Advances in Computer Science and Engineering
Volume 6, Number 2, 2011, Pages 181-197

SOME PERFORMANCE ISSUES ON LINEAR ALGEBRA ALGORITHMS IN DISTRIBUTED AND GRID COMPUTING ENVIRONMENTS

GIULIANO LACCETTI, MARCO LAPEGNA, VALERIA MELE and DIEGO ROMANO

Department of Mathematics and Applications
University of Naples Federico II
via Cintia - Monte S. Angelo
80126 Naples, Italy
e-mail: giuliano.laccetti@unina.it
marco.lapegna@unina.it
valeria.mele@unina.it
diego.romano@unina.it

Abstract

Due to the dynamic and heterogeneous nature of grid infrastructures, scientific applications with frequent and tight synchronizations among the nodes are unable to achieve high efficiencies, so the client-server paradigm is a programming model very often used in these environments. According to this model, Data Grid applications are usually divided into independent activities that are concurrently solved by the servers. On the other hand, since many scientific applications are characterized by large collections of input data and by dependencies between the tasks, the development of efficient algorithms without unnecessary synchronizations and data transfers is a difficult task. The present work addresses the problem of implementing and assessing a strategy for efficient task scheduling and data management in case of dependencies among tasks in a numerical linear algebra problem. To this end, we used the Block

---

Keywords and phrases: parallel and distributed computing, product matrix algorithms, performance analysis.

Received November 22, 2010

Matrix Multiplication Algorithm implemented in the NetSolve distributed computing environment as case study, and we introduced some efficiency parameters to assess the algorithm.

1. Introduction and Motivation

A grid infrastructure aggregates scattered computing and data resources in order to create a single computing system image [8]. The *hardware* of this single computing system is often characterized by slow and non-dedicated Wide Area Networks connecting very fast and powerful processing nodes (that can also represent supercomputers or large clusters) scattered on a huge geographical territory, whereas its *operating system* (the grid middleware) is responsible to find and allocate resources for scientists applications, taking into account the status of the whole grid. Many papers focus on this aspect of grid computing, addressing issues such as resources brokering, e.g., [6, 17], performance contract definition and monitoring, e.g., [5, 12, 14], and migration of the applications in case of contract violations, e.g., [11, 16]. In any case, it is important to underline that distributed computing environments are composed by heterogeneous computational resources, both from the static (processors, operating systems, arithmetic, ...) and from the dynamic (workload of the systems, effective bandwidth of the networks, ...) point of view, such that an efficient synchronization among the nodes is very difficult. For this reason, one of the main approaches to the development of distributed applications is based on the client-server programming model where the application is divided into a large number of essentially independent tasks that are dispatched to several servers, and a “coordinator” task managed by the client module. In the parallel computing community, problems that can be solved with this approach are called “*pleasingly*” or “*embarrassingly*” parallel and one of the most significant examples in this sense is the SETI@home project [15]. However, beyond such example of mere networked computing, several common scientific applications are characterized by a very large set of input data and dependencies among subproblems, so that the choice of the most powerful computational resources made by the middleware is not sufficient to achieve good performance, but the definition of suitable methodologies is also essential to minimize synchronization among tasks and to distribute application data onto the grid components in order to overlap communication and computation. As a case study, we consider a Block Matrix Multiplication (BMM) algorithm for a client-server distributed computing environment, because it is a basic linear algebra computational kernel representative of similar other computational kernels like LU, LL^T and QR factorizations, and for this reason often required by several applications. On the other hand, it encompasses a large amount of data movements among CPUs and memories, and the task of minimizing the synchronization overhead among the nodes by using effective data caching strategies is challenging. Few papers are available in this research area, e.g., [3, 7]. In a previous work, we introduced a distributed client-server algorithm for this problem [4], so that in this paper, we mainly introduce a performance evaluation procedure aimed to assess the algorithms.

In this work, the algorithms are implemented in the computational environment able to support a client-server programming model, and where the underlying computing environment is in charge of the resources selection by means of its own dynamic allocation strategies, and the computational information about the servers (including their availability, load, processor speed) are hidden to the client. During recent years, several computing environments have been developed with the scope of addressing these topics, while allowing, at the same time, a friendly access to remote resources. Among them, we mention NetSolve [1] and Condor [10].

Our work is structured as follows: In Section 2, we will shortly introduce different distributed algorithms for the BMM problem; in Section 3, we introduce our performance evaluation procedure based on some parameters aimed to asses the algorithms on these environments; finally in Section 4, we describe the computational experiments.

2. Distributed Algorithms for Block Matrix Multiplication

Consider the following matrix multiplications problem for two dense matrices A and B:

$$C = A \cdot B. \tag{1}$$

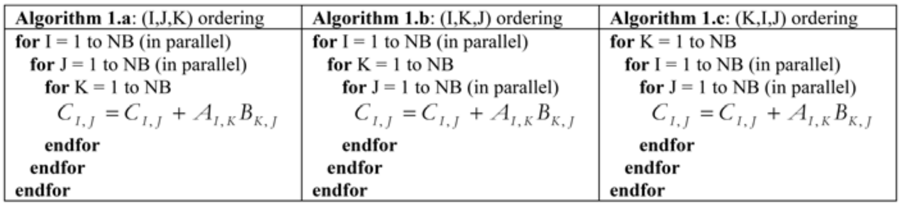
For the sake of simplicity, we will assume that each matrix $X \in \{A, B, C\}$ is a square $n \times n$ matrix, and it is block partitioned as follows:

$$X = \begin{pmatrix} X_{1,1} & \cdots & X_{1,NB} \\ \vdots & & \vdots \\ X_{NB,1} & \cdots & X_{NB,NB} \end{pmatrix}. \tag{2}$$

Blocks $X_{I,J}$ have dimension $r \times r$, with n divisible by r, so that $NB = n/r$ is an integer. From previous definitions, we have that each block of the C matrix is given by:

$$C_{I,J} = \sum_{K=1}^{NB} A_{I,K} B_{K,J}, \quad \forall I,\ J = 1, \ldots, NB. \tag{3}$$

From equation (3), we observe that blocks $C_{I,J}$ can be computed independently each other, so that Figure 1 shows three parallel versions obtained by the permutation of the loops indices in the standard BMM algorithm $C = A \cdot B$.



Algorithm 1.a: (I,J,K) ordering

```
for I = 1 to NB (in parallel)
  for J = 1 to NB (in parallel)
    for K = 1 to NB
      $C_{I,J} = C_{I,J} + A_{I,K} B_{K,J}$
    endfor
  endfor
endfor
```

Algorithm 1.b: (I,K,J) ordering

```
for I = 1 to NB (in parallel)
  for K = 1 to NB
    for J = 1 to NB (in parallel)
      $C_{I,J} = C_{I,J} + A_{I,K} B_{K,J}$
    endfor
  endfor
endfor
```

Algorithm 1.c: (K,I,J) ordering

```
for K = 1 to NB
  for I = 1 to NB (in parallel)
    for J = 1 to NB (in parallel)
      $C_{I,J} = C_{I,J} + A_{I,K} B_{K,J}$
    endfor
  endfor
endfor
```

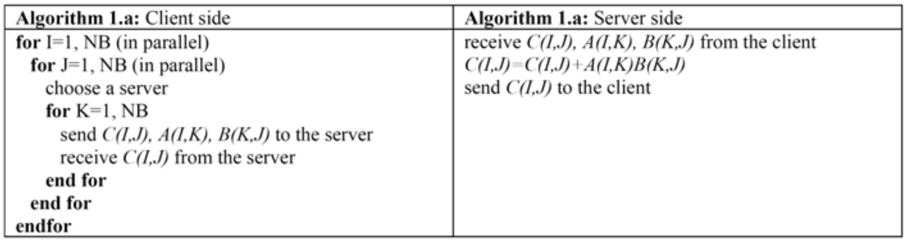
Figure 1. Three standard versions for the BMM algorithm.

Note that other versions, obtained by the permutation of indices I and J, are equivalent to these ones and that all the versions are based on the same computational kernel:

$$C_{I,J} = C_{I,J} + A_{I,K} B_{K,J}. \tag{4}$$

In a client-server implementation, for given values of I, J and K, the client sends to a server the three blocks $A_{I,K} B_{K,J}$ and $C_{I,J}$, so that the server can update the block $C_{I,J}$ and send back the result to the client. It is important to remark that the only possible parallelism is always on indices I and J, so that only the blocks $C_{I,J}$ can be computed independently among them. This is not possible using index K, because of the risk of "race condition" on accessing to the blocks $C_{I,J}$ for different values of K. As a consequence, in order to reduce the synchronization overhead accessing these blocks in a client-server implementation, it is essential to define which algorithm in Figure 1 must be used to compute the several matrix operations involving blocks $A_{I,K} B_{K,J}$ and $C_{I,J}$. For a more deep analysis, we observe that in the (I, J, K) ordering (Algorithm 1.a), the client generates NB^2 independent threads of computation, each of them managing the sum on index K in equation (3).

With the (I, K, J) ordering (Algorithm 1.b), the client generates only *NB* independent threads of computation, each of them generating *NB* parallel tasks at every step of index *K*. Finally, with the (K, I, J) ordering (Algorithm 1.c) at each step of index *K*, the client generates NB^2 parallel tasks that have to be completed before the client can generate new tasks.



Algorithm 1.a: Client side

```
for I=1, NB (in parallel)
  for J=1, NB (in parallel)
    choose a server
    for K=1, NB
      send C(I,J), A(I,K), B(K,J) to the server
      receive C(I,J) from the server
    end for
  end for
endfor
```

Algorithm 1.a: Server side

```
receive C(I,J), A(I,K), B(K,J) from the client
C(I,J)=C(I,J)+A(I,K)B(K,J)
send C(I,J) to the client
```

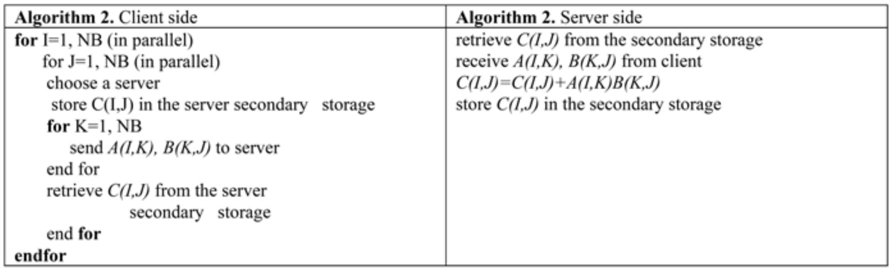
Figure 2. The client-server implementation of the BMM algorithm with (I, J, K) ordering.

In Figure 2, as an example, we report the distributed client-server implementation of the (I, J, K) ordering of the BMM algorithm. Now it is important to remark that, in the client-server programming model, data are stored in the client memory (or in a repository close to the client) and they are sent in chunks to servers for the computations; once the computation has been completed, the results are returned to the client. However, the data movement between client and server in a computational grid is similar to the data transfer between memories and processing unit in a single Non Uniform Memory Access (NUMA) machine. A NUMA machine is characterized by a memories hierarchy where fast and small memories (main memory and caches) are positioned at the higher level, whereas slow and large memories (secondary and remote memories) are located at the lower ones. Table 1 shows typical peak bandwidth, latency and size, for four different memory levels when accessed from the server. The illustrated values refer to a common workstation usually available in a distributed computing environment and are not representative of leading edge technology.

Table 1. Typical values for bandwidth and latency for different memory levels

	Bandwidth	Latency	Size
Server main memory	10 GByte/sec	2-10 ns	4 GBytes
Server secondary storage	100 MByte/sec	5 ms	512 TBytes
Remote client (LAN)	12.5 MByte/sec	10 ms	10 TBytes
Remote client (WAN)	< 1 MByte/sec	100 ms	> 100 TBytes

It is commonly acknowledged that the key strategy to achieve high performances with a NUMA machine is an extensive use of caching methodologies at each level of the memory hierarchy. In this model, the highest levels are usually managed by the compilers or by some highly optimized mathematical software library, but lowest levels must be managed by the application. Since scientific applications rarely can be divided in totally independent tasks and some data dependencies are always present among them, the definition of methodologies and the development of software tools for an effective data distribution among the components of a computational grid assume a key role in grid computing.



Algorithm 2. Client side

```
for I=1, NB (in parallel)
    for J=1, NB (in parallel)
     choose a server
      store C(I,J) in the server secondary   storage
     for K=1, NB
        send A(I,K), B(K,J) to server
     end for
     retrieve C(I,J) from the server
                  secondary   storage
     end for
endfor
```

Algorithm 2. Server side

```
retrieve C(I,J) from the secondary storage
receive A(I,K), B(K,J) from client
C(I,J)=C(I,J)+A(I,K)B(K,J)
store C(I,J) in the secondary storage
```

Figure 3. The distributed client-server implementation of the BMM algorithm with (I, J, K) ordering and caching of intermediate results in the server secondary storage.

The use of a server secondary storage as a cache for the intermediate results, therefore allows to locate them to a higher level in the memory hierarchy and avoids unnecessary data transfers toward the client memories. Furthermore, if the entire sequence has to be repeated several times, then it is possible to overlap data communication and stage computation by keeping intermediate data in higher level memories. The following Algorithm 2 in Figure 3 implements the described caching strategy for the (I, J, K) ordering of the BMM algorithm. A similar approach to data management in distributed environments is described in [7], where the server main memory replaces the server secondary storage as cache. The main advantage of the approach described in the current paper is the larger amount of space available for caching the intermediate data, with a data access time still negligible compared to the time for accessing the remote client memory.

3. Algorithms Analysis

For a complete performance analysis, we have to notice that in a distributed environment, classical parameters like Speedup and Efficiency cannot be used since the number of used nodes is not defined by the user through the applications, but they are determined by the computational environment. Furthermore, the primary goal for using these environments is the opportunity of aggregating scattered and unused resources rather than simply reducing the execution time [9]. In any case, we can study the behavior of the total execution time when the problem dimension n changes, aiming at measuring the influence of the computational environment on the BMM distributed algorithm.

We begin our study by comparing the computational cost of the three algorithms in Figure 1. Firstly, denote $t_{ijk} > 0$ to be the execution time (computation and communication) necessary to resolve the computational kernel (4) and $T_{ijk}(NB)$, $T_{ikj}(NB)$, $T_{kij}(NB)$ to be, respectively, the total execution times to solve the problem (1) with dimension $NB = n/r$ using the three algorithms in Figure 1. With the previous definition:

Lemma 1. *Given the total execution times*:

$$T_{ijk}(NB), \quad T_{ikj}(NB), \quad T_{kij}(NB),$$

then

$$T_{ijk}(NB) \leq T_{ikj}(NB) \leq T_{kij}(NB). \tag{5}$$

Proof. By the DAGs in Figure 2, it is easy to prove that:

$$T_{ijk}(NB) = \max_{i,\,j} \sum_{k} t_{ijk}, \quad T_{ikj}(NB) = \max_{i} \sum_{k} \max_{j} t_{ijk}, \quad T_{kij}(NB) = \sum_{k} \max_{i,\,j} t_{ijk}.$$

So, the inequalities hold.

Therefore, the (I, J, K) ordering described by Algorithm 1.a is more suitable to a distributed client-server implementation compared to the other two orderings. The least suitable one is the (K, I, J) ordering. Furthermore, it is reasonable to suppose $T_{ijk}(1) = T_{ikj}(1) = T_{kij}(1)$. This is justified because with only one block $(NB = 1)$, the three algorithms are equivalent. Since the (I, J, K) ordering exhibits the smaller total execution time, in the following, we concentrate our attention only on this one, but similar results hold for the other orderings.

To assess the performance of the client-server implementation described in Figure 2 in a distributed computing environment, let us examine before an ideal case, where the environment is composed by homogeneous, dedicated and unbounded resources (e.g., number of nodes and networks bandwidth). In this environment, let $T^*_{ijk}(NB), T^*_{ikj}(NB), T^*_{kij}(NB)$ be, respectively, the ideal total execution times of the algorithms in Figure 1. From a theoretical point of view, we can assume that when the number of blocks *NB* increases, there are always available nodes and network bandwidth to perform the tasks. In this case, the execution time of each task $t_{ijk} = \tau$ is equal for all the values of *I*, *J*, *K*, and the ideal total execution times in a distributed environment are

$$T^*_{ijk}(NB) = T^*_{ikj}(NB) = T^*_{kij}(NB) = NB \cdot \tau. \tag{6}$$

Now we consider the problem (1) of size $\alpha \cdot n = \alpha \cdot NB \cdot r$, where $\alpha \geq 1$ is a scaling parameter, with the purpose of studying the influence of the computational environment on the algorithm. Then we define $T^*_{ijk}(\alpha \cdot NB)$ as the total execution time to solve such larger problem in the ideal case, and we define the parameter

$$R^*_{ijk}(NB, \alpha) = \frac{T^*_{ijk}(\alpha \cdot NB)}{T^*_{ijk}(NB)}. \tag{7}$$

This parameter assesses the ideal growth factor for the total execution time when an α-times larger problem is solved. Of course, it is easy to prove that:

$$R^*_{ijk}(NB, \alpha) = \alpha. \tag{8}$$

Same results hold also for $T^*_{ikj}(\alpha \cdot NB)$ and $T^*_{kij}(\alpha \cdot NB)$, so that equation (8) shows a linear growth with *NB* for the ideal total execution time for all the algorithms in Figure 1, when the matrix dimension grows and the block dimension $r = n/NB$ is constant.

However, the number of nodes and the sustained network bandwidth are limited and it is not possible to have a perfect parallelism. For such a reason, it is fully reasonable to introduce the assumptions:

Heuristic 1.

$$T_{ijk}(NB) \geq T^*_{ijk}(NB). \tag{9}$$

Heuristic 2.

$$T_{ijk}(\alpha \cdot NB) \geq \alpha T_{ijk}(NB). \tag{10}$$

These assumptions mean that the actual total execution time cannot be smaller than the ideal one, and when we solve an α-times larger problem in a real environment, we cannot achieve an actual growth factor smaller than the ideal one. Therefore, we define the parameter

$$R_{ijk}(NB, \alpha) = \frac{T_{ijk}(\alpha \cdot NB)}{T_{ijk}(NB)} \tag{11}$$

as the measure of the actual growth factor for $T_{ijk}(NB)$ when an α-times larger problem is solved. It is easy to prove, by using Heuristic 2, that $R_{ijk}(NB, \alpha) \geq \alpha$.

By comparing parameters $R_{ijk}(NB, \alpha)$ and $R^*_{ijk}(\alpha, NB)$, we can now evaluate the influence of the computational environment on the distributed algorithm. Now we define the parameter:

$$E_{ijk}(NB, \alpha) = \frac{R_{ijk}(NB, \alpha)}{R^*_{ijk}(NB, \alpha)}. \tag{12}$$

This parameter evaluates how much $R_{ijk}(NB, \alpha)$ is larger than $R^*_{ijk}(NB, \alpha)$.

Of course, by using (8) and Heuristic 2, we have

$$E_{ijk}(\alpha, NB) = \frac{R_{ijk}(\alpha, NB)}{R^*_{ijk}(\alpha, NB)} = \frac{R_{ijk}(\alpha, NB)}{\alpha} \geq 1.$$

Furthermore:

Lemma 2. *Given the previous definitions, we have*

$$T_{ijk}(\alpha \cdot NB) = \alpha E_{ijk}(NB, \alpha) T_{ijk}(NB). \tag{13}$$

Proof. This is because

$$E_{ijk}(NB, \alpha) = \frac{R_{ijk}(NB, \alpha)}{R^*_{ijk}(NB, \alpha)} = \frac{T_{ijk}(\alpha \cdot NB)}{T_{ijk}(NB)} \frac{T^*_{ijk}(NB)}{T^*_{ijk}(\alpha \cdot NB)} = \frac{T_{ijk}(\alpha \cdot NB)}{\alpha T_{ijk}(NB)},$$

then the thesis.

From (13), we observe that $\alpha E_{ijk}(NB, \alpha)$ is the actual growth factor for the total execution time, so $E_{ijk}(NB, \alpha)$ can be taken as a measure of the influence of the computational environment on the performance of the algorithm. Of course, $R^{*}_{ijk}(NB, \alpha)$ is a not decreasing function of α, so, by the Heuristic 2, we can assume that the same property holds also for $R_{ijk}(NB, \alpha)$. Therefore, if $E_{ijk}(NB, \alpha)$ is a constant or a moderately increasing function of α, then we can consider the algorithm as suitable for a distributed execution and able to exploit the parallelism of the computational environment. However, in general, a limitation in the resources (number of nodes, networks bandwidth, …) prevents the parallel execution of a large number of tasks, so we found that $E_{ijk}(NB, \alpha)$ is a significantly increasing function of α. Therefore, in order to understand the actual gain obtained when using a distributed environment in place of a sequential one, it can be useful to compare $R_{ijk}(NB, \alpha)$ not only with the scale factor α as in the definition of $E_{ijk}(NB, \alpha)$, but also with other functions like $f(\alpha) = \alpha^2$ or $f(\alpha) = \alpha^3$. Actually, let us note that $f(\alpha) = \alpha^3$ can be considered the worst growth factor for the total execution time, because it is the growth factor of a BMM algorithm in a distributed environment with only one server.

We define therefore:

$$E^{(2)}_{ijk}(NB, \alpha) = \frac{R_{ijk}(NB, \alpha)}{\alpha^2} \quad \text{and} \quad E^{(3)}_{ijk}(NB, \alpha) = \frac{R_{ijk}(NB, \alpha)}{\alpha^3}. \qquad (14)$$

These parameters compare $R_{ijk}(NB, \alpha)$, respectively, with $f(\alpha) = \alpha^2$ and $f(\alpha) = \alpha^3$. If $E^{(2)}_{ijk}(\alpha, NB)$ is a constant or moderately increasing function, then we can yet consider as convenient to execute the algorithm in a distributed environment; but a significant increasing function $E^{(2)}_{ijk}(\alpha, NB)$ means that we are not able to gain any benefits from the execution in a distributed environment with respect to the execution in a sequential environment.

Finally, we conclude this section with a comparison between Algorithm 1 in Figure 3 (*IJK* ordering *without* data caching) and Algorithm 2 in Figure 6 (*IJK* ordering *with* data caching). Let now τ_S and τ_R be, respectively, the access times to the server secondary storage and to the remote client memories, and $C_{ijk}^{*(1)}$ and $C_{ijk}^{*(2)}$ be the ideal total communication costs for Algorithm 1 and Algorithm 2, respectively. We concentrate our attention only on the communication cost because the computation cost is equal in both algorithms and because it is the dominant part in the total execution time. We firstly observe that, since the computation of the kernel (4) requires the communication of $4r^2$ data among client and server, in (6), we have $\tau = 4\tau_R r^2$, so the ideal communication cost for the complete computation of each block $C(I, J)$ with Algorithm 1 is:

$$C_{ijk}^{*(1)}(NB) = 4NB\tau_R r^2. \tag{15}$$

Since $\tau_R = \gamma\tau_S$ with $10 < \gamma < 100$, the ideal communication cost of the Algorithm 2 is $\tau = 2(\tau_R + \tau_S)r^2$, so that:

$$C_{ijk}^{*(2)}(NB) = 2NB(\tau_R + \tau_S)r^2 < C_{ijk}^{*(1)}(NB). \tag{16}$$

We define:

$$S^*(NB) = \frac{C_{ijk}^{*(2)}(NB)}{C_{ijk}^{*(1)}(NB)} \tag{17}$$

as the measure of the ideal reduction factor for the (I, J, K) ordering when a caching strategy is used. It is easy to prove:

Lemma 3. *Given the definition of* $S^*(NB)$, *we have*

$$S^*(NB) = \frac{C_{ijk}^{*(2)}(NB)}{C_{ijk}^{*(1)}(NB)} = \frac{2NB(\tau_R + \tau_S)r^2}{4NB\tau_R r^2} = \frac{\gamma + 1}{2\gamma}. \tag{18}$$

Equation (18) shows the ideal value for the reduction factor when a caching strategy is used. It should be compared with the actual reduction factor, that is with the ratio:

$$S(NB) = \frac{C_{ijk}^{(2)}(NB)}{C_{ijk}^{(1)}(NB)}, \tag{19}$$

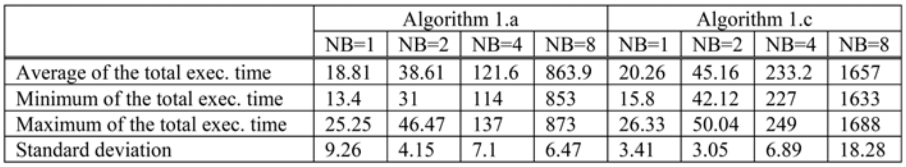
where $C_{ijk}^{(1)}$ and $C_{ijk}^{(2)}$ are the actual communication costs of the two algorithms.

4. Computational Experiments

In this section, we describe the results of several tests aimed to evaluate our algorithms using the procedure described in Section 3. For our experiments, we used NetSolve 2.0 distributed computing infrastructure [1]. This is a software environment based on a client-agent-server paradigm that provides a transparent and inexpensive access to remote hardware and software resources.

A first set of experiments is aimed to evaluate the effectiveness of Algorithm 1.a when compared to Algorithm 1.b and Algorithm 1.c. Then on the basis of Equation (5), we implement the (I, J, K) ordering (Algorithm 1.a) and the (K, I, J) ordering (Algorithm 1.c), i.e., the best and the worst expected version. In this first experiment, the servers are located at the University of Tennessee and the client is located in our Department. This software infrastructure can be called *Wide Area System* (*WAS*), because of the underlying geographical networks. In these experiments, we evaluated the total execution times for calculations in problem (1), considering square matrix of order $n = 250,$ 500, 1000, 2000 and a fixed block size $r = 250.$ With these values, the number of blocks $NB = 1,$ 2, 4, 8.

Table 2. Timing results in seconds for Algorithm 1.a and Algorithm 1.c on a WAS



	Algorithm 1.a				Algorithm 1.c			
	NB=1	NB=2	NB=4	NB=8	NB=1	NB=2	NB=4	NB=8
Average of the total exec. time	18.81	38.61	121.6	863.9	20.26	45.16	233.2	1657
Minimum of the total exec. time	13.4	31	114	853	15.8	42.12	227	1633
Maximum of the total exec. time	25.25	46.47	137	873	26.33	50.04	249	1688
Standard deviation	9.26	4.15	7.1	6.47	3.41	3.05	6.89	18.28

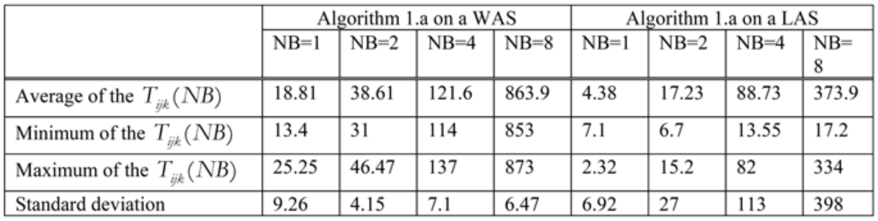
In Table 2, there are the total execution times for Algorithm 1.a and Algorithm 1.c on the WAS. In order to report realistically, the impact of the fluctuation in the network traffic, the values are, respectively, the averages of $T_{ijk}(NB)$ and $T_{kij}(NB)$ over 10 executions. Furthermore, the table lists the minimum and maximum achieved total execution times and the standard deviation over the 10 executions. Best performance of Algorithm 1.a is evident with an average execution time lower in each test. The high standard deviation values are motivated by the variability of the workload in the geographic networks. Table 3 reports further results of the comparison between the two algorithms on the basis of other parameters (11), (12) and (14) defined in Section 3 with $\alpha = 2,$ that is doubling the size of the matrix in each experiment. In this table we note that, even if the Algorithm 1.a typically shows parameter values better than Algorithm 1.c, in both cases, we achieve increasing values for $E^{(2)}(2, NB)$ and $E^{(3)}(2, NB)$. More precisely, we observe a behavior of the total execution times worse than the one of a sequential execution in a environment with a single server, so the use of a WAS in this case is not feasible.

A second set of experiments is aimed to compare the execution times of Algorithm 1.a on two different NetSolve systems: a WAS as previously described and a Local Area System (LAS), where all resources are connected to the Local Area Network of our Department at 100 Mbits. The results of the experiments on a LAS, conducted with the same values of *n* and *r* used in previous experiments for a WAS, are shown in Table 4.

Table 3. Performance analysis of Algorithm 1.a and Algorithm 1.c on a WAS

	Algorithm 1.a			Algorithm 1.c		
	$NB = 1$	$NB = 2$	$NB = 4$	$NB = 1$	$NB = 2$	$NB = 4$
$R(2, NB)$	2,05	2,23	2,23	2,23	3,15	7,10
$E(2, NB)$	1,03	1,11	1,11	1,11	1,57	3,55
$E^{(2)}(2, NB)$	0,51	0,56	0,56	0,56	0,79	1,78
$E^{(3)}(2, NB)$	0,26	0,28	0,28	0,28	0,39	0,89

Table 4. Timing results in seconds for Algorithm 1.a on two different systems



	Algorithm 1.a on a WAS				Algorithm 1.a on a LAS			
	NB=1	NB=2	NB=4	NB=8	NB=1	NB=2	NB=4	NB= 8
Average of the $T_{ijk}(NB)$	18.81	38.61	121.6	863.9	4.38	17.23	88.73	373.9
Minimum of the $T_{ijk}(NB)$	13.4	31	114	853	7.1	6.7	13.55	17.2
Maximum of the $T_{ijk}(NB)$	25.25	46.47	137	873	2.32	15.2	82	334
Standard deviation	9.26	4.15	7.1	6.47	6.92	27	113	398

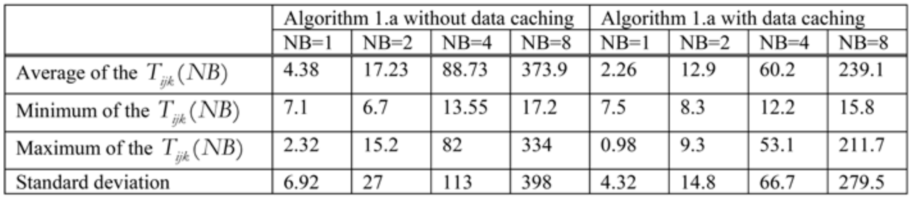
This table shows the average, the minimum, the maximum and the standard deviation of $T_{ijk}(NB)$ over 10 executions. Firstly, we observe for the Local Area System an average execution time and a standard deviation much smaller than those obtained in the Wide Area System, which is due to the smallest latency and the higher bandwidth network. In order to quantify the performance gain that we achieve using a Local Area Network System, in Table 5, we show the values of the parameters for the assessment of the performance introduced in Section 3. We observe that, unlike WAS, the values of $E^{(2)}(2, NB)$ and $E^{(3)}(2, NB)$ are descending. More precisely, we measured decreasing values for $E^{(2)}(2, NB)$, and that means a growth factor included between $R(2, NB) = 2$ (the ideal case) and $R(2, NB) = 4$ (a still convenient case). Then, the use of Algorithm 1.a on a Local Area System produces a real gain on the total execution time, as compared to a sequential execution in an environment with a single server.

Table 5. Performance analysis of Algorithm 1.a on two different infrastructures

	Algorithm 1.a on a WAS			Algorithm 1.a on a LAS		
	$NB = 1$	$NB = 2$	$NB = 4$	$NB = 1$	$NB = 2$	$NB = 4$
$R(2, NB)$	2,05	3,15	7,10	9,47	5,15	4,21
$E(2, NB)$	1,03	1,57	3,55	4,73	2,57	2,11
$E^{(2)}(2, NB)$	0,51	0,79	1,78	2,37	1,29	1,05
$E^{(3)}(2, NB)$	0,26	0,39	0,89	1,18	0,64	0,53

A third set of experiments is aimed to test the data caching strategy described in Section 2. In order to manage data efficiently, NetSolve includes two tools: the Request Sequencing and the Data Storage Infrastructure (DSI), but they are unable to implement the caching strategy previously described. Therefore, in order to fully implement a caching methodology, it has been necessary to modify the NetSolve DSI implementation to some extent, as described in [4]. The experiments have been carried out on the LAS in our department, where the servers are workstations running at 2.4 GHz, each of them provided with a Parallel ATA disk adapter with a peak transfer rate of 100 MByte/sec. The secondary storage is used as a cache for the intermediate results in each server. Then, we implemented Algorithm 1.a without data caching as shown in Figure 1, and Algorithm 1.a with data caching as shown in Figure 3. Table 6 shows the average, the minimum, the maximum and the standard deviation of $T_{ijk}(NB)$ over 10 executions. The matrix dimension $n = 250,\ 500,$ 1000 and 2000 and the block size $r = 250$ are the same in both tables. The results show a significant reduction of the total execution time for the computation of the entire matrix multiplication when a caching strategy is used.

Table 6. Timing results in seconds for Algorithm 1.a with different caching strategies on a LAS



	Algorithm 1.a without data caching				Algorithm 1.a with data caching			
	NB=1	NB=2	NB=4	NB=8	NB=1	NB=2	NB=4	NB=8
Average of the $T_{ijk}(NB)$	4.38	17.23	88.73	373.9	2.26	12.9	60.2	239.1
Minimum of the $T_{ijk}(NB)$	7.1	6.7	13.55	17.2	7.5	8.3	12.2	15.8
Maximum of the $T_{ijk}(NB)$	2.32	15.2	82	334	0.98	9.3	53.1	211.7
Standard deviation	6.92	27	113	398	4.32	14.8	66.7	279.5

Furthermore, in Table 7, we report the values for the performance analysis introduced in Section 3. We observe that for Algorithm 1.a with a caching strategy, we achieve growth factors $R(2,\ NB)$ smaller than the ones from the same algorithm without data caching. Algorithm 1.a on a LAS without data caching.

Table 7. Performance analysis of two versions of Algorithm 1.a on a local area system

	Algorithm 1.a on a LAS without data caching			Algorithm 1.a on a LAS with data caching		
	$NB = 1$	$NB = 2$	$NB = 4$	$NB = 1$	$NB = 2$	$NB = 4$
$R(2,\ NB)$	9,47	5,15	4,21	5,71	4,67	3,97
$E(2,\ NB)$	4,73	2,57	2,11	2,85	2,33	1,99
$E^{(2)}(2,\ NB)$	2,37	1,29	1,05	1,43	1,17	0,99
$E^{(3)}(2,\ NB)$	1,18	0,64	0,53	0,71	0,58	0,50

Table 8. Actual and ideal ratios among the communications costs of the two version of Algorithm 1.a

	$NB = 1$	$NB = 2$	$NB = 4$	$NB = 8$
$S(NB)$	1,24	0,75	0,68	0,64
$S^*(NB)$	0,55	0,55	0,55	0,55

Finally, the effectiveness of the changes in the IBP infrastructure described in Section 4, is confirmed by Table 8, showing the achieved values for the actual and ideal ratios among the communication costs $S(NB)$ and $S^*(NB)$ introduced in Section 3. Actually, mainly for large problems, we observe that the achieved values for $S(NB)$ are very close to the ideal values $S^*(NB)$.

5. Conclusions

This work mainly pursuits a double purpose. Firstly, it describes an effective methodology for task scheduling and for data placement among resources related to the implementation of a Block Matrix Multiplication in a client-server distributed environment. Secondly, it introduces a procedure to assess the performance of algorithms in a distributed client-server environment. The procedure is based on a performance model that is validated with several experimental results in two different distributed environments: a Local Area System based on computing nodes connected by local networks and a Wide Area System based on geographical networks. From the performance analysis, we achieve some interesting conclusions reported in Section 4. In any case, the main result is the confirmation that the use of suitable caching strategies for the implementation of a Block Matrix Multiplication algorithm in a distributed system based on local area networks is competitive with more expensive parallel system based on dedicated computing nodes or networks. Even if the experiments refer only to the BMM problem, the general structure of the algorithm is general enough, so we believe that similar results could be achieved also on other linear algebra problems, like LU, LL^T and QR factorizations.

References

[1] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi and S. Vadhiyar, User's Guide to NetSolve V. 2.0, Univ. of Tennessee, 2004. See also Net-Solve home page-URL: http://icl.cs.utk.edu/netsolve/index.html.

[2] A. Bassi, M. Beck, T. Moore, J. S. Plank, M. Swany, R. Wolski and G. Fagg, The internet backplane protocol: a study in resource sharing, Future Generation Computing Systems 19 (2003), 551-561.

[3] O. Beaumont, V. Boudet, F. Rastello and Y. Robert, Matrix multiplication on heterogeneous platforms, IEEE Trans on Parallel and Distributed Systems 12 (2001), 1033-1051.

[4] L. Carracciuolo, G. Laccetti and M. Lapegna, Implementation of effective data management policies in distributed and grid computing environments-in PPAM07: Parallel Processing and Applied Mathematics, Wyrzykowski et al., eds, LNCS 4967, Springer-Verlag, 2008, pp. 902-911.

[5] P. Caruso, G. Laccetti and M. Lapegna, A Performance Contract System in a Grid Enabling, Component Based Programming Environment, EGC2005: Advances in Grid Computing, P. M. A. Sloot et al., eds, LNCS 3470, Springer-Verlag, 2005, pp. 982-992.

[6] K. Czajkowsky, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, A resource selection management architecture for metacomputing systems, Proc. of PPS/SPDP'98 Workshop on Job Scheduling Strategies for parallel Proc., 1998.

[7] J. Dongarra, J. F. Pineau, Y. Robert, Z. Shi and F. Vivien, Revisiting matrix product on master worker platforms, Parallel and Distributed Processing Symposium, 2007, IPDPS 2007.

[8] I. Foster and C. Kesselman, eds., Computational grid, The Grid: Blueprint for a Future Generation Computing Infrastructure, Morgan Kaufman, 1998.

[9] G. Fox, Message passing from parallel computing to the grid, IEEE Computing in Science and Engineering 4 (2002), 70-73.

[10] M. Litzkow, M. Livny and M. Mutka, Condor: a hunter of idle workstation, Proc. of 8th Int. Conf. Distributed Computing Systems, IEEE press, 1988, pp. 104-111. See also URL www.cs.wisc.edu/condor.

[11] A. Murli, V. Boccia, L. Carracciuolo, L. D'Amore, G. Laccetti and M. Lapegna, Monitoring and Migration of a PETSc-based Parallel Application for Medical Imaging in a Grid computing PSE, IFIP International Federation for Information Processing, Vol. 239: Grid-based Problem Solving Environments: Implications for Development and Deployment of Numerical Software, P. W. Gaffney and J. C. T. Pool, eds., Springer-Verlag, 2007, pp. 421-432.

[12] F. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche and S. Vadhiyar, Numerical Libraries and the Grid: The GrADS Experiment with ScaLAPACK, Univ. of Tennessee Technical Report UT-CS-01-460, 2001.

[13] J. Planck, M. Beck, W. Elwasif, T. Moore, M. Swany and R. Wolsky, IBP, The Internet Backplane Protocol: Storage in the Network, NetStore 99: Network Storage Symposium, Seattle, 1999.

[14] R. Ribler, J. Vetter, H. Simitci and D. Reed, Autopilot: adaptive control of distributed applications, Proc. of High Performance Distributed Computing Conference, 1998, pp. 172-179.

[15] Seti@home home page - URL: http://setiathome.ssl.berkeley.edu/

[16] S. Vadhiar and J. Dongarra, A performance oriented migration framework for the grid, 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003, pp. 130-137.

[17] S. Vadhiar and J. Dongarra, A meta scheduler for the grid, Proc. 11th IEEE Symposium on High Performance Distributed Computing, 2002, pp. 343-351.