A Parallel Adaptive version of the Block-based Gauss-Jordan Algorithm

N. Melab and E-G. Talbi and S. Petiton
Laboratoire d'Informatique Fondamentale de Lille (LIFL-CNRS URA 369)
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cedex -France-

{melab,talbi,petiton}@lifl.fr

Abstract

This paper presents a parallel adaptive version of the block-based Gauss-Jordan algorithm used in numerical analysis to invert matrices. This version includes a characterization of the workload of processors and a mechanism of its adaptive folding/unfolding. The application is implemented and experimented with MARS[1] *in dedicated and non-dedicated environments. The results show that an absolute efficiency of* 92% *is possible on a cluster of DEC/ALPHA processors interconnected by a Gigaswitch network and an absolute efficiency of* 67% *can be obtained on an Ethernet network of SUN-Sparc4 workstations. Moreover, the adaptability of the algorithm is experimeted on a non-dedicated meta-system including both the two parks of machines.*

Keywords: *Gauss-Jordan Algorithm, Adaptive Parallelism, Network of Workstations (NOW), Meta-systems.*

1. Introduction

Networks of workstations (NOWs) have two main characteristics: workstations are heterogeneous, meaning all have neither the same architecture nor the same operating system. Moreover, they are *adaptive environments* i.e. workstations frequently leave and join the network due to their availability/lack of availability or failure. These factors make that it is necessary to change the parallel programming methodology in order to take into account such adaptivity, we refer to that *parallel adaptive programming*. An application developed according to this philosophy is called a parallel adaptive application i.e. a parallel application whose processes vary in *number* and *location* as a function of the load (availability of processors, CPU time, etc) of the machine.

[1] *MARS* is a parallel adaptive programming environment.

In this paper, we present the *MARS* [2] programming methodology of adaptive applications through our parallel adaptive version of the block-based Gauss-Jordan algorithm. This is a direct method used in numerical analysis for inverting matrices. A classical parallel version of this application has already been developed in [5, 6]. In [5], the application is implemented on a MIMD machine which is constituted by 8 *PEs* (Processor Elements), 4 data migration controllers and 3 levels shared memory, interconnected by an OMEGA network. The particularity of that implementation is that it includes a real-time scheduler, which schedules the tasks of the application according to a tasks' graph determined by hand. In [6], the proposed method is implemented with the Occam language on a network of 8 transputers connected according to a cube topology.

Unlike these implementations, our parallel version is *adaptive* and aims at:

- Exploiting the wasted time of the machines i.e. as soon as a machine becomes available it is owned for executing some work of the application. We refer that operation to an *application unfold*.
- Keeping respected the machines' ownership, it means that when a machine is requisitioned by its owner the work, part of the application, must be pulled out. We refer that operation to an *application fold*.
- Solving, in some way, the fault tolerance problem: if a failure of a given machine occurs then only the work running on it is re-started on another machine.

In order to achieve that, we use the *MARS* system, a new parallel adaptive programming environment developed at *LIFL (Université de LilleI)*.

The remainder of this paper is organized as follows: Section 2 presents the sequential version of the algorithm and Section 3 details its parallelization. Section 4 describes our

proposed fault-tolerant parallel adaptive version of the algorithm and its implementation with *MARS*. In Section 5, we present its performance evaluation. In Section 6, we draw the conclusion and give some of our further investigations.

2. The sequential algorithm

Let A and B be two matrices of dimension n and B be the inverted matrix of A. Let also A and B be partitioned into $(q \times q)$ matrix blocks of a fixed dimension b ($b = n/q$). The block-based Gauss-Jordan algorithm is the following :

Input: A, $B \leftarrow I(n,n)$, q.
Output: $B = A^{-1}$
For k=1 **to** q **do**
  $A^k_{k,k} \leftarrow (A^{k-1}_{k,k})^{-1}$; $B^k_{k,k} \leftarrow A^k_{k,k}$;
  For j=k+1 **to** q **do** $A^k_{k,j} \leftarrow A^k_{k,k} A^{k-1}_{k,j}$ **End For** (**1**)
  For j=1 **to** k-1 **do** $B^k_{k,j} \leftarrow A^k_{k,k} B^{k-1}_{k,j}$ **End For** (**2**)
  For j=k+1 **to** q **do** (**3**)
    For i=1 **to** q and i $\neq$ k **do**
      $A^k_{i,j} \leftarrow A^{k-1}_{i,j} - A^{k-1}_{i,k} A^k_{k,j}$
    End For
  End For
  For j=1 **to** k-1 **do** (**4**)
    For i=1 **to** q **and** i $\neq$ k **do**
      $B^k_{i,j} \leftarrow B^{k-1}_{i,j} - A^{k-1}_{i,k} B^k_{k,j}$
    End For
  End For
  For i=1 **to** q **and** i $\neq$ k **do** (**5**)
    $B^k_{i,k} \leftarrow - A^{k-1}_{i,k} A^k_{k,k}$
  End For
End For

Each step of the algorithm is made up of five loops. At each step k ($k = 1..q$), the loop 1 (respectively 2) computes the blocks belonging to the line of the pivot ($A^k_{k,k}$) i.e. those of the line k of the matrix A (respectively B) ; the loop 3 (respectively 4) calculates the blocks of all the columns of the matrix A (respectively B) with index above (respectively below) that of the column of the pivot i.e. k. The loop 5 computes the blocks of the column number k of the matrix B except $B^k_{k,k}$. The figure 1 illustrates these computations done at the step $k = 3$ with $q = 5$. The matrices A and B are partitioned into blocks, which are represented by square tiles. The numbers in a bold form inside these tiles designate the loops' numbers that compute their associated matrix blocks.

3. Parallelization

The parallelization of the block-based Gauss-Jordan algorithm consists of exploiting two kinds of parallelism: the

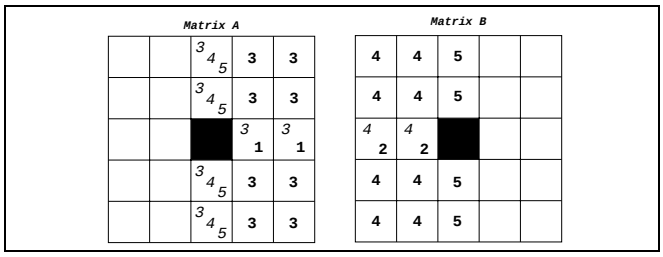

Figure 1. The data dependencies of the loops at the step 3 with $q = 5$

inter-steps parallelism and the *intra-step parallelism*. The first one means that the q steps are partially or completely executed in parallel. The partial parallelism means that at the step k only the inversion of the pivot $A^{k+1}_{k,k}$ of the step $k+1$ is executed by anticipation, and the remainder of this step is triggered only at the end of the step k. The *intra-step parallelism* consists mainly of exploiting the parallelism involved in each of the five loops of the algorithm. It falls into two categories: the *inter-loops parallelism* and the *intra-loop parallelism*.

On the other hand, successive steps of the algorithm are dependent. The *inter-steps parallelism* requires then a complex management of matrix blocks' dependencies that would make its exploitation inefficient. Consequently, only the *intra-step parallelism* is exploited in our implementation, so we develop it in the next sections.

3.1. The inter-loops parallelism

The *inter-loops parallelism* means that the loops are executed in parallel. Here, it is necessary to take into account the dependencies between the loops. We identify two ones: the internal loop of the loop 3 depends on the loop 1 because it uses the block $A^k_{k,j}$ that is computed by the loop 1 ; on the other hand, the internal loop of the loop 4 is dependent on the loop 2 because it uses the block $B^k_{k,j}$ that is computed by the loop 2. Thus, the parallel execution of the internal loop number j of the loop 3 (respectively 4) is triggered after the computation of the block $A^k_{k,j}$ (respectively $B^k_{k,j}$). The figure 1 illustrates the data dependencies between the five loops. Each block associated with a square tile carrying a loop's number in a bold (respectively italic) form is computed (respectively read) by that loop.

3.2. The intra-loops parallelism

The *intra-loops parallelism* consists of executing each of the five loops of the algorithm in parallel. Particularly, the internal loops of the loops numbers 3 and 4 are run each of

them in parallel. There are two kinds of work units in these loops: the *matrix product* and the *matrix triadic* i.e. respectively $X.Y$ and $(Z - X.Y)$, where X, Y and Z are matrix blocks. A crucial issue that must be considered is the granularity of the work units. Indeed, the size of the matrix blocks required by each work unit must be such that the computation cost needed by this work balances its communication cost involved in its sending to a remote node.

3.3. The parallel algorithm

The parallel algorithm is constituted by two modules: the *Application Manager Module (AMM)* and the *Worker Module (WM)*. At run-time, *AMM* is executed by the *master task* and *WM* by the *worker task*. The figure 2 illustrates the architecture of the Gauss-Jordan PAA, the details of the figure are given in the following paragraphs. In the remainder of this article, we will confuse the term *AMM* with the master task and *WM* with the worker task.

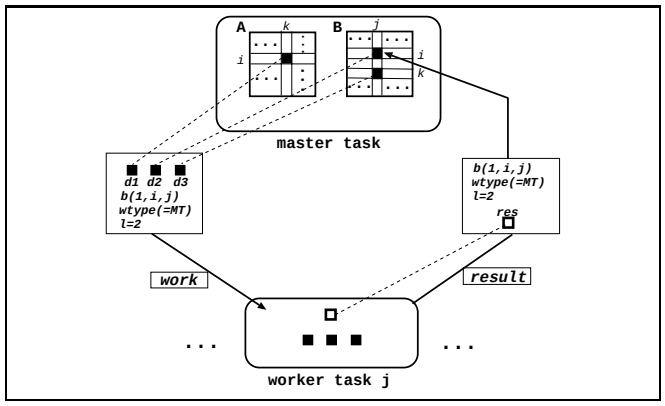

Figure 2. The parallel Gauss-Jordan algorithm

- The AMM module The *AMM* module loads the two matrices A and B and then updates them until it gets the final result. At the begining of each step k ($1 \leq k \leq q$) of the algorithm, it computes the inverse of the current pivot $A^k_{k,k}$ and uses it for updating A and B. Then, it controls and synchronizes the execution of the loops. An array of synchronization bits is used for managing the loops' dependencies.

The *AMM* module extracts from A and B matrix blocks in order to constitute the work units (matrix products and matrix triadics) which are sent to the *WMs* according to a *receiver-initiative strategy*. Each sending of a work unit is done by an *LRPC* which creates a *worker thread* (*WT*) within a *WM* for executing the work unit. After this latter is completed, the *AMM* uses its resulting matrix block to keep up to date the corresponding block in A or B. For example, the figure 2 shows the sending by *AMM* of the work unit which represents the matrix tridiac of the loop number 4 i.e. $B^{k-1}_{i,j} - A^{k-1}_{i,k} B^k_{k,j}$. It also illustrates the sending back by *WM* of the resulting block i.e. $B^k_{i,j}$. The work components are the three data blocks $B^{k-1}_{i,j}$ ($d2$), $A^{k-1}_{i,k}$ ($d1$) and $B^k_{k,j}$ ($d3$), the resulting block descriptor b, the type of work $wtype$ (Matrix Triadic (MT) in our example and MP for Matrix Product) and the loop number l (for control and synchronization). The resulting block descriptor is used for storing the identity of the block to update on the matrix A or B. Its three components represent respectively a number designating the matrix to be updated (0 for A and 1 for B), the line number i and the column number j of the block to be updated.

- The WM module It is indicated above that the work distribution is done according to the *receiver-initiative strategy*. Indeed, each *WM* continually asks *AMM* for work. If there is no work then *WM* waits a *WAIT_PERIOD* time (we fixed it as equal to 10 seconds). Otherwise, it receives the work to do and executes it. After that, it sends the computed block to *AMM*. In the figure 2, the resulting block is characterized by: its descriptor b, its data res, the type $wtype$ of the work unit which computes it and the number l of the loop this work unit belongs to.

4. MARS-based Adaptability

MARS is a multi-threaded environment which allows programming parallel adaptive applications. It is built on top of PM^2 [4], a parallel multi-threaded execution support which couples two librairies: the PVM[1] communication library and a threads' package called *MARCEL*. It aims mainly at keeping respected the machines' ownership and exploiting the wasted time when executing applications in adaptive environments. In such environments nodes (workstations, processors of parallel machines, etc.) are dynamically pulled out from and/or added to the virtual machine executing a parallel adaptive application. As a consequence, work must be dynamically interrupted when a node has to leave the virtual machine and resumed later as soon as a node joins the virtual machine. Then, one has to define how to characterize the partial (interrupted) work, how to fold/unfold it and how to schedule it.

4.1. The partial work

In addition to the information describing an initial work given above i.e. b, l and $wtype$, other information must be added to describe a partial work. These information are

written in a bold form in the figure 3: the blocks' descriptors $b1$, $b2$ and $b3$ of respectively $B_{i,j}^{k-1}$ ($d2$), $A_{i,k}^{k-1}$ ($d1$) and $B_{k,j}^{k}$ ($d3$), I, J, tid and res. The blocks' descriptors are used in order to avoid storing the blocks $d1$, $d2$ and $d3$ themselves. These latter are extracted from the matrices A and B using their descriptors at the unfolding time. The parameters I and J designate respectively the line and column numbers of the next element (resumption point) to compute if a node computing a matrix product is requisitioned. One has to note that the matrix product calculation can either be the matrix product work unit (apart) or that of a matrix triadic work unit. Moreover, this latter is never interrupted when the blocks' minus operation is being computed. That is to say if $X.Y$ of the matrix triadic ($Z - X.Y$) is completed (let us assume $T = X.Y$), the work ($Z - T$) can not be folded, it has to complete because it is not costly in terms of CPU time. The parameter res contains the partial result of the matrix product.

The partial work units (their descriptors) folded by the WMs are stored in the *Partial Work List* (*PWL* in figure 3) managed by the *AMM*. The role of the other list i.e. *AWL* is to manage the fault-tolerance problem.

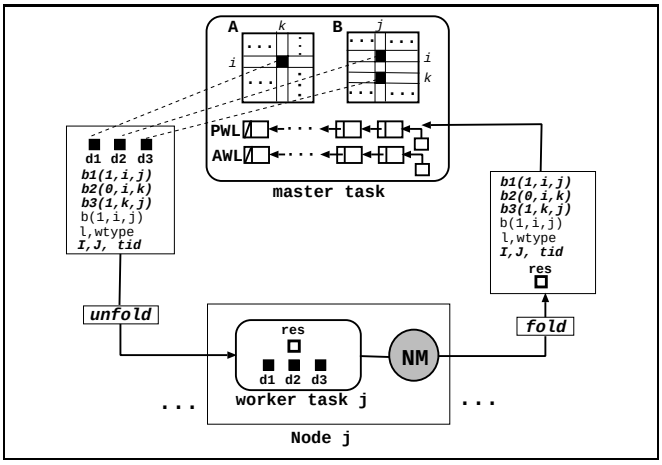

Figure 3. Folding/Unfolding work

4.2. Application folding/unfolding

When a machine is requisitioned by its owner the MARS system informs the AMM about this event which triggers a *folding* operation on that machine. The role of this primitive is to take the partial work (contained in the structure of the figure 3) and to send it back to the *AMM*. The operation is executed by using an *LRPC* which adds the partial work descriptor to the list *PWL*.

On the other hand, an *unfolding* operation occurs when a machine becomes *idle*. The WM running on it asks AMM for work by using an *LRPC* which pulls out a partial work from *PWL*. Then, the *WM* creates a *worker thread* to resume this partial work. The descriptors of the matrix blocks i.e. the data of the partial work are taken from *PWL* and are used to extract the concrete data blocks, as it is shown in the figure 3, from the matrices A and B.

In our implementation, no restriction is imposed regarding the number of folding/unfolding operations of the same work unit but it would be interesting to see if it is necessary to limit it by a threshold to be experimentally determined.

5. Performance Evaluation

In [3], we have studied the influence of the granularity of parallelism on the efficiency of the parallel execution of the algorithm, and the scalability of the algorithm. The results show that the efficiency depends on the size of the blocks. The absolute efficiency can reach 92% for an application with matrices of size 1500×1500 and blocks of size 300×300 on a farm of DEC-ALPHA/OSF1 processors interconnecetd with a Gigaswitch network. On the other hand, absolute efficiencies greater than 60% (67% on a network of four workstations) are obtained on a network of a reasonable number of SUN-Sparc4/Solaris2 workstations.

In this section, we aim at experimentally studying the overhead induced in the management of the reaction of the application to the adaptivity of the material execution support. We considered a meta-system of heterogeneous machines including 8 DEC-ALPHA/OSF-1 processors, 16 SUN-Sparc4/Solaris 2 workstations and 3 SUN-Sparc4/SunOs workstations ; all the 27 machines are interconnected by an Ethernet network. From the application side, we fixed the size of the matrices as equal to 1500×1500 and the size of the blocks as equal to 300×300. We experimented the same application 50 times in the *LIFL Laboratory (Université de LilleI)* on a working day and so machines frequently join and leave the meta-system due to their availability/requisitioning and failures. The number of machines effectively available varies between 20 (minimum number for the 50 runnings) and 27 (maximum number for the 50 runnings) until the end of the execution where it decreases to 0. The average number (average of the average numbers of machines used for each of the 50 runnings) of the machines effectively used is 22.47. The table 1 presents some of the obtained results.

The parameters considered in our study are the CPU time, the communication and waiting time (CW) and the total execution time (ExecTime) of the master task (AMM module). The CPU time is the computing time meaning the time spent in extracting blocks from the matrices A and B, updating the two matrices with the results returned by

Table 1. Matrices' size: 1500×1500, Blocks' size: 300×300, Number of machines: 27

	CPU(in s)	CW(in s)	ExecTime(in s)
Non adaptive	108.011	355.193	463.204
Adaptive	111.744	391.218	502.962

the workers, etc. The CPU time includes also the time consumed by the processes of other users when the experimentations are done on a working day. The communication and waiting time represents the time spent by the master task in distributing work and waiting for results. The total execution time is the sum of the CPU time and the communication and waiting time.

The second and third lines of the table 1 show the results obtained by considering respectively a non adaptive context (week-end night: machines are not requisitioned and there was no failures)) and an adaptive context (working day). We remark that the CPU time consumed by the management of the adaptability is not important : it represents more less than $\frac{111.744-108.011}{111.744} = 3.34\%$ of the total CPU time. Indeed, the 3.34% of CPU time includes the time consumed by the processes of other applications since the experimentations are done on a working day.

In this paper, no experimental comparison is done with the related work [5, 6] because adaptability is not studied in these works and no significant experimental results are presented.

Finally, we can note that in our implementation, the matrices A and B are lodged on the central memory of the machine where the *AMM* resides. Consequentlty, the largest matrices possible for experimentation on machines of $64Kbytes$ is of 1500×1500. Therefore, an out-of-core computing solution is needed. A such solution is presented and analysed in [3].

6. Conclusions and Future Work

We have presented the *MARS* programming methodology of adaptive applications on meta-systems including *NOWs*, and proposed a parallel adaptive version of the block-based Gauss-Jordan algorithm. This one exploits the inter-loops and intra-loops parallelisms. With *MARS*, our application exploits the wasted time. It also keeps respected the machines' ownership. Moreover, our application integrates a scheduling strategy which uses a list *PWL* for storing the partial work. Another list *AWL* is used for fault tolerance of the workers.

The application is implemented and experimented with *MARS* in dedicated and non-dedicated environments. An absolute efficiency of 92% is possible on a dedicated cluster of DEC/ALPHA processors interconnected by a Gigaswitch network and an absolute efficiency of 67% can be obtained on a dedicated Ethernet network of SUN-Sparc4 workstations. Moreover, the algorithm is experimeted on a non-dedicated meta-system of 27 heterogeneous machines show that the management of the adaptability is not costly even in presence of processes of other users.

Actually, we are implementing our out-of-core solution [3]. Our first further investigation will be its performance evaluation. Another feature we address in the near future is the experimentation of our application on a national wide area network.

References

[1] A. Geist, A. Beguelin, and J. Dongarra et al., editors. *PVM: Parallel Virtual Machine, A User's guide and tutorial for Networked Parallel Computing*. MIT Press, 1994.

[2] Zouhir Hafidi, El Ghazali Talbi, and Jean-Marc Geib. Méta-systèmes : Vers l'intégration des machines parallèles et les réseaux de stations hétérogènes. *Calculateurs Parallèles, Ed. Hermès*, Vol. 9(4):435–450, 1997.

[3] N. Melab, S. Petiton, and E-G. Talbi. A new parallel adaptive block-based Gauss-Jordan algorithm. *Publication interne AS-180, LIFL, Université de LilleI*, Fév 1998.

[4] R. Namyst and J.F. Méhaut. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. *North Holland, Parco'95 proc. Gent Belgium*, pages 279–285, Sept 1995.

[5] Serge G. Petiton. Parallelization on an MIMD computer with real-time scheduler, gauss-jordan example. *Aspects of Computation on Asynchronous Parallel Processors, M.H. Wright (Editor), Elseiver Science Publishers B.V. (North-Holland), IFIP*, 1989.

[6] Ole Tingleff. Systems of linear equations solved by block gauss jordan method using a transputer cube. *Technical report IMM-REP-1995-08, Institute of Mathematical Modelling, Technical University of Danmark*, 1995.