchard D, Reeve J (eds.). Springer: Berlin, 1998; 875–886.
30. Cormen T, Leiserson CE, Rivest R. *Introduction to Algorithms*. MIT Press: Cambridge, MA, 1990.
31. Gerbessiotis AV, Valiant LG. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing* 1994; **22**:251–267.
32. Dekel E, Sahni S. Parallel scheduling algorithms. *Operations Research* 1983; **31**(1):24–39.
33. Golumbic MC. *Algorithmic Graph Theory and Perfect Graphs*. Academic: New York, 1980.
34. MATRA Systèmes & Information LIP. Pile of PC—POPC. http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html.
35. INRIA Sophia-Antipolis. Pc cluster. http://www.inria.fr/sophia/parallel.
36. T3E IDRIS. Cray t3e. http://www.idris.fr.
37. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderman V. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press: Cambridge, 1994.

