A Parallel Chain Matrix Product Algorithm on the InteGrade Grid*

Edson Norberto Cáceres, Henrique Mongelli,
Leonardo Loureiro, Christiane Nishibe
Dept. de Computação e Estatística
Universidade Federal de Mato Grosso do Sul
Campo Grande - MS, Brazil
E-mails: {edson,mongelli}@dct.ufms.br,
{metalbr,cnishibe}@gmail.com

Siang Wun Song
Dept. of Computer Science
Universidade de São Paulo
São Paulo - SP, Brazil
E-mail: song@ime.usp.br

Abstract

The InteGrade middleware intends to exploit the idle time of computing resources in computer laboratories. In this work we investigate the performance of running parallel applications with communication among processors on the InteGrade grid. Since costly communication on a grid can be prohibitive, we explore the so-called systolic or wavefront paradigm to design the parallel algorithms in which no global communication is used. We consider the matrix chain product problem and design a parallel algorithm to evaluate the performance of the InteGrade middleware. We show that this application running under the InteGrade middleware and MPI takes slightly more time than the same application running on a cluster with only LAM-MPI support. The results can be considered promising and the time difference between the two is not substantial.

1. Introduction

A trend in parallel and distributed computer systems is the use of grid computing. With the sharing of existing computer resources, universities, private and public corporations can use grid computing to enhance their computing infrastructure.

The InteGrade project is an on-going multi-university research initiative with the objective of designing a grid computing middleware to exploit and utilize the idle computing power of existing resources in computer laboratories [7, 11]. The InteGrade middleware allows the use of existing computing infrastructure to run useful applications. At the same time, the middleware needs to ensure that the users of the shared computing resources do not have degraded quality of service. Transparency to the users and ease of utilization are the main goals of the InteGrade middleware. This middleware is responsible for job submission, checkpointing, security, job migration, etc. Many publications on InteGrade can be found on the InteGrade webpage [11].

The InteGrade project is being developed jointly by researchers of several institutions: Department of Computer Science of Universidade de São Paulo, Departments of Informatics of Pontifícia Universidade Católica (Rio de Janeiro) and Universidade Federal do Maranhão, and Department of Computing and Statistics of Universidade Federal de Mato Grosso do Sul.

With an object oriented architecture, InteGrade implements each module of the system that communicates with the other modules through remote method invocations. InteGrade uses CORBA [8] as its infrastructure of distributed objects, thus benefiting from an elegant and solid architecture. One important result is the ease of implementation, since the communication with the system modules is abstracted from the remote method invocations.

Many existing grid computing systems restrict their use to applications that can be decomposed into independent tasks such as *Bag-of-Tasks* [3]. InteGrade was designed with the objective of allowing the development of applications to solve a broad range of problems in parallel. In addition to handling bag-of-tasks type applications, InteGrade also deals with parallel applications with dependencies that require communication among processors. For the purpose of evaluating the InteGrade middleware under such conditions, we design a parallel algorithm to solve the chain matrix product problem.

*Partially supported by FAPESP Proc. No. 2004/08928-3 CNPq Proc. Nos. 55.0895/07-8, 30.5362/06-2, 30.2942/04-1, 62.0123/04-4, 48.5460/06-8, 62.0171/06-5 and FUNDECT 41/100.115/2006.

Experimental results are shown in this paper.

An important question we wish to address concerns the overhead of the grid middleware. On the one hand, a grid middleware ensures an integrated environment, to ease the concern of the user, with special modules to handle the job submission, checkpointing, security, task migration, etc., in contrast to running a parallel algorithm in a cluster without such a middleware. One the other hand, it is natural that overhead is incurred. We executed a parallel application both on a cluster using LAM-MPI and on the grid using InteGrade middleware and MPI. Our results show a slight performance degradation when the parallel applications are run on the InteGrade. The difference in performance with respect to running on a cluster is, however, small. This is encouraging and shows a small and acceptable overhead of the InteGrade middleware.

2 The Systolic Algorithm Paradigm with Low Communication Demand

In the early eighties, systolic arrays have been proposed to implement numerically intensive applications, e.g. image and signal processing operations such as the discrete Fourier transform, product of matrices, matrix inversion, etc. for VLSI implementation on silicon chips [12]. After many such algorithms have been proposed in an *ad hoc* manner, an important method was proposed to formalize the design of systolic algorithms. Given a sequential algorithm specified as nested loops, more formally as a system of uniform recurrence equations, dependence transformation methods [16, 17, 19] map the specified computation into a time-processor space domain that can be mapped onto a systolic array.

The systolic array paradigm has low communication demand because it does not use costly global communication and each processor communicates with few other processors. It is thus suitable for implementation on a cluster of computers in which we wish to avoid costly global communication operations. A recent work [9] explores the systolic array paradigm in cluster computing. This approach, however, is not adequate in a heterogeneous environment where the performance of the computers may vary along time. Since the systolic structure is based on tightly-coupled connections, the existence of one single slow processor can compromise and degrade the overall performance. The systolic approach, therefore, is vulnerable in a heterogeneous environment where machines perform differently.

In [18] we proposed a redundant systolic solution with high-availability to deal with this problem. There are many techniques for dependable computing based on check-pointing and roll-back recovery [20]. The redundant approach is simple but we introduce some overhead to coordinate the actions of the redundant processors. We show that this overhead is worth the performance improvement it provides. The experimental results show that the incurred overhead is small compared to the overall performance we get over the non-redundant solution. We analyzed the overhead that results from the need to coordinate the actions of the redundant processors and showed that this overhead is worth the performance improvement it provides.

3 The Coarse-Grained Multicomputer CGM model

One of the earliest models to consider communication costs and to abstract the characteristics of parallel machines with a few parameters is the *Bulk Synchronous Parallel Model (BSP)* [21]. It gives reasonable predictions on the performance of the algorithms implemented on existing, mainly distributed memory, parallel machines. A BSP algorithm consists of a sequence of super-steps separated by *synchronization barriers*. In a super-step, each processor executes a set of independent operations using local data available in each processor at the start of the super-step, as well as communication consisting of send and receive of messages. An *h-relation* in a super-step corresponds to sending or receiving at most h messages in each processor.

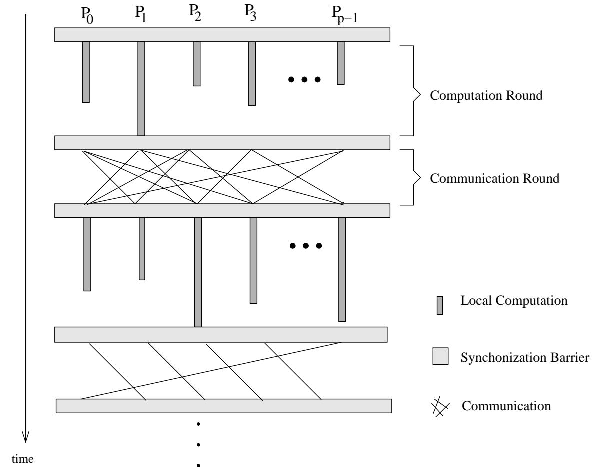


Figure 1. The Coarse-Grained Multicomputer.

Figure 1 shows a a similar and simpler model which is called the *Coarse Grained Multicomputers - CGM*,

proposed by Dehne *et al* [4, 5]. It uses only two parameters: the input size n and the number of processors p. On a CGM p processors are connected through any interconnection network. The term *coarse granularity* comes from the fact that the problem size in each processor n/p is considerably larger than the number of processors p.

A CGM algorithm consists of a sequence of rounds, alternating well defined local computing and global communication. Normally, during a computing round we use the best sequential algorithm for the processing of data available locally. A CGM algorithm is a special case of a BSP algorithm where all the communication operations of one super-step are done in the h-relation. In comparison with the BSP model, the CGM allows only bulk messages in order to minimize message overhead. Due to the similarity of the two models, we will use the term BSP/CGM.

More precisely, let n denote the input size of the problem. A BSP/CGM consists of a set of p processors each with local memory and each processor is connected by a router that can send messages in a point-to-point fashion. A BSP/CGM algorithm consists of alternating local computation and global communication rounds separated by a synchronization barrier.

In a computing round, we usually use the best sequential algorithm in each processor to process locally its data. In each communication round the total data exchanged by each processor (sends/receives) is limited by $O(n/p)$. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. In the BSP/CGM model, the communication cost is modeled by the number of communication rounds.

Finding an efficient algorithm on the BSP/CGM model is equivalent to minimizing the number of communication rounds as well as the total local computation time. The BSP/CGM model has the advantage of producing results which are closer to the actual performance on commercially available parallel machines. It is particularly suitable in current parallel machines in which the global computing speed is considerably larger than the global communication speed.

The CGM algorithms implemented on currently available multiprocessors present speedups similar to the speedups predicted in theory [4]. The CGM algorithm design goal is to minimize the number of supersteps and the amount of local computation.

4. Matrix Chain Problem

The matrix chain product problem is defined as follows. Consider the evaluation of the product of n matrices

$$M = M_1 \times M_2 \times \ldots \times M_n$$

where M_i is a matrix of dimensions $d_i \times d_{i+1}$.

Since matrix multiplication satisfies the associative law, the final result is the same for any order the matrices are multiplied. However, the order of multiplication affects the total number of operations to compute M. The problem is to find an optimal order of multiplying the matrices, such that the total number of operations is minimized [14, 15, 10].

The first polynomial time algorithm for the matrix chain product problem was proposed by Godbole [6]. The algorithm uses the Dynamic Programming technique and runs in $O(n^3)$ time with $O(n^2)$ space.

We give the main ideas of the Dynamic Programming approach to solve the matrix chain product problem. Details can be found in [1]. Dynamic Programming is a technique that computes the solution of a problem by first computing the solutions of the subproblems. The computation proceeds from smaller subproblems to larger subproblems, and the partial solutions of the subproblems are stored for future use so that they need not be recomputed again.

Let us give a simple example. Consider the matrix chain product of the following, say $n = 4$, matrices.

$$\underbrace{M}_{10\times100} = \underbrace{M_1}_{10\times20} \times \underbrace{M_2}_{20\times50} \times \underbrace{M_3}_{50\times1} \times \underbrace{M_4}_{1\times100}$$

Let the dimensions of M_1, M_2, M_3 and M_4 be 10×20, 20×50, 50×1 and 1×100, respectively. In other words, $d_1 = 10$, $d_2 = 20$, $d_3 = 50$, $d_4 = 1$ and $d_5 = 100$.

The trivial matrix product algorithm to multiply a matrix of dimension $a \times b$ by another of dimension $b \times c$ requires abc operations, giving rise to a $a \times c$ matrix.

If we compute the matrix chain product in the following way

$$M_1 \times (M_2 \times (M_3 \times M_4))$$

then we would use 125000 operations.

However, if we compute the same product as

$$(M_1 \times (M_2 \times M_3)) \times M_4$$

then we would require only 2200 operations.

The best way to compute the matrix chain product can be obtained by a Dynamic Programming method as follows.

We wish to compute the product of n matrices

$$M = M_1 \times M_2 \times \ldots \times M_n$$

where M_i is a matrix of dimensions $d_i \times d_{i+1}$.

Let $m_{i,j}$ be the minimum cost (in terms of number of operations) to compute

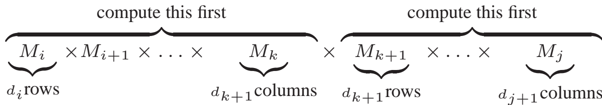
$$M_i \times M_{i+1} \times \ldots \times M_j$$

for $1 \leq i \leq j \leq n$.

We can thus formulate $m_{i,j}$ as follows.

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}(m_{i,k} + m_{k+1,j} + d_i d_{k+1} d_{j+1}) & \text{if } i < j \end{cases}$$

We can easily understand this formulation by considering the following product. The expression to be minimized above attempts to find the best point (k) to compute two partial products or sub-chains first, with the minimum number of operations.



The main idea is find the costs of multiplying all the sub-chains and combine them, first we compute all sub-chains of size 2 and save their costs, then we compute all sub-chain of size 3 using the costs already computed and so on [13, 2].

The goal is to minimize $m_{1,n}$, i.e. the cost to computer $M = M_1 \times M_2 \times \ldots \times M_n$. To this end, we first compute $m_{i,i}$ (difference of the two indices = 0). Obviously $m_{i,i} = 0$. Then we compute $m_{i,i+1}$ (difference of the two indices = 1), $m_{i,i+2}$ (difference of the two indices = 2), and so on. All such values computed are stored for further use to compute $m_{i,j}$ with larger difference between the indices.

In the example product, we wish to obtain m_{14}. So we compute the following values, row by row, with increasing difference between the indices.

$$\begin{array}{llll} m_{11} = 0 & m_{22} = 0 & m_{33} = 0 & m_{44} = 0 \\ m_{12} = 10000 & m_{23} = 1000 & m_{34} = 5000 & \\ m_{13} = 1200 & m_{24} = 3000 & & \\ m_{14} = 2200 & & & \end{array}$$

Algorithm 1 THE SEQUENTIAL MATRIX CHAIN PRODUCT ALGORITHM

Input: (1) Array $d[0..n]$ containing the dimensions $d_1, d_2, \ldots, d_n, d_{n+1}$ of the n matrices. $d.length$ is the size of array d.

Output: Array $m[i, j]$ that will contain the minimum cost to obtain the matrix chain product $M = M_1 \times M_2 \times \ldots \times M_n$.

1: $m(d.length - 1, d.length - 1)$; // matrix of costs
2: **for** $i \leftarrow 0$ **to** $d.length - 2$ **do**
3: $\quad m[i, i] \leftarrow 0$;
4: **end for**
5: **for** $round \leftarrow 1$ **to** $d.length - 2$ **do**
6: $\quad$ **for** $i \leftarrow 1$ **to** $d.length - 2 - round$ **do**
7: $\quad\quad j \leftarrow i + round$;
8: $\quad\quad m[i, j] \leftarrow \infty$;
9: $\quad\quad$ **for** $k \leftarrow 1$ **to** $j - 1$ **do**
10: $\quad\quad\quad aux \leftarrow m[i, k] + m[k + 1, j] + d[i] \times d[k + 1] \times d[j + 1]$;
11: $\quad\quad\quad$ **if** $aux < m[i, j]$ **then**
12: $\quad\quad\quad\quad m[i, j] \leftarrow aux$;
13: $\quad\quad\quad$ **end if**
14: $\quad\quad$ **end for**
15: $\quad$ **end for**
16: **end for**
17: return $m[0, d.length - 2]$;

The sequential matrix chain product algorithm is shown in Algorithm 1.

4.1. The parallel algorithm

In this section we present an $O(p)$ communication round and $O(n^3/p)$ complexity BSP/CGM algorithm for computing the solution of the matrix chain product problem with $n + 1$ dimensions (n matrices) and p processors. In the parallel algorithm, the cost array $m[i, j]$ will be divided into p parts of dimension $n/p \times n$. Each processor P_i will compute part i, $1 \leq i \leq p$. At the first parallel round, processor P_i will compute a block of dimension $n/p \times n/p$ and send it to processor P_{i-1}, at the next parallel round a new block will be computed using the block received and so on. At the first parallel round all processor will be working, at end of round r processor P_{p-r+1} will stop working. Figure 2 shows an example for $p = 4$, where P_i is the processor and R_i is the parallel round.

At the first round, all the p processors do useful work. Then at each round, one processor stops working and so on. The load is thus not balanced, which is a characteristic of many systolic algorithms. This is a drawback of the approach. What we gain is the modest communica-

Algorithm 2 THE PARALLEL MATRIX CHAIN PRODUCT ALGORITHM

Input: (1) Array d of dimensions (2) The number of processors p (3) The rank or id i of the processor.
Output: Array $m[i, j]$ containing the minimum cost to obtain the matrix chain product M.

1: $block \leftarrow (d.length - 1)/p$;
2: $m(block, d.length - 1)$; // matrix of costs
3: **for** $round \leftarrow 0$ **to** $p - i$ **do**
4: compute $m[0..block - 1, round * block..((round + 1) * block) - 1]$;
5: **if** $i \neq 1$ **then**
6: send computed block to P_{i-1};
7: **end if**
8: **if** $i \neq p$ **then**
9: receive block from P_{i+1};
10: **end if**
11: **end for**
12: **if** $i = p$ **then**
13: return $m[0, d.length - 2]$;
14: **end if**

tion demand, as mentioned earlier. Notice that data are transmitted in a wavefront or systolic manner, with each processor communication with a few other processors. The parallel matrix chain product algorithm is shown in Algorithm 2.

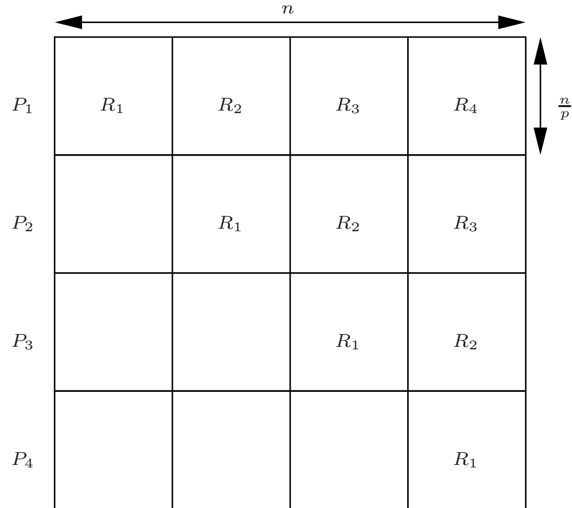


Figure 2. Parallel matrix chain product: division of tasks

n	1	2	4	8
256	0.180	0.165	0.159	-
512	2.756	2.449	2.045	-
1024	38.651	31.210	30.880	-
2048	325.776	276.156	272.819	-

Table 2. Running times for matrix chain product on the grid using InteGrade MPI

4.2. Experimental Results

n	1	2	4	8
256	0.158	0.135	0.095	0.078
512	2.464	2.205	1.704	1.174
1024	32.609	28.283	21.207	14.407
2048	259.346	227.029	175.939	117.089

Table 1. Running times for matrix chain product on the cluster using LAM-MPI

We have run the BSP/CGM matrix chain product algorithm on a cluster composed by 12 nodes consisting of 6 CPU Intel Pentium IV of 1.7Ghz and 6 CPU AMD Athlon of 1.6GHz. The nodes are connected by a 1Gb fast-Ethernet switch. The data used in the tests were generated randomly.

The matrix chain product parallel algorithm is implemented using standard ANSI C and, on the cluster we used LAM-MPI library while on the cluster used as an InteGrade grid we used the InteGrade middleware and MPI. The purpose of the experiment is to compare the two executions, on the cluster using LAM-MPI and on the grid using the InteGrade middleware and MPI.

Table 1 and Figure 3 show the running times (in seconds) for the matrix chain product parallel algorithm running on the cluster using LAM-MPI.

Table 2 and Fig. 4 show the running times (in seconds) for the matrix chain product parallel algorithm running on the grid using InteGrade middleware and MPI.

Tables 3 and 4 present a comparison between the running times on a cluster using standard LAM-MPI and on the grid running the InteGrade middleware and MPI. Column I and column II show the times on the cluster and on the grid, respectively. Figure 5 shows the corresponding curve.

We observe that the running time on the cluster using only LAM-MPI without the InteGrade middleware is slightly better than the times on the grid. Only in one case the times are the same. Notice that in the grid, the

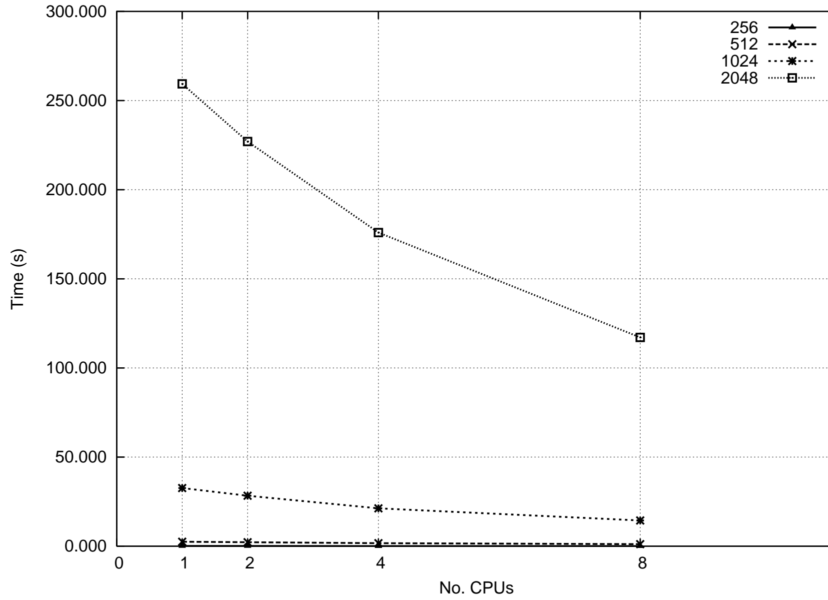


Figure 3. Running times for matrix chain product on the cluster using LAM-MPI

	256		512	
P	I	II	I	II
1	0.158	0.180	2.464	2.756
2	0.135	0.165	2.205	2.449
4	0.095	0.159	1.704	2.045
8	0.078	-	1.174	-

Table 3. Comparing running times for the matrix chain product algorithm

	1024		2048	
P	I	II	I	II
1	32.609	38.651	259.346	325.776
2	28.283	31.210	227.029	276.156
4	21.207	30.880	175.939	272.819
8	14.407	-	117.089	-

Table 4. Comparing running times for the matrix chain product algorithm

InteGrade middleware determines the choice of the machines.

The results can be considered promising and the time difference between the two is not substantial. This shows the overhead of the InteGrade middleware is acceptable, in view of the benefits obtained to facilitate the use of grid computing by the user.

5. Conclusions

The InteGrade project is an on-going research initiative that exploits the idle computing resources of existing hardware in computer laboratories. This paper intends to investigate the performance of running parallel applications with communication among processors under the InteGrade middleware.

Due to the high communication cost in cluster and grid computing, we are interested in designing parallel applications with low demand on communication. To this end, we revisit the systolic array approach and propose to design wavefront parallel algorithms with the nice property of each processing having to communicate with only a few others.

We presented a parallel systolic or wavefront algorithm for the matrix chain product problem to evaluate the performance of the InteGrade middleware. The application running under the InteGrade grid takes slightly more time than those running under the standard MPI in a cluster. The results are considered to be satisfactory, since the time difference is not substantial. This shows the overhead of the InteGrade middleware is acceptable, in contrast to the benefits obtained to ease the use of grid computing by the user.

As a final note, although the purpose of designing and using the proposed matrix chain product parallel algorithm is to evaluate the performance of the InteGrade grid middleware, the proposed algorithm is interesting

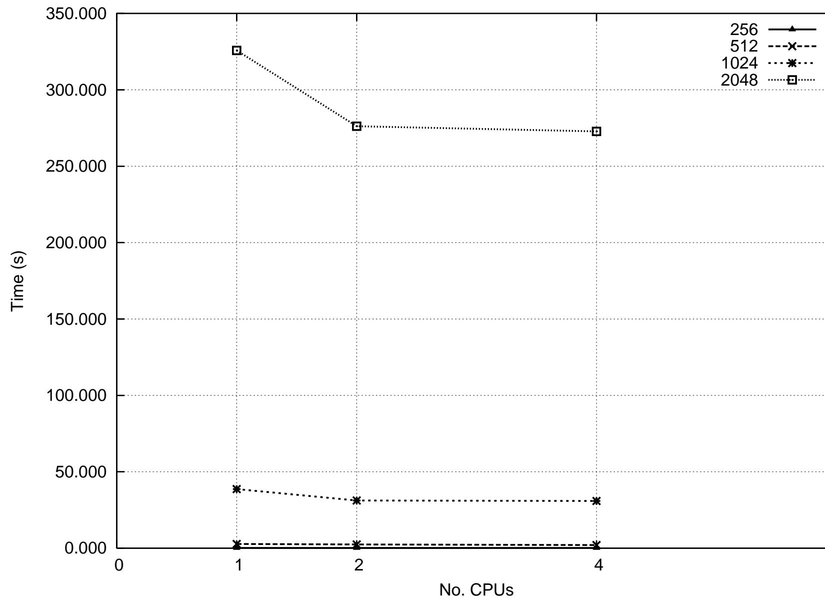


Figure 4. Running times for matrix chain product on the grid using InteGrade MPI

in its own right. This parallel algorithm can also be used to solve many other problems whose solutions are obtained by a similar Dynamic Programming technique. These includes the problem of finding the optimal binary search tree, the parenthesis matching problem, etc [1].

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms - Second Edition*. The MIT Press, 2001.

[2] A. Czumaj. Parallel algorithm for the matrix chain product and the optimal triangulation problems (extended abstract). In *STACS '93: Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 294–305, London, UK, 1993. Springer-Verlag.

[3] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. *Lecture Notes in Computer Science*, 2790:169–180, 2003.

[4] F. Dehne. Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999. Editorial Note.

[5] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[6] S. Godbole. On efficient computation of matrix chain products. *IEEE Trans. Computers*, 22(9):864–866, 1973.

[7] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[8] O. M. Group. *CORBA v3.0 Specification*. Needham, MA, July 2002. OMG Document 02-06-33.

[9] U. K. Hayashida, K. Okuda, J. Panneta, and S. W. Song. Generating parallel algorithms for cluster and grid computing. In *The 2005 International Conference on Computational Science - ICCS 2005*, volume 3514 of *Lecture Notes in Computer Science*, pages 509–516. Springer Verlag, 2005.

[10] T. C. Hu and M. T. Shing. Computation of matrix chain products: Part i, part ii. Technical report, Stanford, CA, USA, 1981.

[11] InteGrade. http://www.integrade.org.br/portal/, 2008.

[12] H. T. Kung. Why systolic architectures. *IEEE Transactions on Computers*, 15:37–46, 1982.

[13] H. Lee, J. Kim, S. Hong, and S. Lee. Parallelizing matrix chain products, 1997.

[14] H. Lee, J. Kim, S. J. Hong, and S. Lee. Processor allocation and task scheduling of matrix chain products on parallel systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):394–407, 2003.

[15] K. Li. Analysis of parallel algorithms for matrix chain product and matrix powers on distributed memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):865–878, 2007.

[16] D. I. Moldovan. *Parallel Processing: from Applications to Systems*. Morgan Kaufmann Publishers, 1993.

[17] K. Okuda. Cycle shrinking by dependence reduction. In *Proceedings 2nd International Euro-Par Conference*, volume 1123 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, 1996.

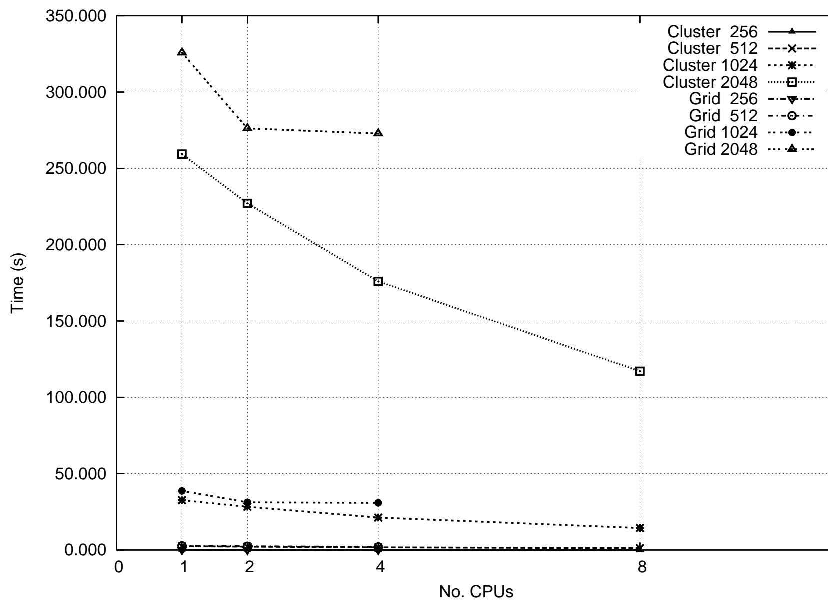


Figure 5. Compare running times for matrix chain product on the cluster and on the grid

[18] K. Okuda, S. W. Song, and M. T. Yamamoto. Reliable systolic computing through redanduncy. In *11th Asia-Pacific Computer Systems Architecture Conference (AC-SAC 2006)*, volume 4186 of *Lecture Notes in Computer Science*, pages 423–429. Springer Verlag, 2006.

[19] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques*. Masson, 1989.

[20] M. Treaster. A survey of fault-tolerant and fault-recovery techniques in parallel systems. *ArXiv Computer Science e-prints*, pages 1–11, January 2005.

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