A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems

Muthucumaru Maheswaran and Howard Jay Siegel

Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285 USA
{maheswar, hj}@ecn.purdue.edu

Abstract

A heterogeneous computing system provides a variety of different machines, orchestrated to perform an application whose subtasks have diverse execution requirements. The subtasks must be assigned to machines (matching) and ordered for execution (scheduling) such that the overall application execution time is minimized. A new dynamic mapping (matching and scheduling) heuristic called the hybrid remapper is presented here. The hybrid remapper is based on a centralized policy and improves a statically obtained initial matching and scheduling by remapping to reduce the overall execution time. The remapping is non-preemptive and the execution of the hybrid remapper can be overlapped with the execution of the subtasks. During application execution, the hybrid remapper uses run-time values for the subtask completion times and machine availability times whenever possible. Therefore, the hybrid remapper bases its decisions on a mixture of run-time and expected values. The potential of the hybrid remapper to improve the performance of initial static mappings is demonstrated using simulation studies.

Keywords: dynamic scheduling, heterogeneous computing, list scheduling, mapping, matching, parallel processing, scheduling.

1. Introduction

Different portions of a computationally intensive application often require different types of computations. In general, a given machine architecture with its associated compiler, operating system, and programming environment does not satisfy the computational requirements of all portions of an application equally well. However, a heterogeneous computing (HC) environment that consists of a heterogeneous suite of machines and high-speed interconnections provides a variety of architectural capabilities, which can be orchestrated to perform an application that has diverse computational requirements [2, 10, 14, 15]. The performance criterion for HC used in this paper is to minimize the completion time, i.e., the overall execution time of the application on the machine suite.

One way to exploit an HC environment is to decompose an application task into subtasks, where each subtask is computationally well suited to a single machine architecture. Different subtasks may be best suited for different machines. The subtasks may have data dependencies among them, which could result in the need for inter-machine communications. Once the subtasks are obtained, each subtask is assigned to a machine (matching). The subtasks and inter-machine data transfers are ordered (scheduling) such that the overall completion time of the application is minimized. It is well known that such a matching and scheduling (mapping) problem is, in general, NP-complete [3]. Therefore, many heuristics have been developed to

obtain near-optimal solutions to the mapping problem. The heuristics can be either static (matching and scheduling decisions are made prior to application execution) or dynamic (matching and scheduling decisions are made during application execution).

Most static mapping heuristics assume that accurate estimates are available for (a) subtask computation times on various machines and (b) inter-machine data transfer times. Often, it is difficult to accurately estimate the above parameters prior to application execution. Therefore, this paper proposes a new dynamic algorithm, called the hybrid remapper, for improving the initial static matching and scheduling. The hybrid remapper uses the run-time values that become available for subtask completion times and machine availabilities during application execution time. It is called the hybrid remapper because it uses some results based on an initial static mapping in conjunction with information available only at execution time.

The hybrid remapper heuristics presented here are based on the list scheduling class of algorithms (e.g., [1, 9]). An initial, statically obtained mapping is provided as input to the hybrid remapper. If the initial mapping is not provided, it should be obtained before running the hybrid remapper by executing a static mapping algorithm such as the baseline [18], genetic-algorithm-based mapper [18], or Levelized Min Time [9].

The hybrid remapper executes in two phases. The first phase of the hybrid remapper is executed prior to application execution. The set of subtasks is partitioned into blocks such that the subtasks in a block do not have any data dependencies among them. However, the order among the blocks is determined by the data dependencies that are present among the subtasks of the entire application. The second phase of the hybrid remapper, executed during application run time, involves remapping the subtasks. The remapping of a subtask is performed in an overlapped fashion with the execution of other subtasks. As the execution of the application proceeds, run-time values for some subtask completion times and machine availability times can be obtained. The hybrid remapper attempts to improve the initial matching and scheduling by using the run-time information that becomes available during application execution and the information that was obtained prior to the execution of the application. Thus, hybrid remapper's decisions are based on a mixture of run-time and expected values.

This research is part of a DARPA/ITO Quorum Program project called MSHN (Management System for Heterogeneous Networks). MSHN is a collaborative research effort that includes NPS (Naval Postgraduate School), NRaD (a Naval Laboratory), Purdue, and USC (University of Southern California). It builds on Smart-Net, an operational scheduling framework and system for managing resources in a heterogeneous environment developed at NRaD [6]. The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver the requested qualities of service.

The organization of this paper is as follows. The matching and scheduling problem and the associated assumptions are defined in Section 2. Three variants of the hybrid remapper heuristics are described in Section 3. Section 4 examines the data obtained from the simulation studies conducted to evaluate the performance of the hybrid remapper heuristic. In Section 5, related work is discussed. Finally, Section 6 gives some future research directions.

2. Problem Definition

The following assumptions are made regarding the application. The application is decomposed into multiple subtasks and the data dependencies among them are known and are represented by a directed acyclic graph (DAG). That is, the nodes in the DAG represent the subtasks and the links represent the data dependencies. An estimate of the expected computation time of each subtask on each machine in the HC suite is known a priori. This assumption is typically made when conducting mapping research (e.g., [4, 7, 13, 16]). Finding the expected computation time is another research problem. Approaches based on analytical benchmarking and task profiling are surveyed in [14, 15]. Any loops and data conditionals are assumed to be contained inside a subtask.

It is assumed that the hybrid remapper is running on a dedicated workstation and all mapping decisions are centralized. Once a subtask is mapped onto a machine it is inserted into a local job queue on that particular machine. The execution of the subtask is managed by the job control environment of the local machine. The subtask executions are non-preemptive. All input data items of a subtask must be received before its execution can begin, and none of its output data items are available until its execution is completed. These assumptions make the matching and scheduling problem in HC systems more manageable. Nevertheless, solving the mapping problem with these assumptions is a significant step toward solving the more general problem.

An application task is decomposed into a set of subtasks S, where s_i is the i-th subtask. Let the HC environment consist of a set of machines M, where m_j be the j-th machine. The estimated expected computation time of subtask s_i on machine m_j is given by $e_{i,j}$. The earliest time at which machine m_j is available is given by $A[j]$, where $|A| = |M|$.

The data communication time between two machines has two components: a fixed message latency for the first byte to arrive and a per byte message transfer time. An $|M| \times |M|$ communication matrix is used to hold these values for the HC suite. Similar matrices are used by other researchers in HC (e.g., [7, 13, 16]).

To facilitate the discussion in Section 3, a hypothetical node called an exit node is defined for the DAG as follows. An exit node (subtask) is a node with 0 computation time that is appended to the DAG such that there is a 0 data transfer time communication link to this node from every node in the DAG that does not have an output edge. The critical path for a node in the DAG is defined as the longest path from the given node to the exit node.

3. The Hybrid Remapper Algorithm

3.1. Overview

The notion behind most dynamic mapping algorithms is that due to the dynamic nature of the mapping problem, it is not efficient to use a fixed mapping computed statically. Therefore, most dynamic mappers regularly either generate the mapping or refine an existing mapping at various times during task execution. That is, dynamic mapping algorithms solve the mapping problem by solving a series of partial mapping problems (consisting of only a subset of the original set of subtasks). The partial mapping problem is usually solved by a static mapping heuristic. Because the mapping is performed in real time, it is necessary to use a fast algorithm to avoid any machine idle times that occur from having to wait for the mapper to complete its execution. In the hybrid remapper algorithm presented here, the partial mapping problem is solved using a list-based scheduling algorithm.

In the following subsections, three variants of the hybrid remapper algorithm are described. The first phase, common for all three variants of the hybrid remapper, involves partitioning the subtasks into blocks and assigning ranks to each subtask (where the rank indicates the subtask's priority for being mapped, as defined below). The variants of the hybrid remapper differ in the second phase by the minimization criteria they use and by the way they order the subtasks examined by the partial mapping problem. One variant of the hybrid remapper attempts to minimize the expected partial completion time at each remapping step, and the others attempt to minimize the overall expected completion time. Two variants of the hybrid remapper order the subtasks at each remapping step using ranks computed at compile time, and the other using a parameter computed at run time.

3.2. Partitioning and Rank Assignment

This first phase uses the initial static mapping, expected subtask computation times, and expected data transfer times to preprocess the DAG that represents the application. Initially, the DAG is partitioned into $\underline{B}$ blocks numbered consecutively from 0 to $B-1$. The partitioning is done such that the subtasks within a block are independent, i.e., there are no data dependencies among the subtasks in a block. All subtasks that send data to a subtask s_j in block k must be in any of blocks 0 to $k-1$. Furthermore, for each subtask s_j in block k there exists at least one incident edge (data dependency) such that the source subtask is in block $k-1$, i.e., an incident edge from some s_i. The $(B-1)$-th block includes the subtasks without any successors and the 0-th block includes only those subtasks without any predecessors. The exit node is not included in any block in the DAG partitioning. The three blocks obtained using this partitioning algorithm for an example seven node DAG is shown in Figure 1(a).

Once the subtasks in the DAG are partitioned, each subtask is assigned a rank by examining the subtasks from block $B-1$ to block 0. The rank of each subtask in the $(B-1)$-th block is set to its expected computation time on the machine to which it was assigned by the initial static matching. Now consider the k-th block, $0 \le k < B-1$. Recall $e_{i,x}$ is the expected computation time of the subtask s_i on machine m_x. Let $c_{i,j}$ be the data transfer time for a descendent s_j of s_i to get all the relevant data items from s_i. The value of $c_{i,j}$ will be dependent on the machines assigned to subtasks s_i and s_j by the initial mapping, and the information in the communication matrix. Let $\text{iss}(s_i)$ be the immediate successor set of subtask s_i such that there is an arc from s_i to each member of $\text{iss}(s_i)$ in the DAG. In the equation below, each $e_{i,x}$ implies subtask s_i is assigned to machine m_x by the initial mapping. With these definitions, the rank of a subtask s_i is given by:

$$\text{rank}(s_i) = e_{i,x} + \max_{s_j \in \text{iss}(s_i)} (c_{i,j} + \text{rank}(s_j))$$

Figure 1(b) illustrates the rank assignment process for the subtask s_i. The rank of a subtask can be interpreted as the length of the critical path from the point the given subtask is located on the DAG to the exit node, i.e., the time until the end of the execution of all its descendents. Two variants of the hybrid remapper described here are based on the heuristic idea that by executing the subtasks with higher ranks as quickly as possible, the overall expected completion time for the application can
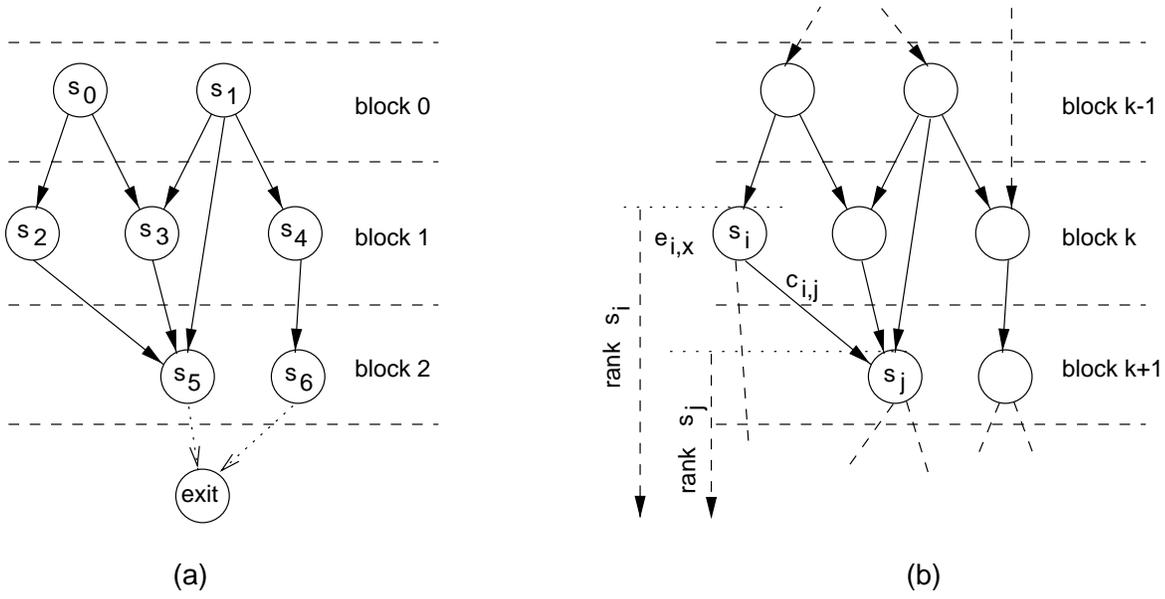
Figure 1: (a) Partitioning a DAG into blocks and (b) assigning ranks to the nodes of a DAG.

be minimized.

3.3. Common Portion of the Run-Time Phase

In all three variants of the hybrid remapper, the execution of the subtasks proceeds from block 0 to block $B-1$. A block k is considered to be executing if at least one subtask from block k is running. Also, the execution of several blocks can overlap with each other in time, i.e., subtasks from different blocks could be running at the same time.

The hybrid remapper changes the matching and scheduling of the subtasks in block k while the subtasks in block $(k-1)$ or before are being executed. The hybrid remapper starts examining the block k subtasks when the first block $(k-1)$ subtask begins its execution. When block k subtasks are being mapped, it is highly likely that run-time completion time information can be used for many subtasks from blocks 0 to $k-2$. There may be some subtasks from blocks 0 to $k-2$ that are still running or waiting execution when subtasks from block k are being considered for remapping. For such subtasks, expected completion times are used.

3.4. Minimum Partial Completion Time Static Priority (PS) Algorithm

As mentioned earlier, the hybrid remapper uses a list-scheduling type of algorithm to recompute the matching and scheduling for the subtasks in each block. In a list-scheduling type of algorithm, the subtasks are first ordered based on some priority. Then, each subtask is mapped to a machine by examining the list of subtasks from the highest priority subtask to the lowest priority subtask. The machine to which each subtask is assigned depends on the matching criterion used by the particular algorithm.

In this variant of the hybrid remapper, the priority of a subtask is equal to the rank of that subtask that was computed statically in the first phase (Subsection 3.2). The matching criterion used for subtask s_i is the minimization of the partial completion time, defined below. Thus, this variation is referred to as the minimum partial completion time static priority (PS) algorithm.

Let m_x be the machine on which s_i is being considered for execution. Then let $\text{pct}(s_i,x)$ denote the partial completion time of the subtask s_i on machine m_x, $\text{dr}(s_i)$ be the time at which the last data item required by s_i to begin its execution arrives at m_x, and $\text{ips}(s_i)$ be the immediate predecessor set for subtask s_i such that there is an arc to s_i from each member of $\text{ips}(s_i)$ in the DAG. For any subtask s_i in block 0, $\text{pct}(s_i,x) = e_{i,x}$. For any subtask s_i not in block 0, where $s_j \in \text{ips}(s_i)$, and s_j is currently mapped onto machine m_y,

$$\text{dr}(s_i) = \max_{s_j \in \text{ips}(s_i)}(c_{j,i}+\text{pct}(s_j,y))$$

$$\text{pct}(s_i,x) = e_{i,x}+\max(A[x], dr(s_i))$$

In the computation of $\text{pct}(s_i,x)$, the above equation is recursively used until subtask s_j is such that its run-time

completion time on machine m_y is available or subtask s_j is in block 0. The subtask s_i is remapped onto the machine m_x that gives the minimum pct(s_i,x), and $A[x]$ is updated using pct(s_i,x). Then the next subtask from the list is considered for remapping.

3.5. Minimum Completion Time Static Priority (CS) Algorithm

The notion behind the PS algorithm was that by remapping the highest rank subtask s_i to execute on the machine that will result in the smallest expected partial completion time, the overall completion time of the application may be minimized. Instead of this approach, the variant of the hybrid remapper described here attempts to minimize the overall completion time by remapping each subtask s_i in block k such that the length of the critical path through subtask s_i is reduced. Thus, this variation is referred to as the minimum completion time static priority (CS) algorithm. The reason for considering both PS and CS is that in PS the remapping is faster but CS attempts to derive a better mapping because it considers the whole critical path through s_i.

Let m_x be the machine on which s_i is being considered for execution. Then let the longest completion time path from a block 0 subtask to the exit node through the subtask s_i be ct(s_i,x). The overall completion time of the application task is determined by one such longest path through a block k subtask. Consider the subtask s_i in Figure 2. Assume that the longest path through s_i is shown by bold edges in Figure 2. For any subtask s_i,

$$\text{ct}(s_i,x) = \max_{s_j \in \text{iss}(s_i)}(\text{pct}(s_i,x)+c_{i,j}+\text{rank}(s_j))$$

$$= \text{pct}(s_i,x)+ \max_{s_j \in \text{iss}(s_i)}(c_{i,j}+\text{rank}(s_j))$$

The subtask s_i is remapped onto the machine m_x that gives the minimum ct(s_i,x), and $A[x]$ is updated using pct(s_i,x). Then the next subtask in the list is considered for remapping.

3.6. Minimum Completion Time Dynamic Priority (CD) Algorithm

The rank of a subtask s_i is computed prior to application execution. Therefore, if s_i is remapped to a machine other than the one it was assigned to by the initial static mapping, the rank of s_i may not give the length of the critical path from s_i to the exit node.

The algorithm presented here is same as the CS algorithm, except ranks are no longer used in ordering the subtasks within a block. Instead of using the statically computed ranks, this algorithm uses the value of ct(s_i,x), where m_x is the machine assigned to s_i in the initial mapping, to order the subtasks within a block. Thus, this variation is referred to as the minimum completion time dynamic priority (CD) algorithm.

The example shown in Figure 3 illustrates why using ranks computed at compile time to order the subtasks within a block may not lead to the best overall completion time. In the given example, the DAG shown in Figure 3(a) is mapped onto two machines m_0 and m_1. Figure 3(b) shows the subtask computation time matrix, which gives the computation time of a subtask on different machines. The initial static mapping is shown in Figure 3(c). The numbers inside each bar correspond to the subtask index and the execution time of the subtask, in ''subtask index/execution time'' notation. The times are given in seconds. The data transfer times are negligible if the source and destination machines are the same, otherwise, for this example there is a fixed time of two seconds for the data transfer. In Figure 3(a), the number outside each node indicates the rank of that subtask derived using the initial mapping.

When block 2 is considered for remapping by either the PS or CS algorithm, s_5 is mapped first and then s_4 is mapped. Suppose s_0 finishes its execution in 20 seconds instead of 10 seconds and s_1 finishes in 10 seconds. This causes the subtask s_4 to become critical and s_5 to become non-critical, i.e., s_5 is not part of the critical path anymore. By using the rank numbers that were statically computed, the PS and CS algorithms map s_5 before s_4. Thus, s_5 will be mapped to the best machine and this can delay the completion of s_4. Instead of using the statically computed ranks, the CD algorithm considers ct(s_i,x), where subtask s_i is assigned to m_x in the initial mapping. For this example, subtask s_4 is assigned to machine m_0 and subtask s_5 is assigned to machine m_1. Therefore, the CD algorithm considers ct$(s_4,0)$ and ct$(s_5,1)$ to determine the remapping order.

$$\text{ct}(s_4,0) = 20+15+10+10 = 55$$

$$\text{ct}(s_5,1) = 10+20+10+2+10 = 52$$

Because the value of ct$(s_5,1)$ is less than the value of ct$(s_4,0)$, s_4 is considered for remapping before s_5 by the CD algorithm. This example illustrates that using ct(s_i,x), where m_x is the machine that is assigned to s_i in the initial mapping, enables the remapping algorithm to track the critical path better than using the static ranks.
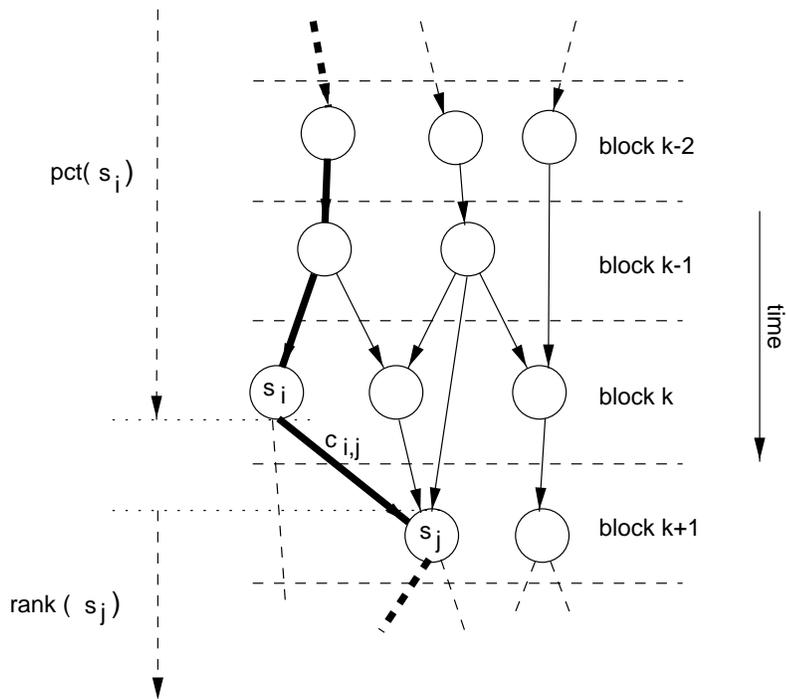
Figure 2: Estimating the completion time by considering the longest path through s_i.

4. Experimental Results and Discussion

4.1. Simulation Parameters

A simulator was implemented to evaluate the performance of the hybrid remapper variants. Various parameters are given as input to the simulator. Some parameters are specified as fixed values, e.g., number of machines, and others as a range of values with a maximum and a minimum value, e.g., subtask computation time. When a range is specified, the actual value is set to a random value within the specified range. Each data point in the results presented in this section is an average of 100 simulation runs. The experiments were performed on a Sun Ultra with a SPARC processor running at 165 MHz.

To generate a DAG that represents an application, the number of subtasks, maximum out degree of a node, number of data items to be transferred among different subtasks, range for subtask computation times, and range for data item sizes are provided as input to the simulator. Using these input parameters the simulator creates a table with the subtasks along the columns and data items along the rows. If a subtask s_j produces a data item d_i then the cell (i,j) has the label PRODUCER and if the subtask s_j consumes a data item d_i then the cell (i,j) has the label CONSUMER. A data item has one and only one producer, but may have zero or more consumers. For a given data item, a producer is randomly picked and then consumers are picked such that the resulting graph is acyclic and the maximum out degree constraints are satisfied.

To define the HC suite, the number of machines is provided as input. The simulator randomly generates valid subtask computation times to fill a table that determines the subtask computation times on each machine in the HC suite. For these experiments it is assumed that a fully connected, contention-free communication network is used. The inter-machine communication times are source and destination dependent. Communication times are specified by a range value. The run-time value of a parameter such as the subtask execution time or inter-subtask data communication time can be different from the expected value of the parameter. The variation is modeled by generating simulated run-time values by sampling a probability distribution function (PDF) that has the expected value of the parameter as the mean.

4.2. Generational Scheduling

In this subsection, the generational scheduling (GS) algorithm [5] is briefly described. The performance of the hybrid remapper is compared with the performance of the GS algorithm in the next subsection. The GS algorithm is a dynamic mapping heuristic for HC systems.

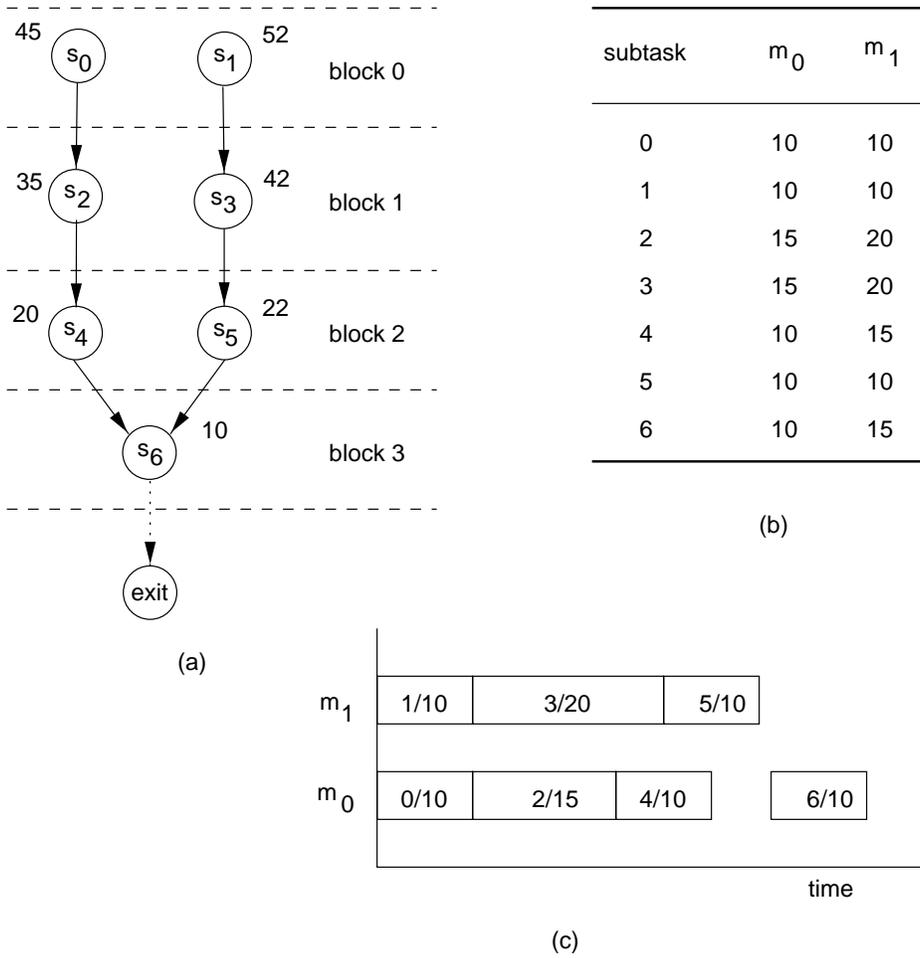| subtask	m_0	m_1
0	10	10
1	10	10
2	15	20
3	15	20
4	10	15
5	10	10
6	10	15

(a)

(b)

(c)

Figure 3: An example mapping to illustrate the benefit of the CD algorithm: (a) the partitioned DAG, (b) the subtask computation time matrix, and (c) the initial mapping.

Initially, the GS forms a partial scheduling problem by pruning all the subtasks with unsatisfied precedence constraints from the initial DAG that represents the application. The initial partial scheduling problem consists of subtasks that correspond to those in block 0 of the hybrid remapper approach. The subtasks in the initial partial scheduling problem are then mapped onto the machines using an auxiliary scheduler. The auxiliary scheduler considers the subtasks for assignment in a first come first serve order. A subtask is assigned to a machine that minimizes the completion time of that particular subtask.

When a subtask from the initial partial scheduling problem completes its execution, the GS algorithm performs a remapping. During the remapping, the GS revises the partial scheduling problem by adding and removing subtasks from it. The completion of the subtask that triggered the remapping event may have satisfied the precedence constraints of some subtasks. These subtasks are added to the initial partial scheduling problem. The subtasks that have already started execution are removed from the initial partial scheduling problem. Once the revised partial scheduling problem is obtained, the subtasks in it are mapped onto the HC machine suite using the auxiliary scheduler. This procedure is iteratively performed until the completion of all subtasks.

4.3. Hybrid Remapper

From the discussions in Section 3, it can be noted that the hybrid remapper is provided with an initial mapping that is derived prior to application execution using a static matching and scheduling algorithm. The simulator generates a random DAG, using the parameters it receives as input, at the beginning of each simulation run. An initial static mapping for this DAG is obtained by matching and scheduling this DAG onto the HC suite using the baseline algorithm [18].

The baseline algorithm that is used to derive the initial mapping is a fast static matching and scheduling algorithm. It partitions the subtasks into blocks using an algorithm similar to the one described in Subsection 3.2. Once the subtasks are partitioned into blocks, they are ordered such that a subtask in block k comes before a subtask in block l, where $k < l$. The subtasks in the same block are arranged in descending order based on the number of descendents of each subtask (ties are broken arbitrarily). The subtasks are considered for assignment by traversing the list, beginning with block 0 subtasks. A subtask is assigned to the machine that gives the shortest time for that particular subtask to complete.

In this simulator, three different PDFs (a) Erlang(2) [12], (b) uniform, and (c) skewed uniform are used to generate the simulated run-time values. For Erlang(2), the expected values are provided as the mean and the PDF is sampled to obtain a simulated run-time value. In Figure 4, 10,000 consecutive random numbers generated by the Erlang(2) random number generator with mean ten is shown using a 200-bin histogram. For the skewed uniform PDF, the following rule is used to generate the simulated run-time value. Let α_1 be the negative percentage deviation, α_2 be the positive percentage deviation, and u be a random number that is uniformly distributed in [0,1]. Then, the simulated run-time value of a parameter τ can be modeled as $\tau \times (100 - \alpha_1 + (\alpha_1 + \alpha_2)u)/100$. For the uniform PDF, $\alpha_1 = \alpha_2 = \alpha$. For the simulation results presented here, Erlang(2) is used unless otherwise noted.
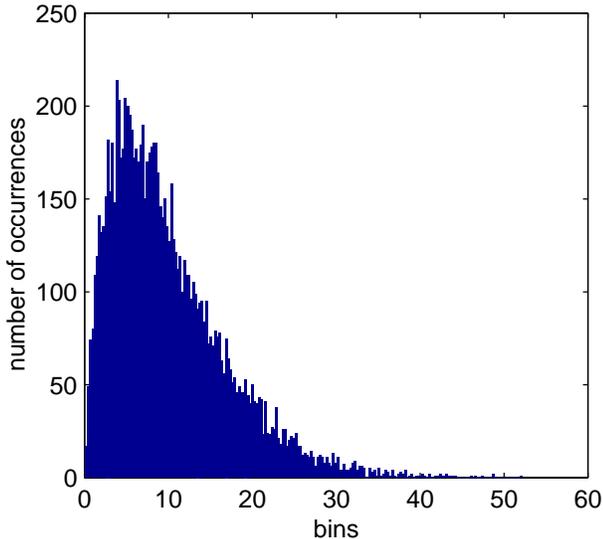

Figure 4: A 200-bin histogram for 10,000 consecutive samples of the Erlang(2) random number generator with mean equal to ten.

In these experiments, baseline refers to first deriving a static mapping using the baseline algorithm and expected subtask computation and communication times, and then, using this mapping, computing the total application execution time based on the simulated run-time values for computation and communication times. Also, in these experiments, ideal refers to deriving a static mapping using the baseline algorithm and simulated run-time values (instead of the expected values) for subtask computation and communication times. Note that this ideal is used for comparison purposes only, and cannot be implemented in practical environments. Also note that the ideal is not necessarily the optimal mapping. These simulated run-time values are also used to evaluate the application task completion time with the hybrid remapper variants.

In Figure 5(a), the performance of the PS algorithm is compared to the mapping that is obtained using the baseline algorithm for ten machines. Figure 5(b) shows a similar comparison for the CS algorithm for ten machines. The performance of the CD algorithm is shown in Figures 6(a) and 6(b). Figure 6(a) compares CD and the baseline for varying numbers of subtasks and ten machines. Figure 6(b) compares the two approaches for varying numbers of machines and 200 subtasks.

From Figures 5(a), 5(b), and 6(a) it can be observed that the performance difference among the three variants is almost negligible. The heuristic improvements performed to obtain the CS and CD variants from the PS variant of the hybrid remapper make the CS and CD use more initial matching and scheduling derived information. That is, while CS and CD use more information in an attempt to derive a better mapping than the PS, the information is based on expected values, rather than run-time values. Thus, there is no significant improvement. Also, in these simulation studies, the initial mapping is obtained using a simple baseline algorithm. The performance of CS and CD may improve if a higher quality initial assignment is used, e.g., if a genetic algorithm based mapper [18] is used for the initial matching and scheduling.

As the number of subtasks increase, the performance difference between each hybrid remapper variant and the baseline increases. This increase in performance can be attributed to two factors: (a) increased number of remapping events and (b) increased average number of subtasks per block. Increasing the number of remapping events provides the hybrid remapper with more opportunities to exploit the run-time values of parameters that are available during application execution. Also, with the increased number of subtasks per block the hybrid remapper can derive schedules that are very different from the initial schedule. Therefore, the average performance of the hybrid remapper increases with increasing number of subtasks.
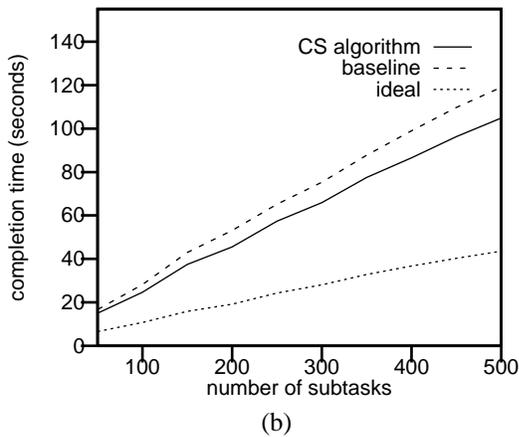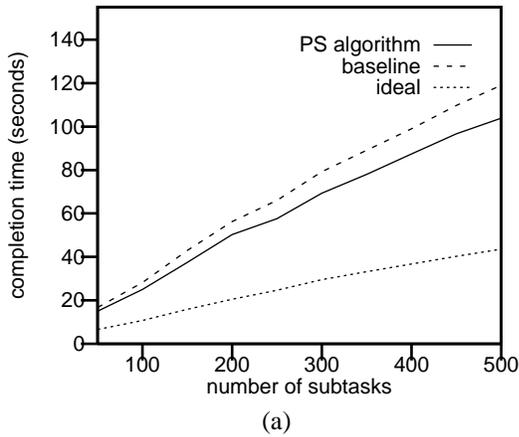
(a)

(b)

Figure 5: Performance of the hybrid remapper versus the baseline for (a) the PS algorithm and (b) the CS algorithm.

Ten machines and 100 subtasks were used in Figure 7. In Figure 7(a), the performance of the CD algorithm is compared with the baseline for varying computation/communication ratios and Figure 7(b) shows the performance comparison of the CD algorithm with the baseline for varying average number of subtasks per block. Figure 7(a) shows that the hybrid remapper performs better as the computation/communication ratio increases. The computation/communication ratio is the average subtask execution time divided by the average inter-subtask communication time. In Figure 7(a), the low computation/communication ratio denotes the range 1.0-10.0, medium computation/communication ratio denotes the range 10.0-200.0, and high computation/communication ratio denotes the range 200.0-4000.0. With increasing computation/communication ratio, the data transfer times become less significant compared to the subtask computation times. The reason for the hybrid remapper not per-

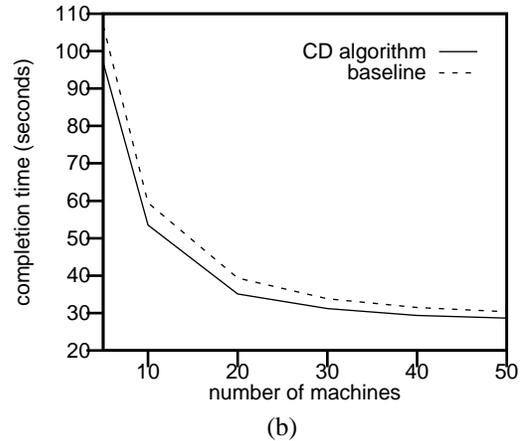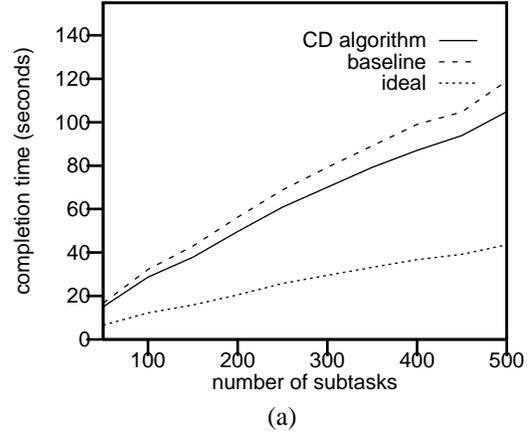forming well with a low computation/communication ratio is currently under investigation.

(a)

(b)

Figure 6: Performance of the CD algorithm versus the baseline for (a) varying the subtasks and (b) varying the machines.

From Figure 7(b) it can be noted that the relative performance of the CD algorithm increases with increasing the average number of subtasks per block. When there are more subtasks per block, it is possible for the hybrid remapper to derive mappings that are very different from the initial mapping.

Figure 8(a) compares the performance of the CD algorithm with the baseline algorithm for a uniform distribution PDF, 20 machines, and 200 subtasks. Figure 8(b) performs the same comparison for a skewed uniform distribution PDF, 20 machines, and 200 subtasks. In the skewed uniform distribution the negative percentage deviation is half of the positive percentage deviation.

As noted earlier, one of the features of the hybrid remapper algorithm that is presented here is overlapping its operation with the execution of the subtasks. To
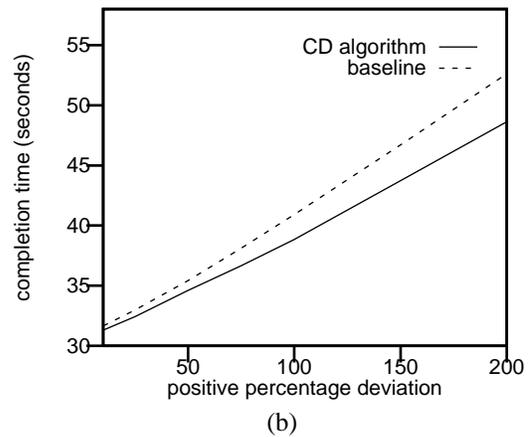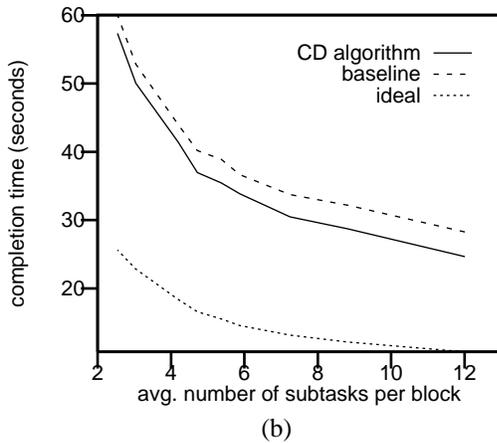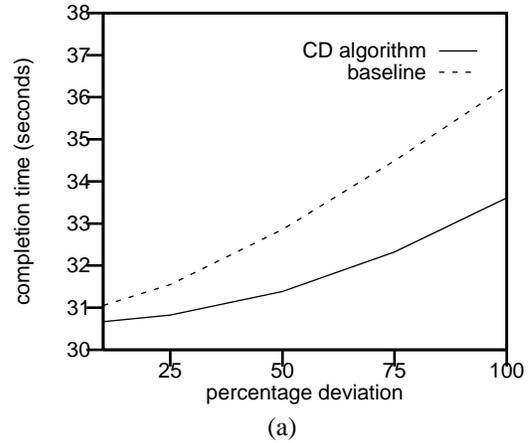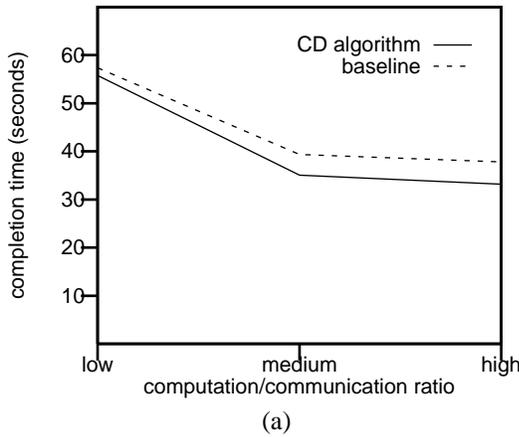
(a)

(b)

Figure 7: Performance of the CD algorithm versus the baseline for (a) varying the computation/communication ratio and (b) varying the average number of subtasks per block.

(a)

(b)

Figure 8: Performance of the CD algorithm versus the baseline for (a) using the uniform distribution for parameter modeling and (b) skewed uniform for parameter modeling.

obtain complete overlap, in the worst case, the remapping time for a block of subtasks should be less than the execution time of the smallest subtask in the previous block. More precisely, the time available for remapping block k is equal to the difference between the time the first block $k-1$ subtask begins execution and the time the first block k subtask can begin execution. Figure 9(a) shows the per block remapping time for the CD algorithm for varying numbers of subtasks and ten machines. In Figure 9(b), the per block remapping time for the CD algorithm is shown for varying numbers of machines and 200 subtasks.

In Figure 10, the performance of the CD algorithm is compared with the GS algorithm for varying numbers of subtasks. From the simulation results it can be observed that the CD algorithm is slightly outperforming the GS algorithm. From the discussions in Section 3.6, it can be

noted that the CD algorithm attempts to minimize the length of the critical path at each remapping step. In the GS algorithm, the critical path through the DAG is not considered when the subtasks are remapped. This is one reason for the better performance of the CD algorithm. The GS algorithm has more remapping events compared to the hybrid remapper. The number of remapping events is equal to the number of subtasks in the GS algorithm and equal to the number of blocks in the CD algorithm. The increased number of remappings allows the GS algorithm to base its assignment decisions on more current values. This may be why the GS is performing only three to four percent worst than the CD algorithm even though GS does not consider the critical path through the DAG. In the GS algorithm, at least one machine may be waiting on the scheduler to finish the mapping process. This scheduler induced wait time on the HC suite was not included in the GS versus CD com-
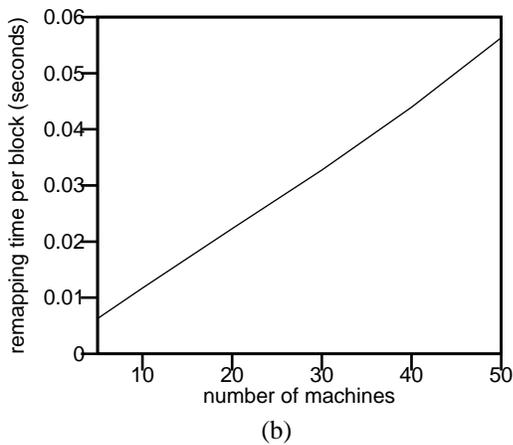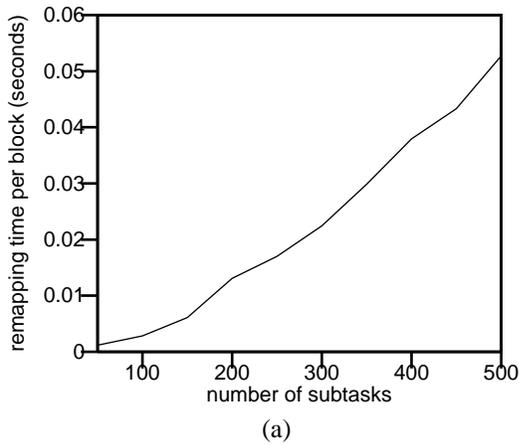
(a)

(b)

Figure 9: Per block remapping time of CD for (a) varying subtasks and (b) varying machines.
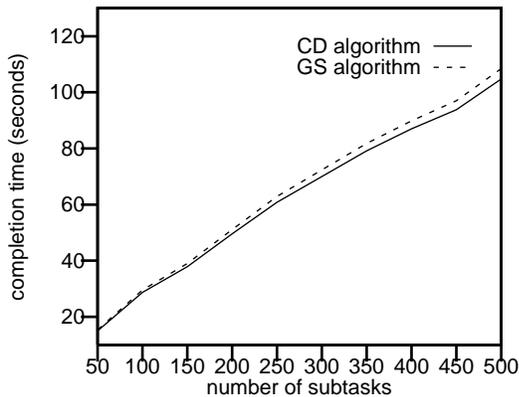
parison.

Figure 10: Performance of the CD algorithm versus the Generational Scheduling algorithm for varying numbers of subtasks.

5. Related Work

Other groups have also studied dynamic mapping heuristics for HC systems (e.g., [5, 8, 11]). A brief description of the GS algorithm and an experimental comparison of the hybrid remapper with the GS algorithm were presented in Section 4. The Self-Adjusting Scheduling for Heterogeneous Systems (SASH) algorithm is presented in [8]. One of the differences between the hybrid remapper and the SASH algorithm is that the hybrid remapper uses a list-scheduling based algorithm to perform the remappings at run time, whereas the SASH algorithm uses a variation of the branch and bound algorithm to generate the partial mappings at each remapping event. Also, unlike the GS and SASH algorithms, the hybrid remapper presented here can use any initial mapping to guide its remapping decisions, i.e., the initial mapping is used to compute the ranks and completion time estimates in the hybrid remapper. It is necessary to experimentally determine how the quality of the initial mapping impacts the overall performance of the hybrid remapper.

In [11], two mapping algorithms are presented. One is based on a distributed model and the other is based on a centralized model. The distributed mapping algorithm is different from the algorithms presented in [5, 8], and the hybrid remapper presented here, which are all centralized algorithms. The centralized mapping algorithm is based on a global queue equalization algorithm.

6. Conclusions and Future Work

The simulation results indicate that the performance of a statically obtained initial mapping can be improved by the hybrid remapper. From the simulation results obtained, performance improvement can be as much as 15% for some cases. The timings also indicate that the remapping time needed per block of subtasks is in the order hundreds of milliseconds for up to 50 machines and 500 subtasks. In the worst case situation, to obtain complete overlap, the computation time for the shortest running subtask must be greater than the per block remapping time.

The experimental studies revealed that the hybrid remapper performs better than the generational scheduling, but the margin of difference was only three to four percent. The hybrid remapper has a better machine utilization compared to the generational scheduling algorithm, because in the hybrid remapper the mapping operations are overlapped with the application execution. Further research is necessary to develop ways to improve the hybrid remapper's performance. This include examining the use of different schemes for partitioning the DAG into blocks, exploring the use of different ways of

ordering subtasks within a block, and investigating the use of different criteria for determining subtask to machine assignments.

The partitioning scheme that is currently used in the hybrid remapper does not consider the usage pattern of the data items produced by a subtask. The partitioning is solely based on the data dependencies. This could cause a subtask with low rank value in a block k to be mapped before a subtask with high rank value in a block l, where $l > k$. Various alternate partitioning schemes need to be explored and evaluated to examine different criteria for forming blocks.

One of the features of the hybrid remapper algorithm presented here is the overlap of the execution of the hybrid remapper algorithm with the execution of the sub-tasks. In the hybrid remapper developed in this research, the remapping event for block k is the readiness to execute of the first block $k-1$ subtask. Hence, the number of remapping events is equal to the number of blocks. In other algorithms, such as the Generational Scheduling algorithm [5], the number of remapping events is equal to the number of subtasks. It is necessary to study the trade-offs of increasing the number of remapping events on the performance of the algorithms and the amount of machine idle time from having to wait for a mapping decision. Also, the interaction of varying the amount of uncertainty in the parameter values and increasing the number of remapping events needs further research.

In this paper, the performance of the hybrid remapper is compared with the performance of the static baseline, and the dynamic generational scheduling algorithm [5]. Further simulation studies are necessary to compare the performance of the hybrid remapper with other dynamic mapping algorithms, such as the queue equalization algorithm [11].

The hybrid remapper developed in this research assumed a fully connected, contention-free communication model. This model needs to be improved to accommodate message contention and restricted inter-machine network topologies that occur in practical situations. Also, enhancements are necessary to support cases where a subtask can have multiple sources (machines) for a needed data item [17].

The performance of the hybrid remapper has been studied using simulations in this research. Exploring the possibility of obtaining performance bounds using analytical methods is yet another possible area of future research.

Another future area of study is to evaluate the performance of the hybrid remapper when the initial mapping is generated by a genetic algorithm (GA) based mapper [18]. Also, it would be interesting to compare the relative performance of the hybrid remapper and the mapping obtained by a static GA-based mapper as the run-time values of the parameters deviate from their expected values.

In summary, a new dynamic mapping algorithm called the hybrid remapper was presented in this paper. The hybrid remapper uses novel heuristic approaches to dynamically improve a statically obtained initial mapping. The potential of the hybrid remapper to improve the performance of initial static mappings was demonstrated using simulation studies.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, ''A comparison of list schedules for parallel processing systems,'' *Comm. of the ACM,* Vol. 17, No. 12, Dec. 1974, pp. 685-690.

[2] M. M. Eshaghian, ed., *Heterogeneous Computing,* Artech House, Norwood, MA, 1996.

[3] D. Fernandez-Baca, ''Allocating modules to processors in a distributed system,'' *IEEE Trans. on Software Engineering,* Vol. SE-15, No. 11, Nov. 1989, pp. 1427-1436.

[4] R. F. Freund, ''The challenges of heterogeneous computing,'' *Parallel Systems Fair at the 8th Int'l Parallel Processing Symp.,* Apr. 1994, pp. 84-91.

[5] R. F. Freund, B. R. Carter, D. Watson, E. Keith, and F. Mirabile, ''Generational scheduling for heterogeneous computing systems,'' *Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '96),* Aug. 1996, pp. 769-778.

[6] R. F. Freund, T. Kidd, D. Hensgen, and L. Moore, ''SmartNet: A scheduling framework for meta-computing,'' *2nd Int'l Symp. Parallel Architectures, Algorithms, and Networks (ISPAN '96),* June 1996, pp. 514-521.

[7] A. Ghafoor and J. Yang, ''Distributed heterogeneous supercomputing management system,'' *IEEE Computer,* Vol. 26, No. 6, June 1993, pp. 78-86.

[8] B. Hamidzadeh, D. J. Lilja, and Y. Atif, ''Dynamic scheduling techniques for heterogeneous computing systems,'' *Concurrency: Practice and Experience,* Vol. 7, No. 7, Oct. 1995, pp. 633-652.

[9] M. A. Iverson, F. Ozguner, and G. J. Follen, ''Parallelizing existing applications in a distributed heterogeneous environment,'' *4th Heterogeneous Computing Workshop (HCW '95),* Apr. 1995, pp. 93-100.

[10] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, ''Heterogeneous computing: Challenges and opportunities,'' *IEEE Computer,* Vol. 26, No. 6, June 1993, pp. 18-27.

[11] C. Leangsuksun, J. Potter, and S. Scott, ''Dynamic task mapping algorithms for a distributed heterogeneous computing environment,'' *4th Heterogeneous Computing Workshop (HCW '95),* Apr. 1995, pp. 30-34.

[12] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, Second Edition,* McGraw-Hill, New York, NY, 1984.

[13] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, ''Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments,'' *5th Heterogeneous Computing Workshop (HCW '96),* Apr. 1996, pp. 98-117.

[14] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, ''Heterogeneous computing,'' in *Parallel and Distributed Computing Handbook*, A. Y. Zomaya, ed., McGraw-Hill, New York, NY, 1996, pp. 725-761.

[15] H. J. Siegel, H. G. Dietz, and J. K. Antonio, ''Software support for heterogeneous computing,'' in *The Computer Science and Engineering Handbook*, A. B. Tucker, Jr., ed., CRC Press, Boca Raton, FL, 1997, pp. 1886-1909.

[16] H. Singh and A. Youssef, ''Mapping and scheduling heterogeneous task graphs using genetic algorithms,'' *5th Heterogeneous Computing Workshop (HCW '96),* Apr. 1996, pp. 86-97.

[17] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, ''Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system,'' *IEEE Trans. on Parallel and Distributed Systems*, Vol. 8, No. 8, Aug. 1997, pp. 857-871.

[18] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. ''Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach,'' *J. of Parallel and Distributed Computing,* Special Issue on Parallel Evolutionary Computing, accepted and scheduled to appear.

Biographies

Muthucumaru Maheswaran is a PhD candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, USA. His research interests include task matching and scheduling in heterogeneous computing environments, parallel languages, data parallel algorithms, distributed computing, scientific computation, and world wide web systems. He has authored or coauthored two journal papers, seven conference papers, two book chapters, and one technical report.

Mr. Maheswaran received a BScEng degree in electrical engineering from the University of Peradeniya, Sri Lanka, in 1990 and a MSEE degree from Purdue University in 1994. Mr. Maheswaran is a member of the Eta Kappu Nu honorary society.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE (1990) and a Fellow of the ACM (1998). He received BS degrees in both electrical engineering and management (1972) from MIT, and the MA (1974), MSE (1974), and PhD degrees (1977) from the Department of Electrical Engineering and Computer Science at Princeton University. Prof. Siegel has coauthored over 250 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing* (second edition 1990). He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing* (1989-1991), and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* (1993-1996) and the *IEEE Transactions on Computers* (1993-1996). He was Program Chair/Co-Chair of three conferences, General Chair/Co-Chair of four conferences, and Chair/Co-Chair of four workshops. He is an international keynote speaker and tutorial lecturer, and a consultant for government and industry.

Prof. Siegel's heterogeneous computing research includes modeling, mapping heuristics, and minimization of inter-machine communication. He is an Investigator on the Management System for Heterogeneous Networks (MSHN) project, supported by the DARPA/ITO Quorum program to create a management system for a heterogeneous network of machines. He is the Principal Investigator of a joint ONR-DARPA/ISO grant to design efficient methodologies for communication in the heterogeneous environment of the Battlefield Awareness and Data Dissemination (BADD) program.

Prof. Siegel's other research interests include parallel algorithms, interconnection networks, and the PASM reconfigurable parallel machine. His algorithm work involves minimizing execution time by exploiting architectural features of parallel machines. Topological properties and fault tolerance are the focus of his research on interconnection networks for parallel machines. He is investigating the utility of the dynamic reconfigurability and mixed-mode parallelism supported by the PASM design ideas and the small-scale prototype.