Adaptive Matrix Multiplication in Heterogeneous Environments*

Bo Hong and Viktor K. Prasanna
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
{bohong, prasanna}@usc.edu

Abstract

In this paper, an adaptive *matrix multiplication algorithm for dynamic heterogeneous environments is developed and evaluated. Unlike the state-of-the-art approaches, where load balancing is achieved through unequal distribution of the matrix data among the heterogeneous nodes, the matrices in our approach are partitioned into blocks of equal size. Task allocation and the block size are adapted during run time. Data pre-fetch is used to perform efficient communication. Our approach enables the use of various task scheduling heuristics. Further, we show that the control and coordination overheads of this approach are negligible when compared with the overall execution time. The effectiveness of the approach is verified through a configurable simulator developed for understanding the performance of heterogeneous computing environments.*

1 Introduction

Recently, distributed heterogeneous computing systems have become attractive platforms for high performance computing. In such systems, distributed resources such as workstations and supercomputers are connected through local and/or wide area networks. By utilizing these distributed resources in a coordinated manner, a heterogeneous computing system can meet the computational demands of complex applications [8].

However, effective use of these resources is still a challenge [6]. Computation and communication resources are typically shared among different users and applications. The network conditions and the effective compute power of each node vary at run time. This is particularly true in the case of the Grid [11]. Thus, in order to improve performance, the execution of applications should be *adaptive* to the changes in the system [2]. This introduces (1) *control overheads* because additional compute power must be spent to perform the adaptation, and (2) *coordination overheads* because additional communication costs must be spent to transfer control messages among the nodes. The efficiency of an algorithm lies not only in the adaptive computation scheme, but also in the minimization of these overheads. Further, in a distributed system, the performance of an application is also affected by the large communication latencies among various computing resources. In this paper, we develop and evaluate an efficient matrix multiplication algorithms in *dynamic* heterogeneous environments, where the performance of the resources vary at run time.

Recently, matrix multiplication for heterogeneous environments has received some attention. The approach in [4] partitions the source matrices into uneven rectangles, based on the compute power of each node, and optimizes the performance by searching for a partition that minimizes the overall amount of communication. However, this approach targets static environments, i.e. the performance of the resources does not vary at run time. Also, the nodes are assumed to be connected through a *homogeneous* network. The GrADS project [5], which focuses on the application development framework for the Grid environment, proposes a numerical linear algebra library for the Grid. This library is based on ScaLAPACK [14]. However, ScaLAPACK is designed for homogeneous distributed memory parallel computers. To the best of our knowledge, we are not aware of any effective techniques to perform matrix multiplication in a dynamic heterogeneous environment.

In this paper, we develop and evaluate an adaptive algorithm for matrix multiplication in a heterogeneous environment. The matrices are partitioned into blocks of equal size. The *basic task unit* of matrix multiplication is one such block. These tasks are adaptively assigned to the computing resources during run time, based on the performance of the resources and the progress of the computation. Data

*Supported by the DARPA Data Intensive Systems Program under contract F33615-99-1-1483 monitored by Wright Patterson Air force Base and in part by the National Science Foundation under award No. 99000613 and an equipment grant from Intel Corporation.

pre-fetch is used to reduce effectively the impact of communication cost. In order to utilize effectively the resources in the environment, both the task allocation and the size of the blocks are adapted during run time.

Our method provides a general framework to compute matrix multiplication in a dynamic heterogeneous environment. The framework coordinates the overall progress of the computation, while computation at each compute node can be performed by using highly optimized matrix multiplication subroutines (e.g. ATLAS [15]). The framework captures the dynamic behavior of the resources by collecting information about resource status and adaptively controls the computation during run time. Analysis shows that the control and coordination overheads are low compared with the overall execution time.

Besides the initial motivation from linear algebra, the proposed matrix multiplication problem reduces to the problem of scheduling independent tasks to heterogeneous resources. This problem, in its general form, has been shown to be NP-complete [12]. Hence various heuristics have been developed to schedule the tasks [7, 13]. Some of the heuristics perform static scheduling while others focus on dynamic scheduling. In this paper, our intent is not to develop new scheduling heuristics. Our proposed solution attacks the critical problems of hiding latency and minimizing control and coordination overheads. Our approach permits the use of various scheduling heuristics. As an illustrative example, we propose an $O(\log M)$ time heuristic, where M is the number of compute nodes, to adapt the task allocation.

A configurable simulator [9] is used to evaluate our matrix multiplication algorithm. The simulator provides a general simulation environment to describe the interactions among various resources in a heterogeneous computing environment. Simulation results show that our approach is robust to system heterogeneities and run-time performance variances of the resources. It shows improved performance compared with the approach in [4], in dynamic heterogeneous environments.

The rest of this paper is organized as follows: Section 2 describes our adaptive algorithm for matrix multiplication. Simulation results are presented in Section 3. Section 4 concludes the paper.

2 Adaptive Matrix Multiplication Algorithm

In this section, we first provide an overview of our approach. This is followed by a detailed discussion on several key issues in optimizing the performance of this algorithm: task allocation, data pre-fetch, block size selection, and control/coordination overheads. We also discuss techniques to minimize the communication. In this paper, we will use the words ‘scheduling’ and ‘allocation’ inter-changeably.

2.1 Algorithm Overview

We consider a heterogeneous computing environment that consists of a set of compute nodes. These compute nodes are connected through a set of network links. In this environment, both the compute power of the compute nodes and the bandwidth of the network links are heterogeneous and dynamic. We are interested in computing matrix multiplication $C = A \times B$ in such an environment. The matrices are of size $N \times N$. The objective is to minimize the overall execution time.

In our approach, there is a single data server, which stores the source matrices. Other nodes in the system are compute nodes that can be assigned particular tasks. The matrices are partitioned into blocks of the equal size $b \times b$ ($1 \leq b \leq N$). The *basic task unit* is to calculate a $b \times b$ block of matrix C. Each task unit will be executed by a chosen compute node, the required source data being provided by the data server. One of the nodes is designated as the coordinator, where the scheduling algorithm is executed. The following procedures briefly describe the functions for the coordinator, data server and compute nodes. Throughout this paper, we refer ‘allocation plan’ or ‘plan’ to the output of the task allocation algorithm.

```
Procedure_for_Coordinator {
   collect resource information
   generate an initial task allocation plan
   do {
     collect information on resource status
     collect information on computation progress
     if one row block of matrix C finishes computation {
        adapt block size b
     }
     adapt the task allocation plan
     assign tasks to compute nodes
   } until the matrix multiplication completes
}

Procedure_for_Data_Server {
  loop {
    accept and en-queue data request from the compute nodes
    de-queue and serve the requests
  }
}

Procedure_for_Compute_Node {
  loop {
    accept and en-queue tasks from the coordinator
    de-queue one task
    perform the following two functions in parallel {
      a. pre-fetch data from the data server for
         future tasks
      b. execute matrix multiplication for current task
    }
    report current resource status to the coordinator
  }
}
```

In our algorithm, there is no fixed scheme for data distribution or task allocation to the compute nodes, the compu-

tation is adapted to the dynamic behavior of the resources. Tasks are assigned to the compute nodes based on their run-time compute power and available bandwidth to the data server.

The impact of communication cost is reduced by using data pre-fetch. Data pre-fetch enables a compute node to perform computation and communication in parallel, therefore hiding (not always fully, though) the impact of communication costs. Block size b is a critical factor that affects the efficiency of data pre-fetch and the overall performance. The selection of b also determines the total number of tasks, which affects the parallelism of the computation. However, these two objectives (parallelism and the efficiency of pre-fetch) conflict with each other and trade-offs exist. We will present an efficient approach for selecting the optimal value of b.

We assume that matrices A, B, and C are of size $N \times N$. $b \times b$ is the block size. M denotes the number of compute nodes. N_j is the jth compute node. P_j denotes the compute power of node N_j. B_j denotes the available bandwidth between node N_j and the data server $(1 \leq j \leq M)$. $Q(N_j)$ denotes the number of tasks to be assigned to node N_j $(1 \leq j \leq M)$. For node N_j, $T_{comp}(N_j)$ denotes the computation time, $T_{comm}(N_j)$ is the data transfer time for one task, and $T_{exec}(N_j)$ is the expected execution time of one task. $T_{fin}(N_j)$ is the expected time that node N_j finishes all the assigned tasks. $T_{fin} = \max_{j=1}^{M} T_{fin}(N_j)$, denotes the expected overall execution time.

Network (start up) latencies are negligible when estimating the data transfer time for a task. For example, in a 100 Mb/s Ethernet, the latency is typically around 150 μs, while the transfer time for a 400 $\times$ 400 double precision matrix is 102 ms. Hence network latencies are only considered when estimating the transfer time of control messages. In our approach, the computation time and data transfer time for one task can be approximated as $T_{comp}(N_j) = 2Nb^2/P_j$ and $T_{comm}(N_j) = 2Nb/B_j$ $(1 \leq j \leq M)$. $T_{exec}(N_j)$ is predicted as $\max(T_{comp}(N_j), T_{comm}(N_j))$ if communication can be overlapped with the computation, otherwise $T_{exec}(N_j) = T_{comp}(N_j) + T_{comm}(N_j)$.

2.2 Task Allocation and Adaptation

In our algorithm, when a compute node completes its current task, this node reports its status to the coordinator. This is designated as a *control point*. At each control point, the coordinator adapts the task allocation.

Our implementation of the task allocation plan contains only the number of tasks to be assigned to each compute node, rather than the detailed mapping of the tasks to the compute nodes. This is because the matrices are partitioned into blocks of equal size. Every task requires the same amount of computation and communication to complete.

The task allocation plan is maintained using a binary tree B with M nodes. Each node in B has two fields $T_{fin}(N_j)$ (the key field) and $Q(N_j)$, $(1 \leq j \leq M)$. We also use an auxiliary heap H with M nodes. The key field of each node in H stores the values of $T_{fin}(N_j) + T_{exec}(N_j)$ (denoted as $T_1(N_j)$, $1 \leq j \leq M$).

At the beginning of the computation, an initial task allocation plan is calculated according to the minimum completion time [7] (MCT) heuristic. B and H are then initialized according to this initial plan. Each consecutive update to the task allocation plan is performed at the control points, where information of a compute node (e.g. the compute power P_k and available bandwidth to the data server B_k for node N_k) is updated. The update procedure is as follows:

```
1  Q(N_k) = Q(N_k) - 1
2  update T_fin(N_k) according to P_k and B_k
3  if T_fin(N_k) increases
4     do
5          select root node N_r in H
6          if T_1(N_r) < T_fin(N_k) then
7               Q(N_k) = Q(N_k) - 1
8               Q(N_r) = Q(N_r) + 1
9               update T_fin(N_k), T_fin(N_r) and B
10              update T_1(N_r), T_1(N_k), and H
11              update T_fin
12    until T_fin no longer decreases
13 else
14    do
15         find node N_m in B
               such that T_fin(N_m) = max_{j=1}^{M} T_fin(N_j)
16         if T_fin(N_k) + T_exec(N_k) < T_fin(N_m) then
17              Q(N_k) = Q(N_k) + 1
18              Q(N_m) = Q(N_m) - 1
19              update T_fin(N_k), T_fin(N_m) and B
20              update T_1(N_m), T_1(N_k), and H
21              update T_fin
22    until T_fin no longer decreases
```

In the above procedure, if $T_{fin}(N_k)$ increases, tasks are removed from node N_k and assigned to other nodes to reduce the overall execution time. Similar operations are applied when $T_{fin}(N_k)$ decreases. In the above procedure, lines 5, 7, 8, 11, 17, 18, and 21 take constant time; each of the lines 9, 10, 15, 19, and 20 takes $O(\log M)$ time. Both `do` loops contain at most a constant number of iterations (less than the number of tasks scheduled to a compute node). Hence the complexity of each update to the task allocation plan is $O(\log M)$.

2.3 Data Pre-Fetch

Data pre-fetch is used to overlap the communication and computation. According to our approximations of $T_{comm}(N_j)$ and $T_{comp}(N_j)$, $T_{comm}(N_j)/T_{comp}(N_j) = P_j/(B_j \cdot b)$. The extent to which communication and computation can be overlapped depends on the block size b,

the compute power P_j and the available bandwidth B_j to the data server. For example, if the data elements are double precision, $b = 100$, $B_j = 100\ Mb/s (= 1.56\ MData_element/s)$, and $P_j = 400\ MFlops$ (sustained performance of Pentium III 550 MHz for matrix multiplication, reported by the ATLAS project [15]), then $T_{comm}(N_j)/T_{comp}(N_j) = 1.28$, which means that $1/1.28 = 78\%$ of the communication time can be overlapped with the computation. Data pre-fetch in our algorithm is performed using a greedy method: for each compute node, if current data transfer finishes and there exist tasks waiting for source data, perform pre-fetch for the next such task. In Section 3, we will validate the efficiency of this data pre-fetch strategy.

2.4 Adaptation of Block Size

As shown by our the performance metric approximations, $T_{comm}(N_j)/T_{comp}(N_j) = P_j/(B_j \cdot b)$, and $T_{exec}(N_j) = \max(T_{comp}(N_j), T_{comm}(N_j))$ or $T_{exec}(N_j) = T_{comp}(N_j) + T_{comm}(N_j)$ $(1 \leq j \leq M)$. If b is small, data transfer time will dominate the execution time for the tasks; the compute node will spend time idling (waiting for the data transfer to complete) even if the computation and the communication can be overlapped. However, choosing a large b reduces the number of tasks and hence the parallelism of the computation. An extreme example is to set $b = N$. Although communication is most efficient for this choice, only one node can be chosen to perform the computation, wasting most computing resources.

Consider the case where P_i and B_i do not change over time, $P_i = P_j$ and $B_i = B_j$, for $1 \leq i, j \leq M$. Assume all compute nodes do data pre-fetch. The execution time of any task on any compute node is $T_{exec} = \max(T_{comp}(N_j), T_{comm}(N_j)) = \max(2Nb^2/P_j, 2Nb/B_j)$. There are a total of $(N/b)^2$ tasks. The optimal task allocation for this scenario is to assign each compute node $\lfloor (N/b)^2/M \rfloor$ tasks. For the remaining $(N/b)^2 - \lfloor (N/b)^2/M \rfloor$ tasks, randomly choose $(N/b)^2 - \lfloor (N/b)^2/M \rfloor$ compute nodes and assign one task to each of these selected nodes. Therefore, $T_{fin} = (\lfloor (N/b)^2/M \rfloor + 1) \cdot T_{exec} = \lceil (N/b)^2/M \rceil \cdot \max(2Nb^2/P_j, 2Nb/B_j)$. For a typical setting of $N = 10000$, $M = 64$, $P_j = 600\ MFLOPS$, $B_j = 100\ Mb/s$ $(1 \leq j \leq 64)$, the impact of b is shown in Fig. 1. Although this is a simple homogeneous and static scenario, it illustrates the performance degradation when b is too small or too large. We will demonstrate in Section 3 that in a dynamic heterogeneous environment, the impact of b on the overall execution time has similar variations; the optimal block size depends on the run time performance variation of the resources.

We search through the range $1 \leq b \leq N$ and choose the

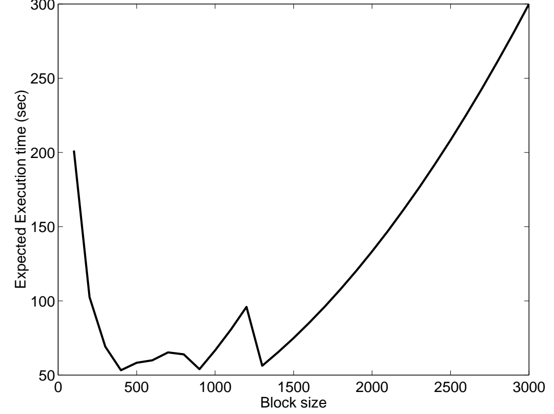

Figure 1. Impact of block size b

smallest block size that results in the minimum expected execution time. During the search process, the performance of a particular block size b may be evaluated by using either a simulator or the task allocation heuristic (T_{fin} is a direct output of the task allocation heuristic). The search process for the optimal block size may be computation intensive as it needs to evaluate the performance of multiple block sizes. However the search can be executed by the coordinator as a background job, with a priority lower than the adaptation of the task allocation plan. Note that the procedure in Section 2.1 needs some minor changes if block size selection is executed in the background. In our approach, b is adapted only after the execution completes for one row block of matrix C.

2.5 Control and Coordination Overheads

In this section, we show that the control and coordination overheads of our algorithm are negligible compared with the overall computation time.

Control overheads: In [7], it is shown that the complexity of MCT is $O(M)$. For a problem with 512 tasks and 16 compute nodes, it takes less than 1 μs on a 400MHz Pentium II. Even though our algorithm needs to update the allocation plan multiple times, the overheads of the update is negligible. For example, for problem sizes 2000×2000 to 6000×6000, there are $10^2 \sim 10^3$ allocation plan updates (as shown by our simulations); Even each update takes 1 μs, the total execution time for adapting the allocation plan is less than 1 ms. This is much smaller than the $10 \sim 100$ second execution time of the matrix multiplication (as shown by our simulations and also reported in [4]).

Coordination overheads: Control messages are transferred in the system when (1) a task is assigned to a compute node, (2) a data transfer request is sent to a data server, or (3) the compute node reports its status to the coordinator upon

the completion of a task. The control messages for these cases consist of only a small amount of information (node ID and matrix block index (i, j) for cases 1 and 2; node ID, compute power, and network bandwidth for case 3). Suppose each control message consists of 16 bytes. Note that there are three messages for each block and the total number of blocks is N^2/b^2, the total amount of control message transfer is $A_{ctrl} = 3 * 16N^2/b^2$ bytes. At the same time, the total amount of data transfer is $A_{data} = (2N^3/b) \times 8$ bytes (assuming each data element is 8 bytes). The cost of transferring control messages is negligible when the matrix size becomes large.

2.6 Minimizing Communication and Communication Bottlenecks

Note that during the computation, a compute node caches the ith row block of matrix A and the jth column block of matrix B, if this node is assigned the task of computing $C(i, j)$. If future tasks assigned to this node is to compute blocks in the same row(column), then this node can utilize the locally cached row(column) block of matrix $A(B)$. This optimization reduces the total amount of communication, rather than hiding the communication cost. Also, a compute node can act as a data server, providing the cached data to other compute nodes. The impact of these optimizations can be found in [10].

3 Performance Evaluation

We use a modular and extensible simulator [9] to evaluate our matrix multiplication algorithm. This simulator is capable of describing the interactions among various resources in a dynamic heterogeneous computing environment. The simulator consists of three classes of entities: scheduler, global information directory, and computing and network resources. These entities interact with each other by sending and receiving messages. The simulator has built-in support for evaluating the performance of adaptive applications in dynamic heterogeneous environments. It models the interactions between the applications and the resources through a closed adaptation loop, in which the computation is adapted according to the run-time performance variations of the resources.

Various components of the simulator are configurable by the user, such as the scheduling algorithm to be used, the dynamic behavior of the resources, the time interval between the adaptations, the method for partitioning the problem, etc. This flexibility in configuration makes it possible to simulate various applications (matrix multiplication in this study) and various dynamic heterogeneous computing environments. The simulator provides a set of commands to collect the key performance statistics (e.g. the start and finish time of each task and the overall execution time of an application) of a simulation. The simulator can also be used to analyze the activities of entities in a simulation, trace the idle time of the compute nodes, and determine possible performance bottlenecks.

3.1 Simulation Results

The simulations are arranged into four sets. The first set tests the efficiency of the greedy data pre-fetch strategy. Second, the impact of control and coordination overheads is presented. The third set of simulations illustrates the impact of block size on the execution time. The last set compares our approach with a state-of-the-art technique proposed in [4]. More simulation results can be found in the technical report version of this paper [10].

In the simulations, P_i and B_i $(1 \leq i \leq M)$ are modeled as normally distributed random variables with mean values $P(i)$ and $B(i)$, and standard deviations $\sigma_p(i)$ and $\sigma_b(i)$. As discussed earlier, network latencies are considered only when estimating the transfer time for control messages. Let $\overline{P} = \sum_{i=1}^{M} P(i)/M$, $\overline{B} = \sum_{i=1}^{M} B(i)/M$, $\sigma P = \sqrt{\sum_{i=1}^{M}(P(i) - \overline{P})^2/(M-1)}$, and $\sigma B = \sqrt{\sum_{i=1}^{M}(B(i) - \overline{B})^2/(M-1)}$. σP and σB represent the system heterogeneities in compute power and network bandwidth. Following a similar classification as in [3], the systems are classified into four categories based on the values of σP and σB: high σP and high σB, high σP and low σB, low σP and high σB, low σP and low σB. These are abbreviated as 'HH', 'HL', 'LH' and 'LL' respectively. For *static* environments, there is no run-time performance variation, $\sigma_p(i) = 0$ and $\sigma_b(i) = 0$; in a *dynamic* environment, we assume that $\sigma_p(i) = 0.15P(i)$ and $\sigma_b(i) = 0.12B(i)$. The heterogeneities of the simulated systems are described in Table 1. For dynamic environments, each simulation was repeated multiple times until a 95% confidence interval with a 10% precision was obtained, and the reported simulated execution time is the average over these simulations.

3.1.1 Summary of Results

We first tested the performance of our greedy data pre-fetch strategy. Two 20-node dynamic HL systems were simulated. System 1 was connected using a 100 Mb/s network and System 2 was connected using a 10 Mb/s network. The two systems had the same $\overline{P}$ (200 $MFLOPS$) and σp (30 $MFLOPS$). We compared our data pre-fetch strategy with non pre-fetch for problem size 6000 $\times$ 6000 and 8000 $\times$ 8000. The idle time of each individual node was recorded using the simulator. The results in Table 2 show

Table 1. Heterogeneities of the simulated systems. $\overline{P}$ and σP are in MFLOPS. $\overline{B}$ and σB are in Mb/s

Systems	$\overline{P}$	σP	$\overline{B}$	σB
9-node HL	440	188	1000	0
9-node HH	440	188	1000	290
50-node HL	210	90	100	0
64-node HH(LAN)	400	84	100	29
64-node HH(WAN)	400	84	1.5	0.32

Table 2. Average idle time of the compute nodes (percentage of the overall execution time)

Method	Pre-fetch		No Pre-fetch	
Problem size	6000	8000	6000	8000
System 1	22.1%	14.8%	36.7%	30.7%
System 2	72.2%	67.8%	76.8%	70.5%

the average idle time of the compute nodes. The simple data pre-fetch strategy used in our approach reduces the average idle time from 30.7% to 14.8% for System 1. The performance improvement in a lower speed network environment (System 2) is smaller. This is because the data transfer time for a task is much larger than the execution time of a task in such a system, hence data pre-fetch becomes less effective.

The second set of simulations studies the impact of control and coordination overheads. These simulations were based on the dynamic 64-node HH LAN and WAN systems in Table 1. The results are shown in Fig. 2. The network latencies for the LAN and WAN environments were actual test results of MPICH-G2 reported in [1]. MPICH-G2 is a well-known programing interface for the Grid environment.

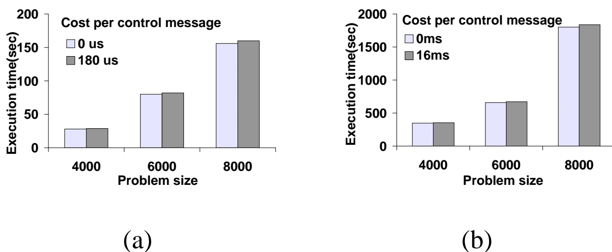

Figure 2. Impact of coordination overheads (a) LAN environment. Network latency $160\mu s$ (b) WAN environment. Network latency $15ms$

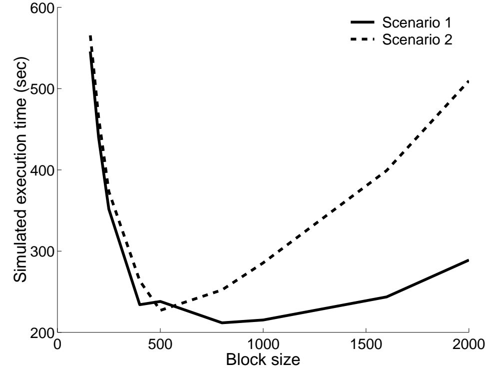

Figure 3. Impact of block size. Problem size is 8000×8000.

100 Mb/s is the typical bandwidth for Ethernet bound environments, while 1.5 Mb/s is a reasonable estimation for WAN environments. Given these parameters, the cost of transferring one message is estimated as 180 μs in a LAN, and 16 ms in a WAN. As can be seen from the results, only a small portion of the overall execution time was spent transferring the control messages. For example, with 15 ms per control message transfer in the WAN environment, the overall execution time for problem size of 8000×8000 increases only 3% when compared with the zero cost per control message scenario. Our simulations showed that the impact of control overheads was negligible, less than 1% for all the cases considered in this paper. This is because the 1 μsec execution time per call of the task allocation heuristic is negligible compared with the $10 \sim 200 sec$ overall computation time. Due to space limitation, the results are omitted here.

The third set of results was extracted from the simulation of a 50-node dynamic HL system shown in Table 1. Block size adaptation occurred multiple times during the simulation. In Fig. 3, Scenario 1 and 2 show two instances of these block size adaptations. For each scenario, we evaluated the performance of possible block sizes using our simulator. In Fig. 3, the impact of block size shows the same trend as analyzed in Section 2.4. Performance is degraded if b is too small or too large. In Scenario 2, some nodes lost 40% of their available compute power, compared with Scenario 1. This performance variation changed the optimal block size from 800 to 500. If the block size had not been adapted between these two scenarios, the execution time would increase by 11.5%. This illustrates the importance of run-time block size adaptation in a dynamic environment.

Comparison with the static matrix multiplication [4](denoted as SHMM, Static Heterogeneous Matrix Multiplication) was performed on several 9-node systems. A similar system connected by a Myrinet network was tested in

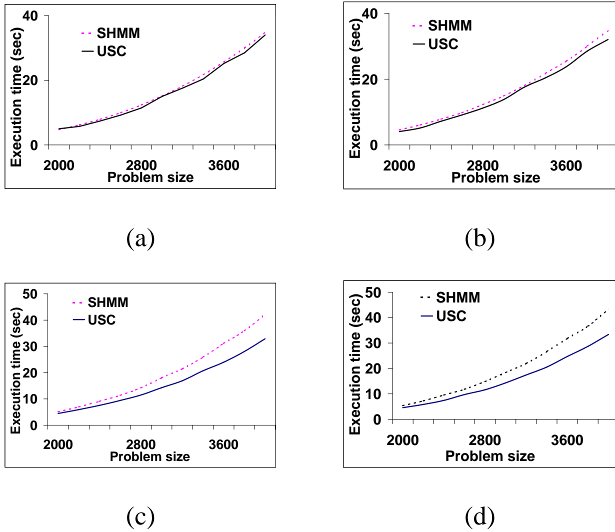

Figure 4. Comparison with the method in [4]. (a) a static HL system, (b) a static HH system, (c) a dynamic HL system, (d) a dynamic HH system. (SHMM represents the method in [4], USC represents our approach)

[4]. $B(i) = 1000\ Mb/s, 1 \leq i \leq 9$, was used as the bandwidth of each link. Note that in these results, the reported execution time of the SHMM method does not take into account the cost of initial data distribution. In our method, no initial data distribution is needed because the data distribution is automatically performed at run time by task allocation.

For a fair comparison of our approach with that in [4], which targets a static heterogeneous environment with a homogeneous network, we first conducted simulations on a static 9-node HL system with a homogeneous network ($\sigma B = 0$). The two methods showed similar performance (See Fig. 4(a)). Fig. 4(b) shows the results on a static 9-node HH system ($\sigma B = 290$). Since our approach considers heterogeneity in both compute power and network bandwidth, it showed an improved performance of up to 11% over SHMM. Figs. 4(c) and 4(d) show the results on the above two systems (9-node HL and HH), when run-time variations exist. Our approach is up to 22% faster than SHMM. Besides modeling the performance of the resources as normally distributed random variables, we also evaluated the performance in a scenario where the performance of some resources drops severely. A system similar to the 9-node HH system in Table 1 was simulated. During the simulations, the performance of the most powerful node was altered such that it loses 75% of its compute power immediately after the computation begins. The results are

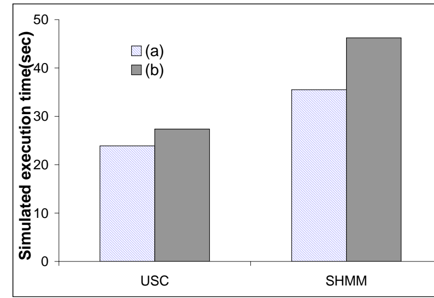

Figure 5. Impact of severe performance variation of the resources. Problem size is 3600×3600. (a) no severe performance variation. (b) scenario in which the most powerful node loses 75% compute power immediately after the computation begins.

shown in Fig. 5. Our approach is adaptive to the changes of the environment, the simulated execution time increased by 14.5%, while a 30.2% increase was observed in the SHMM method.

4 Conclusion

This paper proposed an approach for matrix multiplication in a heterogeneous environment in which the performance of the resources vary at run time. Task assignments are determined at run time based on the performance of the resources in the system. Our approach is adaptive to the changes in both the compute power and the network bandwidth. Simulation results show that our approach provides improved performance compared with the state-of-the-art matrix multiplication algorithms, when the environment is dynamic and heterogeneous. However, the use of our approach in a static heterogeneous environment needs further investigation. Although our approach avoids the cost of initial data distribution, the adaptation becomes unnecessary and the overheads incurred during the adaptation may degrade the overall performance.

Although the effectiveness of our algorithm in dynamic heterogeneous environments has been verified through simulations, several issues need to be addressed. We have observed that a single data server may cause bottlenecks. We are exploring two possible solutions: (1) using multiple data servers and (2) utilizing the cached data on the compute nodes. We are also exploring the relationship between the adaptation intervals and the overall performance. In this paper, we evaluated event-driven adaptation. Another alternative is to perform the adaptations periodically. We are also

investigating the effectiveness of various scheduling algorithms and studying other applications such as LU decomposition and transitive closure.

References

[1] R. Alfieri and F. Spataro. Test con MPIch-G2. http://www.fis.unipr.it/lca/grid/doc/mpichg2.pdf.

[2] A. Alhusaini, C. S. Raghavendra, and V. K. Prasanna. Run-Time Adaptation for Grid Environments. *Heterogeneous Computing Workshop, 15th International Parallel and Distributed Processing Symposium (IPDPS '01)*, 2001.

[3] S. Ali, H. J. Siegel, M. Maheswaran, and S. Ali. Task Execution Time Modeling for Heterogeneous Computing Systems. *9th Heterogeneous Computing Workshop*, 2000.

[4] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix-matrix Multiplication on Heterogeneous Platforms. *International Conference on Parallel Processing (ICPP 2000)*, 2000.

[5] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of Supercomputer Applications*, 15(4), 2001.

[6] P. Bhat, V. K. Prasanna, and C. S. Raghavendra. Efficient Collective Communication in Distributed Heterogeneous Systems. *19th IEEE International Conference on Distributed Computing Systems*, 1999.

[7] T. D. Braun, H. J. Siegel, and N. Beck. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.

[8] R. F. Freund and H. J. Siegel. Heterogeneous Processing. *IEEE Computer*, 26:13–17, 1993.

[9] B. Hong and V. K. Prasanna. A Modular and Extensible Simulator for Performance Evaluation of Adaptive Applications in Heterogeneous Computing Environments. *5th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2002)*, October 2002.

[10] B. Hong and V. K. Prasanna. Adaptive Matrix Multiplication in Heterogeneous Environments. Technical Report, University of Southern California, in preparation, 2002.

[11] I. Foster and C. Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[12] O. Ibarra and C. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM*, 24(2):280–289, 1977.

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[14] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G.Fagg, K. Roche, and S. Vadhiyar. Numerical Libraries And The Grid: The GrADS Experiments With ScaLAPACK. *International Journal of Supercomputer Applications*, 15(4), 2001.

[15] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). *Proceedings of SC '98*, November 1998.