

A Load Balancing Technique for Some Coarse-Grained Multicomputer Algorithms

Thierry Garcia and David Semé
LaRIA
Université de Picardie Jules Verne,
CURI, 5, rue du Moulin Neuf
80000 Amiens, France,
E-mail: seme@laria.u-picardie.fr

Abstract

The paper presents a load balancing method for some CGM (Coarse-Grained Multicomputer) algorithms. This method can be applied on different dynamic programming problems such as: Longest Increasing Subsequence, Longest Common Subsequence, Longest Repeated Suffix Ending at each point in a word and Detection of Repetitions. We present also experimental results showing that our method is efficient.

1 Introduction

In parallel computational theory, one of major goals is to find a parallel algorithm which runs as fast as possible. For example, many problems are known to have efficient parallel algorithms which run in $\Theta(1)$ or $\Theta(\log n)$ computational time, where n is the input size of problems. From the point of view of the complexity theory, the class NC is used to denote the measure. A problem is in the class NC if there exists a parallel algorithm which solves the problem in $O(T(n))$ time using $O(P(n))$ processors, where $T(n)$ and $P(n)$ are polylogarithmic and polynomial functions for n , respectively. Many problems in the class P , which is the class of problems solvable in polynomial time sequentially, are also in the class NC . On the other hand, a number of problems in the class P seem to have no parallel algorithm which runs in polylogarithmic time using polynomial number of processors. Such problems are called P -complete. A problem is P -complete if the problem is in the class P and we can reduce any problem in the class P to the problem using NC -reduction. It is believed that problems in the class NC admit parallelization readily, and conversely, P -complete problems are inherently sequential and difficult to parallelize.

In this paper, we consider parallel algorithm for the Longest Increasing Subsequence (LIS) problem, the Longest Com-

mon Subsequence (LCS) problem, the Longest Repeated Suffix Ending at each point in a word (LRSE) problem and the Detection of Repetitions (DR) problem. Although these problems are primitive combinatorial optimisation problems, these are not known to be in the class NC or P -complete, that is, no NC algorithm have been proposed for this problem, and there is no proof which shows the problem is P -complete.

This paper describes a load balancing method for some Coarse-Grained Multicomputer algorithms which solve the Longest Increasing Subsequence problem, the Longest Common Subsequence problem, the Longest Repeated Suffix ending at each point in a word problem and the detection of Repetitions problem. These algorithms have been developed and implemented in [3, 4, 6, 5].

In these algorithms, there are P processors which work with $\frac{N}{P}$ data items. Therefore, the workload of processors is not optimal. For example, in [5], the first processor works one time, the second processor works two times and so on. Our goal is to give a method that leads to a good load balancing for each processor.

Our method consists of a better distribution of communication rounds and not on message sizes to assign workload to the processor. Then, this better distribution leads to a reduction of the number of processors.

In recent years several efforts have been made to define models of parallel computation that are more realistic than the classical PRAM models. In contrast of the PRAM, these new models are coarse grained, i.e. they assume that the number of processors P and the size of the input N of an algorithm are orders of magnitudes apart, $P \ll N$. By the precedent assumption these models map much better on existing architectures where in general the number of processors is at most some thousands and the size of the data that are to be handled goes into millions and billions.

This branch of research got its kick-off with Valiant [8] introducing the so-called Bulk Synchronous Parallel (BSP) machine, and was defined again in different directions for example LogP by Culler et al. [1], and CGM by Dehne et al. [2].

CGM seems to be the best suited for a design of algorithms that are not too dependent on an individual architecture.

We summarise the assumptions of this model :

- all algorithms perform so-called supersteps, that consist of one phase of interprocessor communication (or communication round) and one phase of local computation,
- all processors have the same size $M=O(\frac{N}{P})$ of memory ($M > P$),
- the communication network between the processors can be arbitrary.

The goal when designing an algorithm in this model is to keep the individual workload, time for communication and idle time of each processor within $\frac{T}{s(P)}$, where T is the runtime of the best sequential algorithm on the same data and $s(P)$, the speedup, is a function that should be as close to

P as possible. To be able to do so, it is considered as a good idea the fact of keeping the number of supersteps of such an algorithm as low as possible, preferably $O(M)$.

As a legacy from the PRAM model it is usually assumed that the number of supersteps should be polylogarithmic in P , but there seems to be no real world rationale for that. In fact, algorithms that simply ensure a number of supersteps that are a function of P (and not of N) perform quite well in practice, see Goudreau and al. [7].

The paper is organised as follows. In the Section 2, we present some problems for which we can apply our load balancing method presented in section 3. Section 4 presents some experimental results and the conclusion ends the paper.

2 Description of the problems

2.1 The LIS problem

Given a sequence A of N distinct integers, a subsequence of A is a sequence L which can be obtained from A in deleting zero or some integers (not necessarily consecutive). A sequence is increasing if each integer of this sequence is larger than the previous integer. Given a sequence $A = \{x_1, x_2, \dots, x_N\}$ of N distinct integers, we define an increasing subsequence or upsequence of length l as a upsequence of $A : \{x_{i_1}, x_{i_2}, \dots, x_{i_l}\}$ with $\forall j, k : 1 \leq j < k \leq l \Rightarrow i_j < i_k$ and $x_{i_j} < x_{i_k}$. A longest or maximal increasing subsequence is one of maximal length. Note that a maximal upsequence is not necessarily unique.

2.2 The LRSE problem

A *string* or *word* is a sequence of zero or more symbols from an alphabet Σ ; the string with zero symbols is denoted by ε . The set of all strings over the alphabet Σ is denoted by Σ^* . A string A of length N ($|A| = N$) is represented by $A[0] \dots A[N-1]$, where $A[i] \in \Sigma$ for $0 \leq i \leq N-1$. A string W is a *substring* (or *sub-word*) of A if $A = UWV$ for $U, V \in \Sigma^*$; we equivalently say that the string W occurs at position $|U| + 1$ in the string A . The position $|U| + 1$ is said to be the *starting position* of W in A and the position $|U| + |W|$ the *ending position* of W in A . We denote $W = A[|U| + 1 \dots |U| + |W|]$. A string W is a *prefix* of A if $A = WU$ for $U \in \Sigma^*$. Similarly, W is a *suffix* of A if $A = UW$ for $U \in \Sigma^*$. The LRSE problem consists of finding the Longest Repeated Suffix Ending at each point in a word.

2.3 The LCS problem

Given two input strings, the Longest Common Subsequence (LCS) problem consists in finding a subsequence of both strings which is of maximal possible length, say

P . Computing P only solves the problem of determining the length of an LCS (LLCS problem). Given a string A over an alphabet Σ (any finite set of symbols), a *subsequence* of A is any string C that can be obtained from A by deleting zero or some symbols (not necessarily consecutive). The *Longest Common Subsequence* (LCS) problem for two input strings $A = A_1A_2 \dots A_N$ and $B = B_1B_2 \dots B_M$ ($M \leq N$) consists in finding another string $C = C_1C_2 \dots C_P$ such that C is a subsequence of both A and B , and is a maximal possible length.

2.4 The DR problem

Let A be a finite alphabet and A^* be the free monoid generated by A . Let A^+ the free semigroup generated by A . A string $X \in A^+$ is fully specified by writing $X = x_1 \dots x_n$, where $x_i \in A$ ($1 \leq i \leq n$). A *factor* of X is a substring of X and its starting position in $\{1, 2, \dots, n\}$. The notation $X[k : l]$ is used to denote the factor of $X : x_k x_{k+1} \dots x_l$. A left (right) factor of X is a *prefix* (*suffix*) of X . A string $X \in A^+$ is *primitive* if setting $X = u^k$ implies $u = X$ and $k = 1$. A *square* in a string X is any nonempty substring of X in the form uu . String X is *square-free* if no substring of X is a square. Equivalently, X is square-free if each substring of X is primitive.

A *repetition* in X is a factor $X[k : m]$ for which there are indices l, d ($k < d \leq l \leq m$) such that :

1. $X[k : l]$ is equivalent to $X[d : m]$,
2. $X[k : d - 1]$ corresponds to a primitive word,
3. $X_{l+1} \neq X_{m+1}$.

p is a period of a repetition $X[k : m]$ of X if $x_i = x_{i+p}$ ($\forall i = k, k + 1, \dots, |X[k : m]| - p$). Therefore, $1 \leq p \leq \frac{|X[k:m]|}{2}$.

3 Description of our load balancing method

In the following, we denote N the number of data items and P the number of processors. Our goal is to define a technique which allows to improve processor's workload. In fact, a CGM algorithm which works on P processors, be able to work on $\frac{P}{2}$ processors. Then, we denote lbP the number of processors really used (where lbP is equal to $\frac{P}{2}$).

In the three past years we were interested in some dynamic programming problems. Our goal was to show how to derive systolic solutions of these problems to CGM algorithms. In fact, these systolic solutions can be divided into two classes : unidirectional linear systolic and bidirectional linear systolic solutions. Since the proposed CGM algorithms (see [3, 4, 6, 5]) were work optimal and time efficient, the workload of each processor was very bad. Then, we had observed that we can improve the workload with a better distribution of the data items and with a reduction of the number of processors.

3.1 Preliminaries and assumptions

In this section, we give some assumptions that will be used in the following.

Proposition 1 *As our approach consists in an optimal distribution of the data items, local computations of all our CGM algorithms presented in our previous papers are never modified.*

Definition 1 *Each processor i has a physical identification number lbP_i and two sets of $\frac{N}{P}$ data items : the i -th set and the $(lbP + i)$ -th set.*

Definition 2 *For all $i < lbP (= \frac{P}{2})$, the processor i should receive all the messages meant for the virtual processor $lbP + i (= \frac{P}{2} + i)$.*

Proposition 2 *Definitions 1 and 2 imply that a processor i can receive messages in two cases, these meant for the processor i and these meant for the virtual processor $lbP (= \frac{P}{2} + i)$.*

Definition 3 *We call M_i and M_{lbP+i} the messages received by the processor i meant for the processor i and for the virtual processor $lbP + i$, respectively.*

Lemma 1 *A processor i must terminate the computation of the data item from message M_i before the computation of the data item from the message M_{lbP+i} .*

Proof: In regard of dynamic programming problems, it is obvious that the data item from message M_i can be processed before the data item from message M_{lbP+i} . \square

Definition 4 *All communication rounds must be deterministic.*

Proposition 3 *Communication rounds of CGM algorithms presented in our previous papers can be modified in order to respect our assumptions.*

3.2 Load Balancing CGM algorithms from unidirectional linear systolic solutions

CGM algorithms based on unidirectional linear systolic solutions (figure 1) have communication rounds using only one direction.

The Longest Increasing Subsequence (LIS) problem

An efficient CGM solution of the LIS problem was described in [3] from a unidirectional linear systolic solution. The left part of the figure 2 represents communication rounds used in the CGM solution presented in [3] for 4 processors. While the right part of the figure 2 is the communication rounds used in our load balancing method. We

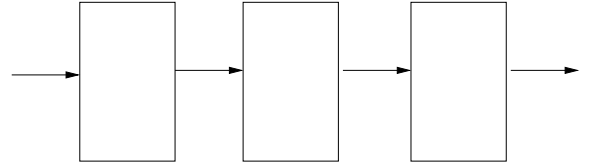


Figure 1: Unidirectional linear systolic array.

represent in black, processors making local computations after the communication round. We note that the number of rounds of the classical CGM solution using 4 processors is 4. And we have the same number of rounds for the load balancing CGM solution using 2 processors. When we observe all the rounds, the number of processors making local computations (black processors in the figure 2) is only 43% in the classical CGM solution (7 black processors) and 75% in the load balancing CGM solution (6 black processors). The global workload of the classical CGM solution is $\frac{P^2 - P + 2}{2P^2}$ and the global workload of the load balancing CGM solution is $\frac{3lbP^2 - lbP + 2}{4lbP^2}$. Thus, the global workload of the classical CGM solution is approximately equal to 50% and the global workload of the load balancing CGM solution is approximately equal to 75% for $P \geq 20$ and $lbP \geq 10$. Nevertheless, in the load balancing CGM solution, we observe that some processors have an excess workload. In fact these processors make twice the local computation. The number of processors with an excess workload is $\frac{lbP - 1}{4lbP}$ for $lbP \geq 2$. The global excess workload of the load balancing CGM solution is approximately equal to 25% for $lbP \geq 10$.

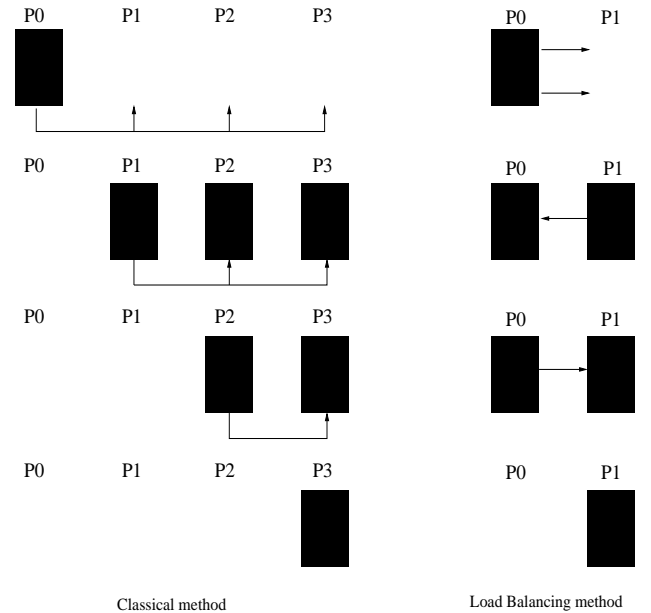


Figure 2: Communication round with normal and load balancing method for the LIS problem (for $P=4$ and $lbP=2$ respectively).

The Longest Repeated Suffix Ending at each point in a word (LRSE) problem

An efficient CGM solution of the LRSE problem was described in [5] from a unidirectional linear systolic solution. The left part of the figure 3 represents communication rounds used in the CGM solution presented in [5] for 4 processors. While the right part of the figure 3 represents the communication rounds used in our load balancing method. We note that the number of rounds of the classical CGM solution using 4 processors is 6. And we have the same number of rounds for the load balancing CGM solution using 2 processors. When we observe all the rounds, the number of processors making local computations (black processors in the figure 3) is only 35% in the classical CGM solution (7 black processors) and 71% in the load balancing CGM solution (6 black processors). The global workload of the classical CGM solution is $\frac{P+1}{4P-2}$ and the global workload of the load balancing CGM solution is $\frac{2P+1}{4P-1}$. Thus, the global workload of the classical CGM solution is approximately equal to 25% and the global workload of the load balancing CGM solution is approximately equal to 50% for $P \geq 20$ and $lbP \geq 10$. Note that no processor have an excess workload for the load balancing CGM solution of the LRSE problem.

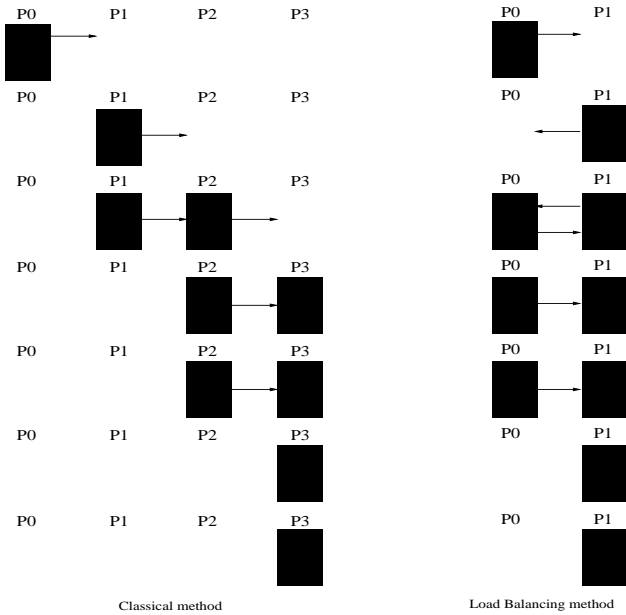


Figure 3: Communication round with normal and load balancing method for the LRSE problem (for $P=4$ and $lbP=2$ respectively).

3.3 CGM solutions from bidirectional linear systolic solutions

CGM algorithms based on unidirectional linear systolic solutions (figure 4) have communication rounds using the two directions.

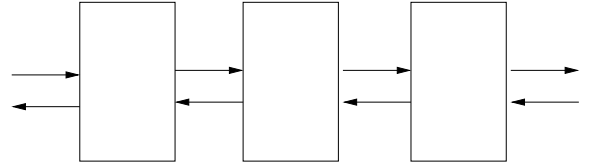


Figure 4: CGM algorithms based on bidirectional linear systolic solutions communications

The Longest Common Subsequence (LCS) problem

An efficient CGM solution of the LCS problem was described in [4] from a unidirectional linear systolic solution. The left part of the figure 5 represents communication rounds used in the CGM solution presented in [4] for 4 processors. While the right part of the figure 5 is the communication rounds used in our load balancing method. We note that the number of rounds of the classical CGM solution using 4 processors is 7. And we have the same number of rounds for the load balancing CGM solution using 2 processors. When we observe all the rounds, the number of processors making local computations (black processors in the figure 5) is only 57% in the classical CGM solution (16 black processors) and 85% in the load balancing CGM solution (12 black processors). The global workload of the classical CGM solution is $\frac{P^2}{2P^2-P}$ and the global workload of the load balancing CGM solution is $\frac{3lbP^2}{4lbP^2-lbP}$. Thus, the global workload of the classical CGM solution is approximately equal to 50% and the global workload of the load balancing CGM solution is approximately equal to 75% for $P \geq 20$ and $lbP \geq 10$. Nevertheless, in the load balancing CGM solution, we observe that some processors have an excess workload. In fact these processors make twice the local computation. The number of processors with an excess workload is $\frac{lbP^2}{4lbP^2-lbP}$ for $lbP \geq 2$. The global excess workload of the load balancing CGM solution is approximately equal to 25% for $lbP \geq 10$.

The Detection of Repetitions (DR) problem

An efficient CGM solution of the DR problem was described in [6] from a unidirectional linear systolic solution. The left part of the figure 6 represents communication rounds used in the CGM solution presented in [6] for 4 processors. While the right part of the figure 6 is the communication rounds used in our load balancing method. We note that the number of rounds of the classical CGM solution using 4 processors is 7. And we have the same number of rounds for the load balancing CGM solution using 2 processors. When we observe all the rounds, the number of processors making local computations (black processors in the figure 5) is only 32% in the classical CGM solution (9 black processors) and 64% in the load balancing CGM

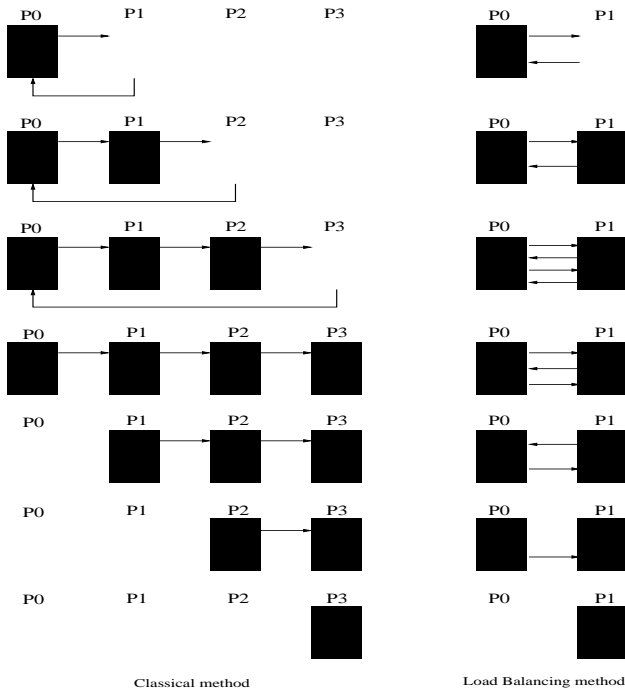


Figure 5: Communication round with normal and load balancing method for the LCS problem (for $P=4$ and $lbP=2$ respectively).

solution (9 black processors). The global workload of the classical CGM solution is $\frac{P^2+P-2}{4P^2-2P}$ and the global workload of the load balancing CGM solution is $\frac{2lbP^2+lbP-1}{4lbP^2-lbP}$. Thus, the global workload of the classical CGM solution is approximately equal to 25% and the global workload of the load balancing CGM solution is approximately equal to 50% for $P \geq 20$ and $lbP \geq 10$. Note that no processor have an excess load for the load balancing CGM solution of the DR problem.

4 Experimental Results

We have implemented the LIS, LCS, LRSE and DR programs in C language using MPI communication library and tested them on a multiprocessor Celeron 466Mhz platform running LINUX. The communication between processors was performed through an Full Duplex 100Mb Ethernet switch. Table 1 to Table 4 present total running times for the normal CGM algorithms and the load balancing CGM algorithms solving the LIS, LCS, LRSE and DR problems (for $P \in \{4, 8, 16\}$ and $P \in \{2, 4, 8\}$). Here, we have $N = 2^k$ and k is an integer such that $12 \leq k \leq 18$. The experimental results are conform with the theoretical results presented in section 3.2. In fact, results of the tables 3 and 4 are nearly the same. This is due to the fact that there exists no excess load on the processors for the LRSE and DR problems as described

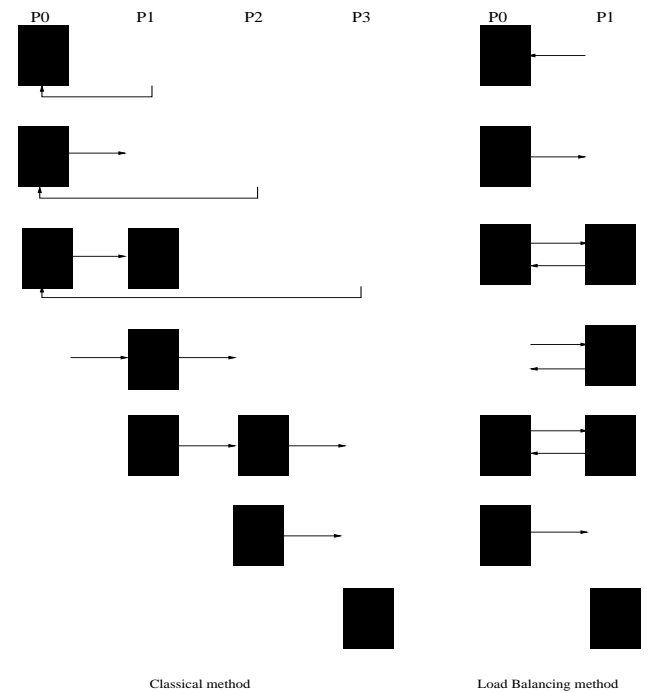


Figure 6: Communication round with normal and load balancing method for the DR problem (for $P=4$ and $lbP=2$ respectively).

in section 3.2. In contrary, results of the tables 1 and 2 are not identical. This comes from the excess workload on processors for the LIS and LCS problems. The excess workload for the LIS problem is about 33% in mean that is not very distant to the theoretical predictions of 25%. For the LCS problem, we have an excess workload of 15% in mean instead of 25% as declared in section 3.2.

N	P=2	P=4	P=8
16384	7.47	4.49	2.51
32768	30.45	17.97	9.76
65536	143.33	76.74	38.92
131072	872.37	343.80	158.63
262144	3584.23	1997.78	731.97
N	P=4	P=8	P=16
16384	5.89	3.34	1.91
32768	23.95	13.47	7.51
65536	111.26	54.08	30.44
131072	508.53	248.92	122.33
262144	2104.24	1123.26	551.10

Table 1: Execution time for the LIS problem (in seconds) with the normal method (for $P = 2, 4$ and 8) and the load balancing method (for $P = 4, 8$ and 16).

N	P=2	P=4	P=8
16384	54.05	47.55	43.05
32768	237.58	191.30	171.49
65536	1025.78	839.00	694.88
131072	4633.99	3583.12	3056.11
262144	18515.89	16820.35	13135.14

N	P=4	P=8	P=16
16384	49.82	41.26	37.77
32768	222.66	169.36	150.54
65536	947.61	726.58	613.33
131072	3768.51	3146.09	2582.57
262144	15082.07	12495.77	11431.88

Table 2: Execution time for the LCS problem (in seconds) with the normal method (for P = 2, 4 and 8) and the load balancing method (for P = 4, 8 and 16).

N	P=2	P=4	P=8
16384	15.95	9.82	5.40
32768	68.73	39.23	21.52
65536	302.48	166.76	85.87
131072	1474.76	748.90	371.06
262144	5988.63	4997.75	1640.80

N	P=4	P=8	P=16
16384	15.89	9.84	6.09
32768	70.95	39.21	24.08
65536	301.64	168.24	96.26
131072	1712.07	742.19	402.70
262144	6970.02	5527.03	1797.66

Table 3: Execution time for the LRSE problem (in seconds) with the normal method (for P = 2, 4 and 8) and the load balancing method (for P = 4, 8 and 16).

N	P=2	P=4	P=8
16384	36.98	21.53	11.56
32768	164.71	86.18	46.15
65536	663.10	444.67	184.57
131072	2975.06	1785.86	1029.63
262144	11983.66	9191.08	4155.06

N	P=4	P=8	P=16
16384	36.95	23.89	12.46
32768	147.89	93.63	49.45
65536	596.20	457.53	198.08
131072	3250.28	1845.57	1012.15
262144	12022.14	9747.33	4089.96

Table 4: Execution time for the DR problem (in seconds) with the normal method (for P = 2, 4 and 8) and the load balancing method (for P = 4, 8 and 16).

5 Concluding remarks

This paper presents a simple load balancing method for some CGM algorithms. We presented also experimental results showing that our method is efficient for these problems. This approach allows to increase the workload of each processor and to reduce the number of processors by a factor two.

References

- [1] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauer, E. Santos, R. Subramonian, and T. Von Eicken. Log”p”:towards a realistic model of parallel computation. *4-th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 1–12, 1996.
- [2] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.
- [3] T. Garcia, J.F. Myoupo, and D. Semé. A work-optimal cgm algorithm for the longest increasing subsequence problem. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’01)*, 2001.
- [4] T. Garcia, J.F. Myoupo, and D. Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. *11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP’03)*, 2003.
- [5] T. Garcia and D. Semé. A coarse-grained multicomputer algorithm for the longest repeated suffix ending at each point in a word. In *International Conference of Computational Science and Its Applications (ICCSA’03)*, volume 2, pages 239–248, 2003.
- [6] T. Garcia and D. Semé. A coarse-grained multicomputer algorithm for the detection of repetitions problem. *Information Processing Letters*, (93):307–313, 2005.
- [7] M. Goudreau, S. Rao K. Lang, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the bsp model. *8th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA’96)*, pages 1–12, 1996.
- [8] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.