Synchronization and Caching Data for Numerical Linear Algebra Algorithms in Distributed and Grid Computing Environments

Giuliano Laccetti
Department of Mathematics and Applications
University of Naples Federico II
via Cintia – Monte S.Angelo, 80126 Naples (Italy)
+39 081 675619
giuliano.laccetti@dma.unina.it

Marco Lapegna
Department of Mathematics and Applications
University of Naples Federico II
via Cintia – Monte S.Angelo, 80126 Naples (Italy)
+39 081 675623
marco.lapegna@unina.it

Valeria Mele
Department of Mathematics and Applications
University of Naples Federico II
via Cintia – Monte S.Angelo, 80126 Naples (Italy)
+39 081 675742
valeria.mele@unina.it

Diego Romano
Department of Mathematics and Applications
University of Naples Federico II
via Cintia – Monte S.Angelo, 80126 Naples (Italy)
+39 081 675742
diego.romano@dma.unina.it

ABSTRACT

Because of the dynamic and heterogeneous nature of a grid infrastructure, the client/server paradigm is a common programming model for these environments, where the client submits requests to several geographically remote servers for executing already deployed applications on its own data. According to this model, the applications are usually decomposed into independent tasks that are solved concurrently by the servers (the so called Data Grid applications). On the other hand, as many scientific applications are characterized by very large set of input data and dependencies among subproblems, avoiding unnecessary synchronizations and data transfer is a difficult task. This work addresses the problem of implementing a strategy for an efficient task scheduling and data management in case of data dependencies among subproblems in the same Linear Algebra application. For the purpose of the experiments, the NetSolve distributed computing environment has been used and some minor changes have been introduced to the underlying Distributed Storage Infrastructure in order to implement the proposed strategies.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming.*

General Terms: Algorithms, Performance.

DaGreS'09, May 18–20, 2009, Ischia, Italy.

Keywords

Synchronization, Data Caching, Grid Computing, Numerical Linear Algebra, Block Algorithms.

1. INTRODUCTION

A Grid infrastructure is built on the top of a collection of disparate and distributed resources (computers, databases, network, software, storage …) with functionalities greater than the simple sum of those addends [8]. The "added value" is a software architecture aimed to aggregate scattered computing and data resources to create a single computing system image. The "hardware" of this single computing system is often characterized by slow and non-dedicated Wide Area Networks (WAN) connecting very fast and powerful processing nodes (that can also represent supercomputers or large clusters) scattered on a huge geographical territory, whereas the "operating system" (the grid middleware) is responsible to find and allocate resources to the scientists' applications, taking into account the status of the whole grid. Many papers focus on this aspect of the grid computing, addressing issues such as resources brokering [e.g. 4,15], performance contract definition and monitoring [e.g. 11,13], and migration of the applications in case of contract violations [e.g. 10,14]. On the other hand, as common scientific applications are characterized by a very large set of input data and dependencies among subproblems, choosing the most powerful computational resources is not sufficient to achieve good performance, but it is essential also to define suitable methodologies to minimize synchronization among tasks, to distribute application data onto the grid components in order to overlap communication and computation and to provide tools that eliminate unnecessary data transfers. There is a small number of papers available in this research area [e.g. 3,7].

However, because a computational grid can be viewed as a single computational resource, it is possible to borrow ideas and methodologies utilized commonly for traditional systems and to adapt them to new environments. As case study, a block matrix multiplication algorithm has been considered, because it is a basic linear algebra computational kernel representative of similar other computational kernel, e.g. *LU*, LL^T and *QR* factorization required by several applications. On the other hand, it encompasses a lot of data movements and the task of minimizing the synchronization overhead among the nodes and using effective data caching strategies is challenging. Preliminary results of such activities are shown in [5].

2. DISTRIBUTED ALGORITHMS FOR MATRIX MULTIPLICATION

For sake of simplicity, assume that *A*, *B* and *C* are square matrices of order *n*, and divided in square blocks *C(I, J)* , *A(I,K)* and *B(K, J)* of order *r,* with *n* divisible by *r*, so that letting *NB* be the number of blocks in each dimension, it is *NB = n/r.*

Figure 1 then shows three versions obtained by the permutation of the loops' indices in a standard algorithm for the matrix multiplication $C = A \cdot B$

Note that other versions, obtained permuting the *I* and *J* indices, are equivalent to these ones and all the versions are based on the same matrix operation:

$$C(I,J) = C(I,J) + A(I,K)B(K,J) \qquad (1)$$

```
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
```

a: *(I,J,K) 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
```

b: *(I,K,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
```

c: *(K,I,J) ordering*

Figure 1: Three standard versions for the block multiplication algorithm

In a client/server implementation, for given values of *I*, *J* and *K*, this operation can be computed by the client sending to a server the three blocks *A(I,K), B(K,J)* and *C(I,J)*, so the server can update the block *C(I,J)* and send back the result to the client.

It is important to note that the only possible parallelism is always on the indices *I* and *J*, i.e. each block *C(I,J)* can be computed independently from the other ones. This is not possible on the index *K*, because of the risk of "race condition" on the access to the blocks *C(I,J)* for different values of *K*. As a consequence, in order to reduce the synchronization overhead accessing these blocks, it's essential to define in a client-server implementation which of the orderings in Figure 1 has to be used to compute the several matrix operations involving the blocks *C(I,J), A(I,K),* and *B(K,J)*.

With the *(I,J,K)* ordering (Figure 1.a) the client generates NB^2 independent threads of computation, each of them managing the sequence on the index *K*. With the *(I,K,J)* ordering (Figure 1.b) the client generates only *NB* independent threads of computation, each of them generating *NB* parallel tasks at every step of the index *K* . Finally, with the *(K,I,J)* ordering (Figure 1.c) at each step of the index *K*, the client generates NB^2 parallel tasks and these need to be completed in order to generate new ones.

For a computational cost analysis, let t_{ijk} denote the execution time (computation and communication) needed to perform the operation (1) and $T^{(a)}$, $T^{(b)}$, and $T^{(c)}$ denote respectively the total execution times for the three orderings in Figure 1. It is easy to find that:

$$T^{(a)} = \max_{i,j} \sum_{k} t_{ijk} \qquad T^{(b)} = \max_{i} \sum_{k} \max_{j} t_{ijk}$$
$$T^{(c)} = \sum_{k} \max_{i,j} t_{ijk} \qquad (2)$$

so that:

$$T^{(a)} \le T^{(b)} \le T^{(c)}$$

The *(I,J,K)* ordering is then more suitable to a distributed client/server implementation compared to the other two orderings. The least suitable one is the *(K, I, J)* ordering. The client/server implementation of the *(I,J,K)* ordering of the matrix multiplication algorithm is then:

```
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 server
      receive C(I,J) from server
    end for
  end for
endfor
```

Client algorithm

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

Server algorithm

Algorithm 1: The client-server implementation of the block matrix multiplication algorithm *(I,J,K)* form

Let us assume the ideal case, where the environment is homogeneous and dedicated to the computation. In this case the execution time $t_{ijk} = t$, is equal for all the values of *I, J, K,* and

$$T^{(a)} = T^{(b)} = T^{(c)} = NB \cdot t$$

This result shows a linear growth with *NB* for the total execution time, when the matrix dimension grows and the block dimension $r=n/NB$ is constant. When we multiply the matrix dimension n by α, the ideal value for the ratio S_α is then:

$$S_\alpha = T^{(a)}_{\alpha n} / T^{(a)}_n = (\alpha nt / r)/(nt / r) = \alpha \qquad (3)$$

S_α measures the ideal growth factor for $T^{(a)}$ when an α times larger problem is solved.

However, it is important to note that, in the client/server programming model, the data are stored in the client and are sent from there in chunks to the servers for computations; once the computation is completed, the results are returned to the client. The data movement from client and servers in a grid is similar to the data transfer between memories and processing unit in a single Non Uniform Memory Access (NUMA) machine.

Table 1: Typical values for bandwidth and latency

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

Fast and small memories are positioned at the higher level, whereas slower memories, that are usually accessed by means of geographic networks, are located at the lower ones. In this model the servers' secondary storage level can be either the disks of each server or an external (to the server) data repository, which is still close enough to make the access time to this level negligible compared to the access time from the client. Table 1 shows typical peak bandwidth and latency of 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.

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 [5,6]. This provides the processing elements with data taken from fast memories at the high level and avoids unnecessary data transfer toward the lowest levels (i.e. toward the client memory). As 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 grid assumes a key role in grid computing.

As an example, consider an application composed by three tasks with dependencies in the form of three pipelined stages, as shown in Figure 2. In this example, the output data from stage 1 represents the input data for stage 2, and the output data from stage 2 represents, in turn, the input data for stage 3.

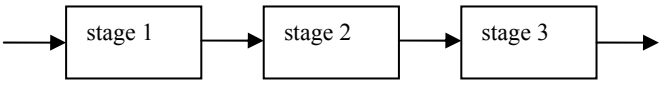

Figure 2: An application with three pipelined stages

A raw implementation of this application with the client/server programming model is depicted in Figure 3a where three servers compute the three stages of the application using the Algorithm 1. In this implementation the output data from stages 1 and 2 are sent back to the client and then sent again to a new server for the computation of the next stage. In this case the input data for stages 2 and 3 will be located at the lowest level of the memory hierarchy when accessed by the servers.

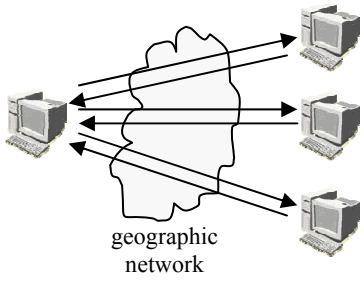

Figure 3a: Raw implementation of the application in Figure 2

In Figure 3b, the use of the server secondary storage as a cache for the intermediate results allows to locate them to a higher level in the memory hierarchy and avoids unnecessary data transfers toward the client memory. Furthermore, by keeping intermediate data in higher level memories it's possible to overlap data communication and stage computation if the entire sequence has to be repeated several times.

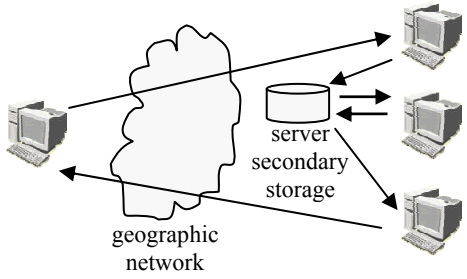

Figure 3b: Implementation with caching of the intermediate results

The following Algorithm 2 implements this strategy for the *(I,J,K)* ordering of the block matrix multiplication algorithm.

```
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
                    Client algorithm
```

```
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

                    Server algorithm
```

Algorithm 2: The client server *(I,J,K)* form of the block matrix multiplication algorithm with caching of intermediate results in the server secondary storage

For a computational cost analysis, let now *Ts* and *Tr* be respectively the access time to the server secondary storage and to the remote client memories, and $T^{(1)}$ and $T^{(2)}$ be respectively the total execution time for Algorithm 1 and Algorithm 2. The communication cost for the complete computation of each block *C(I,J)* with Algorithm 1 is then:

$$T^{(1)} = 4NB\ Tr\ r^2 .$$

Based on the values in Table 1 is $Tr = \gamma\ Ts$ with $10<\gamma<100$, the communication cost for the computation of each block *C(I,J)* by means of the Algorithm 2 is then :

$$T^{(2)} = 2NB\ r^2\ (\ Ts + Tr\ ) < T^{(1)}$$

A similar approach to the 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 section is the larger amount of space available to cache the intermediate data, with an access time to the cache still negligible compared to the client memory.

3. SOFTWARE TOOLS FOR CACHING DATA IN NETSOLVE

For our experiments the NetSolve 2.0 distributed computing infrastructure [1] has been used. 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. In this environment a key role is played by the agent, that gathers hardware performance and available software of the servers in the environment setup phase as well as dynamic information about the workload of the resources. When the agent is contacted by the client by means of the NetSolve client library linked to the user application, it selects the most suitable server to be used on the basis of the stored information and notifies the client. Therefore, the client can send data directly to the selected server that performs the computation by using a code generated through a Problem Description File that acts as interface between NetSolve and the deployed software. Finally the result is directly sent back to the client. This data exchange protocol is executed for every request to NetSolve, i.e. in case of dependency among multiple tasks in the same application, the execution appears as those depicted in Figure 3a, with an unnecessary network traffic.

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.

In order to fully implement a caching methodology as described in Figure 3b, it has been necessary to modify the NetSolve DSI implementation to some extent. More precisely, the DSI infrastructure defines the new data type DSI_OBJECT as a data structure describing the location and several information about the storage area that can be used as a cache. This remote storage area is managed by the Internet Backplane Protocol (IBP) infrastructure [2,12], a middleware for managing and using remote resources. In a typical NetSolve session, a DSI_OBJECT is generated by the client and sent to the servers by means of the NetSolve API, so that they can access the IBP storage (Figure 4)..

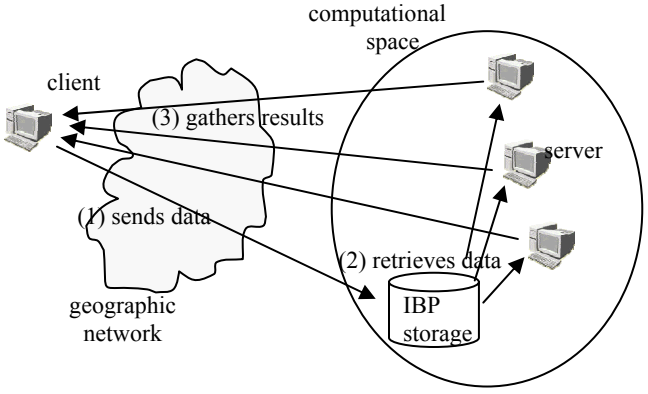

Figure 4: Implementation of the Distributed Storage Infrastructure in NetSolve

Among previous information in a DSI_OBJECT there are the read/write/management *capabilities* of the IBP infrastructure, i.e. unique character strings used as keys to correctly access the data on the storage. The main changes are therefore related to the DSI functions for reading and writing data on the IBP storage, so that

they can be used also by the servers. Actually, as of this writing, the servers cannot call directly these DSI functions because the Problem Description Files used to generate the server codes are unable to manage a DSI_OBJECT. For this reason, in the modified implementation of the DSI infrastructure, the APIs of the DSI functions for accessing the IBP storage include the capabilities of the IBP storage and are used in place of those related to the DSI_OBJECT.

4. COMPUTATIONAL EXPERIMENTS

A first set of experiments has been aimed to verify the effectiveness of the *(I,J,K)* ordering compared to the other orderings. For this purpose, the two orderings *(I,J,K)* and *(K,I,J)* (namely the best expected version and the worst expected version) have been implemented on a NetSolve infrastructures supplied by the University. of Tennessee, with the client located at the Department of Mathematics and Applications of the University of Naples. This software infrastructure can be called Wide Area system. The result $C=AB$ with square matrix of order $n=250, 500, 1000, 1500, 2000,$ and a fixed block size $r = 250$, thus $NB =1, 2, 4, 6, 8$ has been computed. In Figure 5 the execution times of the two orderings is reported. The better performance of the *(I,J,K)* ordering of the algorithm is evident with a smaller average execution time for each test. In order to minimize the impact of the traffic fluctuation in the network, the reported values are the average times over 10 runs

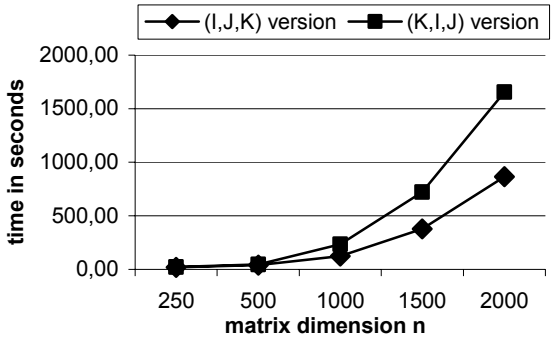

Figure 5. Average execution time of two orderings of the block matrix multiplication.

A second set of experiments is aimed to compare the execution times of the *(I,J,K)* ordering of the block matrix multiplication algorithm on two different NetSolve systems: a Wide Area Network system with the servers operated by the University of Tennessee and a Local Area Network system where all resources are connected by means of the Local Area Network of the Department of Mathematics and Applications of University of Naples at 100 Mbits. For these experiments the average execution time, across 10 runs, are reported in Figure 6. For the same values of n and r used in the previous experiments, it has been estimated that on the Local Area Network system the total execution time is about 50% smaller compared to the Wide Area Network system, because of the smaller latency and the higher bandwidth of the network. In order to quantify the performance gain, let us observe the value for S_2 achieved from $n=1000$ up to $n= 2000$: on the Local Area system $S_2 = 4.34$ it has been measured, whereas on the Wide Area system $S_2 = 7.14$. These values should be compared with the ideal value $S_2=2$ in order to give idea of the overhead introduced by the computational environment (both hardware and software contribution). More precisely, a factor grow for the execution time of about $O(NB^2)$ has been measured for Algorithm 2, compared to about $O(NB^3)$ for Algorithm 1 and to $O(NB)$ for the ideal case. Finally, note that efficiency or similar metrics are very few used parameters for performance analysis in distributed or grid computing, because the resource selection is in charge of the computational environment itself so that the user is unable to define the number of computing nodes, and, further, the primary goal for using these environments is the possibility to aggregate scattered and unused resources rather than a mere execution time reduction [9].

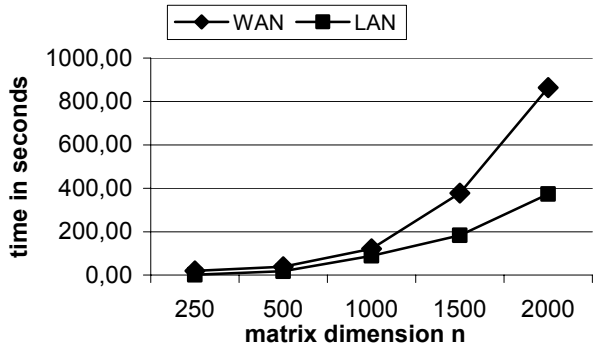

Figure 6. Average execution time for two NetSolve systems

A third set of experiments is aimed to test the software infrastructure needed to implement the data management policy described in Section 2. Some experiments have been carried out on a cluster of 2.4 GHz PCs, each of them provided with a Parallel ATA disk adapter with a peak transfer rate of 100 MByte/sec, and connected using a 1 Gbits switch. Algorithm 1 and Algorithm 2 have been implemented by using NetSolve-2.0 computing environment with the DSI infrastructure modified as described in Section 3.

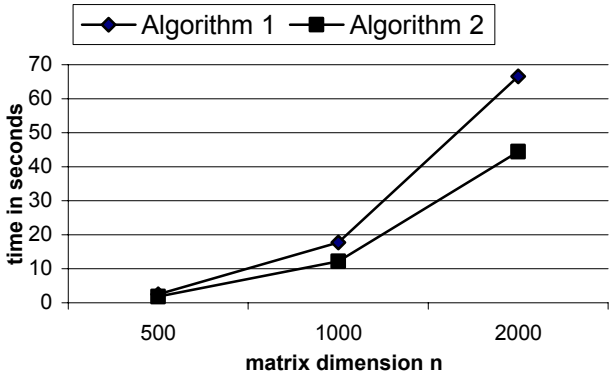

Figure 7. Average execution time for Algorithm1 and Algorithm 2

Figure 7 shows the average total execution times in seconds (across 10 executions) for Algorithm 1 (without data caching) and Algorithm 2 (with data caching) for matrices of order $n = 500, 1000$ and 2000 with square blocks of order $r = 500$. The results show a significant reduction of the total execution time for the computation of the entire matrix multiplication.

5. CONCLUSIONS

This work addresses the problem of implementing a strategy for task scheduling and data management in case of data dependencies among subproblems in the same application. The paper pursuits a double purpose. Firstly, it describes an effective methodology for task scheduling and for the placement of data among the resources of a distributed environment with the client/server programming model. Secondly, it shows how to modify to some extent the Distributed Software Infrastructure, part of the NetSolve distributed computing system, in order to implement the described methodology. The computational experiments confirm the expectations, showing a significant reduction of the execution times when the intermediate data are kept in the secondary storage of the servers. This activity is part of a larger project whose main purpose is to solve multidisciplinary applications, derived from researches conducted by scientists of Naples area, within a new powerful grid infrastructure to be integrated in large national and European grids.

6. ACKNOWLEDGMENTS

This work has been supported by Italian Ministry of Education, University and Research (MIUR) within the activities of the SCOPE project (PON Ricerca" 2000-2006 - Avviso 1575)

7. REFERENCES

[1] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, 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, G. Fagg - The Internet Backplane Protocol: A Study in Resource Sharing -Future Generation Computing Systems, Volume 19, Number 4, May, 2003, pp. 551-561.

[3] O.. Beaumont, V.Boudet, F. Rastello and Y. Robert. Matrix Multiplication on Heterogeneous Platforms. in IEEE Trans on Parallel and Distributed Systems, vol. 12 (2001), pp 1033-1051

[4] K. Czajkowsky, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke – A Resource Selection Management Architecture for Metacomputing Systems – in Proc. of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for parallel Processing, 1998.

[5] L. D'Amore, G.Laccetti M. Lapegna - Block Matrix Multiplication in a Distributed Computing Environment: Experiments with NetSolve – In: Wyrzykowski, R., Dongarra, J., Meyer, N., Wasniewski, J. (eds.). LNCS, vol. 3911, pp. 625–632. Springer, Heidelberg (2006)

[6] J. J. Dongarra, J. Du Croz, I. Duff, S. Hammarling - A proposal for a set of level 3 basic linear algebra subprograms - ACM SIGNUM Newsletter 22(3):2-14, 1

[7] J. Dongarra, J.F. Pineau, Y. Robert, Z. Shi, F. Vivien - Revisiting Matrix Product on Master-Worker Platforms - Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, March 2007 Page(s):1 -8

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

[9] G. Fox. Message Passing from Parallel Computing to the Grid. in IEEE Computing in Science and Engineering. Sept/Oct 2002

[10] 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 aGrid computing PSE - IFIP International Federation for Information Processing , Vol. 239: Grid-Based Problem Solving Environments: Implications for Development and Deployment of Numerical Software, Gaffney P.W.; Pool J.C.T., 2007

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

[12] J. Planck, M. Beck, W. Elwasif, T. Moore, M. Swany, R. Wolsky – IBP , The Internet Backplane Protocol: Storage in the Network. – in NetStore99: Network Storage Sympo-sium, Seattle, 1999.

[13] R. Ribler, J. Vetter, H. Simitci, D. Reed - Autopilot: Adaptive Control of Distributed Applications - Proc. of High Performance Distributed Computing Conference, 1998, pp. 172-179

[14] S. Vadhiar and J. Dongarra – A performance oriented migration framework for the grid - Proceedings of the 3st International Symposium on Cluster Computing and the Grid, 2003

[15] S. Vadhiar and J. Dongarra – A Meta Scheduler for the Grid – in Proc. 11th IEEE Symposium on High Performance Distributed Computing, July 2002