Journal of Parallel and Distributed Computing **59**, 107–131 (1999)
Article ID jpdc.1999.1581, available online at http://www.idealibrary.com on IDEAL®

Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems[1]

Muthucumaru Maheswaran

Department of Computer Science, University of Manitoba, Winnipeg, MB R3T 2N2, Canada
E-mail: maheswar@cs.umanitoba.ca

Shoukat Ali and Howard Jay Siegel

Purdue University, School of Electrical and Computer Engineering, West Lafayette, Indiana 47907-1285
E-mail: alis@ecn.purdue.edu, hj@ecn.purdue.edu

Debra Hensgen

Department of Computer Science, Naval Postgraduate School, Monterey, California 93940
E-mail: hensgen@cs.nps.navy.mil

and

Richard F. Freund

NOEMIX Inc., 1425 Russ Boulevard, Suite T-110, San Diego, California 92101
E-mail: rffreund@home.com

Received December 30, 1998; revised May 30, 1999; accepted July 2, 1999

Dynamic mapping (matching and scheduling) heuristics for a class of independent tasks using heterogeneous distributed computing systems are studied. Two types of mapping heuristics are considered, immediate mode and batch mode heuristics. Three new heuristics, one for batch mode and two for immediate mode, are introduced as part of this research. Simulation studies are performed to compare these heuristics with some existing ones. In total five immediate mode heuristics and three batch mode heuristics are examined. The immediate mode dynamic heuristics consider, to varying degrees and in different ways, task affinity for different machines and machine ready times. The batch mode dynamic heuristics consider these factors, as well as aging of tasks waiting to execute. The simulation results reveal that the choice of which dynamic mapping heuristic to use in a given heterogeneous environment depends on parameters such as (a) the structure of the heterogeneity among tasks and machines and (b) the arrival rate of the tasks. © 1999 Academic Press

[1] This research was supported by the DARPA/ITO Quorum Program under the NPS subcontract numbers N62271-97-M-0900, N62271-98-M-0217, and N62271-98-M-0448, and under the GSA subcontract number GS09K99BH0250. Some of the equipment used was donated by Intel and Microsoft.

Key Words: batch mode mapping; dynamic mapping; mapping heuristics; meta-task mapping; immediate mode mapping.

1. INTRODUCTION

In general, *heterogeneous computing* (HC) is the coordinated use of different types of machines, networks, and interfaces to maximize their combined performance and/or cost-effectiveness [6, 9, 18]. HC is an important technique for efficiently solving collections of computationally intensive problems [7]. As machine architectures become more advanced to obtain higher peak performance, the extent to which a given task can exploit a given architectural feature depends on how well the task's computational requirements match the machine's advanced capabilities. The applicability and strength of HC systems are derived from their ability to match computing needs to appropriate resources. HC systems have resource management systems (RMSs) to govern the execution of the tasks that arrive for service. This paper describes and compares eight heuristics that can be used in such an RMS for dynamically assigning independent tasks to machines.

In a general HC system, schemes are necessary to assign tasks to machines (*matching*) and to compute the execution order of the tasks assigned to each machine (*scheduling*) [3]. The process of matching and scheduling tasks is referred to as *mapping*. *Dynamic* methods to do this operate as tasks arrive. This is in contrast to *static* techniques, where the complete set of tasks to be mapped is known *a priori*, the mapping is done prior to the execution of any of the tasks, and more time is available to compute the mapping (e.g., [4, 27]).

In the HC environment considered here, the tasks are assumed to be independent, i.e., no communications between the tasks are needed. This scenario is likely to be present, for instance, when many independent users submit their jobs to a collection of shared computational resources. A dynamic scheme is needed because the arrival times of the tasks may be random and some machines in the suite may go off-line and new machines may come on-line. The dynamic mapping heuristics investigated in this study are nonpreemptive and assume that the tasks have no deadlines or priorities associated with them.

The mapping heuristics can be grouped into two categories, immediate mode and batch mode heuristics. In the immediate mode, a task is mapped onto a machine as soon as it arrives at the mapper. In the *batch mode*, tasks are not mapped onto the machines as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called *mapping events*. The independent set of tasks that is considered for mapping at the mapping events is called a *meta-task*. A meta-task can include newly arrived tasks (i.e., the ones arriving after the last mapping event) and the ones that were mapped in earlier mapping events but did not begin execution. While immediate mode heuristics consider a task for mapping only once, batch mode heuristics consider a task for mapping at each mapping event until the task begins execution.

The trade-offs among and between immediate mode and batch mode heuristics are studied experimentally. Mapping independent tasks onto an HC suite is a well-known NP-complete problem if throughput is the optimization criterion [12]. For the heuristics discussed in this paper, maximization of throughput is the primary objective, because this performance measure is the most common one in production oriented environments.

Three new heuristics, one for batch mode and two for immediate mode, are introduced as part of this research. Simulation studies are performed to compare these heuristics with some existing ones. In total, five immediate mode heuristics and three batch mode heuristics are examined. The immediate mode heuristics consider, to varying degrees and in different ways, task affinity for different machines and machine ready times. The batch mode heuristics consider these factors, as well as aging of tasks waiting to execute.

Section 2 describes some related work. Section 3 defines an optimization criterion and discusses the mapping approaches studied here. The simulation procedure is given in Section 4. Section 5 presents the simulation results.

This research is part of a DARPA/ITO Quorum Program project called *MSHN* (pronounced "mission") (Management System for Heterogeneous Networks) [11]. MSHN is a collaborative research effort that includes the Naval Postgraduate School, NOEMIX, Purdue, and the University of Southern California. It builds on SmartNet, an implemented scheduling framework and system for managing resources in an HC environment developed at NRaD [8]. 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 heuristics developed here, or their derivatives, may be included in the Scheduling Advisor component of the MSHN prototype.

2. RELATED WORK

Related work in the literature was examined to select a set of heuristics appropriate for the HC environment considered here, and then to perform comparative studies. This section is a sampling of related literature and is not meant to be exhaustive.

In the literature, mapping tasks onto machines is often referred to as scheduling. Several researchers have worked on the dynamic mapping problem from areas including job shop scheduling and distributed computer systems (e.g., [13, 16, 23, 25]).

The heuristics presented in [12] are concerned with mapping independent tasks onto heterogeneous machines such that the completion time of the last finishing task is minimized. The problem is recognized as NP-complete, and worst case performance bounds are obtained for the heuristics. Some of these heuristics are designed for a general HC environment, while the rest target either a heterogeneous two-machine system or a general homogeneous system. Of the heuristics designed for a general HC environment, the A-schedule, the B-schedule, and the C-schedule heuristics are simply variations of the minimum completion time heuristic used here. The Min-min heuristic that is used here as a benchmark for batch mode mapping is based on the D-schedule, and is also one of the heuristics implemented in SmartNet [8].

The scheme in [13] is representative of techniques for mapping communicating subtasks to an HC suite, considering data dependency graphs and communication times between machines. Thus, an environment very different from the set of independent tasks considered here is used. Hence, the heuristics developed for that different environment are not appropriate for the HC environment considered here.

Two dynamic mapping approaches, one using a distributed policy and the other using a centralized policy, are developed in [16]. The heuristic based on the distributed policy uses a method similar to the minimum completion time heuristic at each node. The mapper at a given node considers the local machine and the k highest communication bandwidth neighbors to map the tasks in the local queue. Therefore, the mapper based on the distributed strategy assigns a task to the best machine among the $k+1$ machines. The centralized heuristic referred to in [16] as the global queue equalization algorithm is similar to the minimum completion time heuristic that is used as a benchmark in this paper. The simulation results provided in [16] show that the centralized heuristic always performs better than the distributed heuristic. Hence, the minimum completion time heuristic used here represents the better of the two heuristics presented in [16].

A survey of dynamic scheduling heuristics for job-shop environments is provided in [25]. It classifies the dynamic scheduling algorithms into three approaches: a knowledge-based approach, a conventional approach, and a distributed problem solving approach. The heuristics with a knowledge-based approach take a long time to execute and hence are not suitable for the particular dynamic environment considered here. The classes of heuristics grouped under the conventional and distributed problem solving approaches are similar to the minimum completion time heuristic considered in this paper. However, the problem domains considered in [25] involve precedence constraints among the tasks, priorities, or deadlines and thus differ from the domain here.

In distributed computer systems, load balancing schemes have been a popular strategy for mapping tasks onto the machines (e.g., [19, 23]). In [19], the performance characteristics of simple load balancing heuristics for HC distributed systems are studied. The heuristics presented in [19] do not consider task execution times when making their decisions. In [23], a survey of dynamic scheduling heuristics for distributed computing systems is provided. All heuristics, except one, in [23] schedule tasks on different machines using load sharing techniques, without considering task execution times. (The one heuristic in [23] that does not use load sharing employs deadlines to schedule tasks, and therefore it falls out of the problem domain discussed here.) The load balancing heuristic used in this research is representative of the load balancing techniques in [19] and [23].

SmartNet [8] is an RMS for HC systems that employs various heuristics to map tasks to machines considering resource and task heterogeneity. In this paper, some SmartNet heuristics appropriate for the HC environment considered here are included in the comparative study (minimum completion time, Min-min, and Max-min).

3. MAPPING HEURISTICS

3.1. Overview

The *expected execution time* e_{ij} of task t_i on machine m_j is defined as the amount of time taken by m_j to execute t_i given m_j has no load when t_i is assigned. The *expected completion time* c_{ij} of task t_i on machine m_j is defined as the wall-clock time at which m_j completes t_i (after having finished any previously assigned tasks). Let m be the total number of machines in the HC suite. Let K be the set containing the tasks that will be used in a given test set for evaluating heuristics in the study. Let the *a*rrival time of the task t_i be a_i, and let the time t_i *b*egins execution be b_i. From the above definitions, $c_{ij} = b_i + e_{ij}$. Let c_i be the completion time for task t_i, and it is equal to c_{ij} where machine m_j is assigned to execute task t_i. The *makespan* for the complete schedule is then defined as $\max_{t_i \in K}(c_i)$ [21]. Makespan is a measure of the throughput of the HC system and does not measure the quality of service imparted to an individual task. One other performance metric is considered in [17].

In the immediate mode heuristics, each task is considered only once for matching and scheduling, i.e., the mapping is not changed once it is computed. When the arrival rate is low enough, machines may be ready to execute a task as soon as it arrives at the mapper. Therefore, it may be beneficial to use the mapper in the immediate mode so that a task need not wait until the next mapping event to begin its execution.

Recall from Section 1 that, in immediate mode, the mapper assigns a task to a machine as soon as the task arrives at the mapper, and in batch mode a set of independent tasks that need to be mapped at a mapping event is called a meta-task. (In some systems, the term meta-task is defined in a way that allows inter-task dependencies.) In batch mode, for the *i*th mapping event, the meta-task M_i is mapped at time τ_i, where $i \geqslant 0$. The initial meta-task, M_0, consists of all the tasks that arrived prior to time τ_0, i.e., $M_0 = \{t_j \mid a_j < \tau_0\}$. The meta-task, M_k, for $k > 0$, consists of tasks that arrived after the last mapping event and the tasks that had been mapped but had not started executing, i.e., $M_k = \{t_j \mid \tau_{k-1} \leqslant a_j < \tau_k\} \cup \{t_j \mid a_j < \tau_{k-1}, b_j > \tau_k\}$.

In batch mode, the mapper considers a meta-task for matching and scheduling at each mapping event. This enables the mapping heuristics to possibly make better decisions than immediate mode heuristics. This is because the batch heuristics have the resource requirement information for a whole meta-task and know about the actual execution times of a larger number of tasks (as more tasks might complete while waiting for the mapping event). When the task arrival rate is high, there will be a sufficient number of tasks to keep the machines busy in between the mapping events and while a mapping is being computed. (It is, however, assumed in this study that the running time of each mapping heuristic is negligibly small as compared to the average task execution time.)

Both immediate mode and batch mode heuristics assume that estimates of expected task execution times on each machine in the HC suite are known. The assumption that these estimated expected times are known is commonly made when studying mapping heuristics for HC systems (e.g., [10, 15, 24]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [18].) These estimates can be supplied before a task is submitted for execution, or at the time it is submitted.

The *ready time* of a machine is the earliest wall clock time that the machine is going to be ready after completing the execution of the tasks that are currently assigned to it. Because the heuristics presented here are dynamic, the expected machine ready times are based on a combination of actual task execution times (for tasks that have completed execution on that machine) and estimated expected task execution times (for tasks assigned to that machine that are waiting to execute). It is assumed that each time a task t_i completes on a machine m_j a report is sent to the mapper, and the ready time for m_j is updated if necessary. The experiments presented in Section 5 model this situation using simulated actual values for the execution times of the tasks that have already finished their execution.

All heuristics examined here operate in a centralized fashion and map tasks onto a dedicated suite of machines; i.e., the mapper controls the execution of all jobs on all machines in the suite. It is assumed that each mapping heuristic is being run on a separate machine. (While all heuristics studied here are functioning dynamically, use of some of these heuristics in a static environment is discussed in [4].)

3.2. *On-Line Mode Mapping Heuristics*

Five immediate mode heuristics are described here. These are (i) minimum completion time, (ii) minimum execution time, (iii) switching algorithm, (iv) *k*-percent best, and (v) opportunistic load balancing. Of these five heuristics, switching algorithm and *k*-percent best have been proposed as part of the research presented here.

The minimum completion time (MCT) heuristic assigns each task to the machine that results in that task's earliest completion time. This causes some tasks to be assigned to machines that do not have the minimum execution time for them. The MCT heuristic is a variant of the fast-greedy heuristic from SmartNet [8]. The MCT heuristic is used as a benchmark for the immediate mode, i.e., the performance of the other heuristics is compared against that of the MCT heuristic. As a task arrives, all the machines in the HC suite are examined to determine the machine that gives the earliest completion time for the task. Therefore, it takes $O(m)$ time to map a given task.

The minimum execution time (MET) heuristic assigns each task to the machine that performs that task's computation in the least amount of execution time (this heuristic is also known as LBA (limited best assignment) [1] and UDA (user directed assignment) [8]). This heuristic, in contrast to MCT, does not consider machine ready times. This heuristic can cause a severe imbalance in load across the machines. The advantages of this method are that it gives each task to the machine that performs it in the least amount of execution time, and the heuristic is very simple. The heuristic needs $O(m)$ time to find the machine that has the minimum execution time for a task.

The switching algorithm (SA) is motivated by the following observations. The MET heuristic can potentially create load imbalance across machines by assigning many more tasks to some machines than to others, whereas the MCT heuristic tries to balance the load by assigning tasks for earliest completion time. If the tasks are arriving in a random mix, it is possible to use the MET at the expense of load balance until a given threshold and then use the MCT to smooth the load across the machines. The SA heuristic uses the MCT and MET heuristics in a cyclic fashion depending on the load distribution across the machines. The purpose is to have a heuristic with the desirable properties of both the MCT and the MET.

Let the maximum (latest) ready time over all machines in the suite be $r_{\max}$ and the minimum (earliest) ready time be $r_{\min}$. Then, the *load balance index* across the machines is given by $\pi = r_{\min}/r_{\max}$. The parameter π can have any value in the interval [0,1]. If π is 1.0, then the load is evenly balanced across the machines. If π is 0, then at least one machine has not yet been assigned a task. Two threshold values, π_l (low) and π_h (high), for the ratio π are chosen in [0, 1] such that $\pi_l < \pi_h$. Initially, the value of π is set to 0.0. The SA heuristic begins mapping tasks using the MCT heuristic until the value of the load balance index increases to at least π_h. After that point in time, the SA heuristic begins using the MET heuristic to perform task mapping. This causes the load balance index to decrease. When it decreases to π_l or less, the SA heuristic switches back to using the MCT heuristic for mapping the tasks and the cycle continues.

As an example of the functioning of the SA with lower and upper limits of 0.6 and 0.9, respectively, for $|K| = 1000$ and one particular rate of arrival of tasks, the SA switched between the MET and the MCT two times (i.e., from the MCT to the MET to the MCT), assigning 715 tasks using the MCT. For $|K| = 2000$ and the same task arrival rate, the SA switched five times, using the MCT to assign 1080 tasks. The percentage of tasks assigned using MCT gets progressively smaller for larger $|K|$. This is because the larger the $|K|$, the larger the number of tasks waiting to execute on a given machine, and therefore, the larger the ready time of a given machine. This in turn means that an arriving task's execution time will impact the machine ready time less, thereby rendering the load balance index less sensitive to a load-imbalancing assignment.

At each task's arrival, the SA heuristic determines the load balance index. In the worst case, this takes $O(m)$ time. In the next step, the time taken to assign a task to a machine is of order $O(m)$ whether SA uses the MET to perform the mapping or the MCT. Overall, the SA heuristic takes $O(m)$ time irrespective of which heuristic is actually used for mapping the task.

The *k*-percent best (KPB) heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the $m \times (k/100)$ best machines based on the execution times for the task, where $100/m \leqslant k \leqslant 100$. The task is assigned to a machine that provides the earliest completion time in the subset. If $k = 100$, then the KPB heuristic is reduced to the MCT heuristic. If $k = 100/m$, then the KPB heuristic is reduced to the MET heuristic. A "good" value of k maps a task to a machine only within a subset formed from computationally superior machines. The purpose is not so much to match the current task to a computationally well-matched machine as it is to avoid putting the current task onto a machine which might be more suitable for some yet-to-arrive tasks. This "foresight" about task heterogeneity is missing in the MCT, which might assign a task to a poorly matched machine for an immediate marginal improvement in completion time, possibly depriving some subsequently arriving better matched tasks of that machine, and eventually leading to a larger makespan as compared to the KPB. It should be noted that while both the KPB and the SA combine elements of the MCT and the MET in their operation, it is only in the KPB that *each* task assignment attempts to optimize objectives of the MCT and the MET simultaneously. However, in cases where a fixed subset of machines is not among the k% best for any of the tasks, the KPB will cause more machine idle time compared to the MCT and can result in much poorer performance. Thus the relative performance of the KPB and the MCT may depend on the HC suite of machines and characteristics of the tasks being executed.

For each task, $O(m \log m)$ time is spent in ranking the machines for determining the subset of machines to examine. Once the subset of machines is determined, it takes $O(m \times k/100)$ time, i.e., $O(m)$ time, to determine the machine assignment. Overall the KPB heuristic takes $O(m \log m)$ time.

The opportunistic load balancing (OLB) heuristic assigns a task to the machine that becomes ready next, without considering the execution time of the task onto that machine. If multiple machines become ready at the same time, then one machine is arbitrarily chosen. The complexity of the OLB heuristic is dependent on the implementation. In the implementation considered here, the mapper may need to examine all m machines to find the machine that becomes ready next. Therefore, it takes $O(m)$ to find the assignment. Other implementations may require idle machines to assign tasks to themselves by accessing a shared global queue of tasks [26].

As an example of the workings of these heuristics, consider a simple system of three machines, m_0, m_1, and m_2, currently loaded so that expected ready times are as given in Table 1. Consider the performance of the heuristics for a very simple case of three tasks t_0, t_1, and t_2 arriving in that order. Table 2 shows the expected execution times of tasks on the machines in the system. All time values in the discussion below are the expected values.

The MET finds that all tasks have their minimum completion time on m_2, and even though m_2 is already heavily loaded, it assigns all three tasks to m_2. The time when t_0, t_1, and t_2 will all have completed is 245 units.

The OLB assigns t_0 to m_0 because m_0 is expected to be idle soonest. Similarly, it assigns t_1 and t_2 to m_1 and m_0, respectively. The time when t_0, t_1, and t_2 will all have completed is 170 units.

The MCT determines that the minimum completion time for t_0 will be achieved on m_0, and makes this assignment, even though the execution time of t_0 on m_0 is more than twice that on m_1 (where the completion time would have been only slightly larger). Then MCT goes on to assign t_1 to m_0 and t_2 to m_1 so that the time when t_0, t_1, and t_2 will all have completed is 160 units.

The SA first determines the current value of the load balance index, π, which is 75/200 or 0.38. Assume that π_l is 0.40 and that π_h is 0.70. Because $\pi < \pi_l$, the SA chooses the MCT to make the first assignment. After the first assignment

TABLE 1

Initial Ready Times of the Machines (Arbitrary Units)

m_0	m_1	m_2
75	110	200

TABLE 2

Expected Execution Times (Arbitrary Units)

	m_0	m_1	m_2
t_0	50	20	15
t_1	20	60	15
t_2	20	50	15

$\pi = 110/200 = 0.55 < \pi_h$. So the SA continues to use the MCT for the second assignment as well. It is only after third assignment that $\pi = 145/200 = 0.725 > \pi_h$ so that the SA will then use the MET for the fourth arriving task. The time when t_0, t_1, and t_2 will all have completed here is the same as that for the MCT.

Let the value of k in the KPB be 67% so that the KPB will choose from the two fastest executing machines to assign a given task. For t_0, these machines are m_1 and m_2. Within these two machines, the minimum completion time is achieved on m_1 so t_0 is assigned to m_1. This is the major difference from the working of the MCT; m_0 is not loaded with t_0 even though t_0 would have its minimum completion time (over all machines) there. This step saves m_0 for any yet-to-arrive tasks that may run slowly on other machines. One such task is t_2; in the MCT it is assigned to m_1, but in the KPB it is assigned to m_0. The time when t_0, t_1, and t_2 will all have completed using the KPB is 135 units. This is the smallest among all five heuristics.

3.3. *Batch Mode Mapping Heuristics*

Three batch mode heuristics are described here: (i) the Min-min heuristic, (ii) the Max-min heuristic, and (iii) the Sufferage heuristic. The Sufferage heuristic has been proposed as part of the research presented here. In the batch mode heuristics, meta-tasks are mapped after predefined intervals. These intervals are defined in this study using one of the two strategies proposed below.

The *regular time interval* strategy maps the meta-tasks at regular intervals of time (e.g., every 10 s). The only occasion when such a mapping will be redundant is when (1) no new tasks have arrived since the last mapping and (2) no tasks have finished executing since the last mapping (thus, machine ready times are unchanged). These conditions can be checked for, so redundant mapping events can be avoided.

The *fixed count* strategy maps a meta-task M_i as soon as one of the following two mutually exclusive conditions are met: (a) an arriving task makes $|M_i|$ larger than or equal to a predetermined arbitrary number κ, or (b) all tasks in the set $|K|$ have arrived, and a task completes while the number of tasks which yet have to begin is larger than or equal to κ. In this strategy, the length of the mapping intervals will depend on the arrival rate and the completion rate. The possibility of machines being idle while waiting for the next mapping event will depend on the arrival rate, completion rate, m, and κ. (For the arrival rates in the experiments here, this strategy operates reasonably; in an actual system, it may be necessary for tasks to have a maximum waiting time to be mapped.)

```
(1)  for all tasks $t_i$ in meta-task $M_v$ (in an arbitrary order)
(2)       for all machines $m_j$ (in a fixed arbitrary order)
(3)            $c_{ij} = e_{ij} + r_j$
(4)  do until all tasks in $M_v$ are mapped
(5)       for each task in $M_v$ find the earliest completion
               time and the machine that obtains it
(6)       find the task $t_k$ with the minimum earliest
               completion time
(7)       assign task $t_k$ to the machine $m_l$ that gives the
(8)            earliest completion time
(9)       delete task $t_k$ from $M_v$
(10)      update $r_l$
(11)      update $c_{il}$ for all $i$
(12) enddo
```

FIG. 1. The Min-min heuristic.

The batch mode heuristics considered in this study are discussed in the paragraphs below. The complexity analysis performed for these heuristics considers a single mapping event, and the meta-task size is assumed to be equal to the average of meta-task sizes at all actually performed mapping events. Let the *average meta-task size* be S.

The *Min-min* heuristic shown in Fig. 1 is one of the heuristics implemented in SmartNet [8]. In Fig. 1, let r_j denote the expected time that machine m_j will become ready to execute a task after finishing the execution of all tasks assigned to it at that point in time. First the c_{ij} entries are computed using the e_{ij} and r_j values. For each task t_i, the machine that gives the earliest expected completion time is determined by scanning the ith row of the c matrix (composed of the c_{ij} values). The task t_k that has the minimum earliest expected completion time is determined and then assigned to the corresponding machine. The matrix c and vector r are updated, and the above process is repeated for tasks that have not yet been assigned a machine.

Min-min begins by scheduling the tasks that change the expected machine ready time status by the least amount. If tasks t_i and t_k are contending for a particular machine m_j, then Min-min assigns m_j to the task (say t_i) that will change the ready time of m_j less. This increases the probability that t_k will still have its earliest completion time on m_j and shall be assigned to it. Because at $t = 0$, the machine which finishes a task earliest is also the one that executes it fastest, and from thereon Min-min heuristic changes machine ready time status by the least amount for every assignment, the percentage of tasks assigned their first choice (on basis of expected execution time) is likely to be higher in Min-min than with the other batch mode heuristics described in this section (this has been verified by examining the simulation study data [17]). The expectation is that a smaller makespan can be obtained if a larger number of tasks is assigned to the machines that not only complete them earliest but also execute them fastest.

The initialization of the c matrix in Line (1) to Line (3) of Fig. 1 takes $O(Sm)$ time. The **do** loop of the Min-min heuristic is repeated S times and each iteration takes $O(Sm)$ time. Therefore, the heuristic takes $O(S^2m)$ time.

The *Max-min* heuristic is similar to the Min-min heuristic, and is one of the heuristics implemented in SmartNet [8]. It differs from the Min-min heuristic (given in Fig. 1) in that once the machine that provides the earliest completion time is found for every task, the task t_k that has the *maximum* earliest completion time is determined and then assigned to the corresponding machine. That is, in Line (6) of Fig. 1, "minimum" would be changed to "maximum". The Max-min heuristic has the same complexity as the Min-min heuristic.

The Max-min is likely to do better than the Min-min heuristic in cases where there are many more shorter tasks than longer tasks. For example, if there is only one long task, Max-min will execute many short tasks concurrently with the long task. The resulting makespan might just be determined by the execution time of the long task in this case. Min-min, however, first finishes the shorter tasks (which may be more or less evenly distributed over the machines) and then executes the long task, increasing the makespan compared to the Max-min.

The *Sufferage* heuristic (shown in Fig. 2) is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it. Let the *sufferage value* of a task t_i be the difference between its second earliest completion time (on some machine m_y) and its earliest completion time (on some machine m_x). That is, using m_x will result in the best completion time for t_i, and using m_y the second best.

```
(1)  for all tasks $t_k$ in meta-task $M_v$ (in an arbitrary order)
(2)      for all machines $m_j$ (in a fixed arbitrary order)
(3)          $c_{kj} = e_{kj} + r_j$
(4)  do until all tasks in $M_v$ are mapped
(5)      mark all machines as unassigned
(6)      for each task $t_k$ in $M_v$ (in a fixed arbitrary order)
             /* for a given execution of the for statement,
             each $t_k$ in $M_v$ is considered only once */
(7)          find machine $m_j$ that gives the earliest
                 completion time
(8)          sufferage value = second earliest completion
                 time − earliest completion time
(9)          if machine $m_j$ is unassigned
(10)             assign $t_k$ to machine $m_j$, delete $t_k$
                 from $M_v$, mark $m_j$ assigned
(11)         else
(12)             if sufferage value of task $t_i$ already
                 assigned to $m_j$ is less than the
                 sufferage value of task $t_k$
(13)                 unassign $t_i$, add $t_i$ back to $M_v$,
                     assign $t_k$ to machine $m_j$,
                     delete $t_k$ from $M_v$
(14)     endfor
(15)     update the vector $r$ based on the tasks that
             were assigned to the machines
(16)     update the $c$ matrix
(17) enddo
```

FIG. 2. The Sufferage heuristic.

The initialization phase in Lines (1) to (3), in Fig. 2, is similar to the ones in the Min-min and Max-min heuristics. Initially all machines are marked unassigned. In each iteration of the **for** loop in Lines (6) to (14), pick arbitrarily a task t_k from the meta-task. Find the machine m_j that gives the earliest completion time for task t_k, and tentatively assign m_j to t_k if m_j is unassigned. Mark m_j as assigned, and remove t_k from meta-task. If, however, machine m_j has been previously assigned to a task t_i, choose from t_i and t_k the task that has the higher sufferage value, assign m_j to the chosen task, and remove the chosen task from the meta-task. The unchosen task will not be considered again for this execution of the **for** statement, but shall be considered for the next iteration of the **do** loop beginning on Line (4). When all the iterations of the **for** loop are completed (i.e., when one execution of the **for** statement is completed), update the machine ready time of each machine that is assigned a new task. Perform the next iteration of the **do** loop beginning on Line (4) until all tasks have been mapped.

Table 3 shows a scenario in which the Sufferage will outperform the Min-min. Table 3 shows the expected execution time values for four tasks on four machines (all initially idle). In this case, the Min-min heuristic gives a makespan of 93 and the Sufferage heuristic gives a makespan of 78. Figure 3 gives a pictorial representation of the assignments made for the case in Table 3.

From the pseudo-code given in Fig. 2, it can be observed that first execution of the **for** statement on Line (6) takes $O(Sm)$ time. The number of task assignments made in one execution of this **for** statement depends on the total number of machines in the HC suite, the number of machines that are being contended for among different tasks, and the number of tasks in the meta-task being mapped. In the worst case, only one task assignment will be made in each execution of the **for** statement. Then meta-task size will decrease by one at each **for** statement execution. The outer **do** loop will be iterated S times to map the whole meta-task. Therefore, in the worst case, the time $T(S)$ taken to map a meta-task of size S will be

$$T(S) = Sm + (S-1)\,m + (S-2)\,m + \cdots + m$$
$$T(S) = O(S^2 m).$$

In the best case, there are as many machines as there are tasks in the meta-task, and there is no contention among the tasks. Then all the tasks are assigned in the

TABLE 3

An Example Expected Execution Time Matrix That Illustrates the Situation in Which the Sufferage Heuristic Outperforms the Min-min Heuristic

	m_0	m_1	m_2	m_3
t_0	40	48	134	50
t_1	50	82	88	89
t_2	55	68	94	93
t_3	52	60	78	108

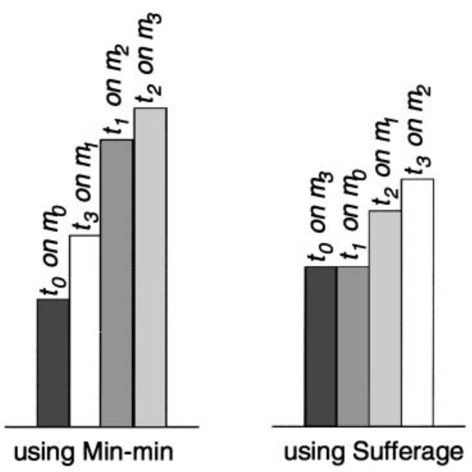

FIG. 3. An example scenario (based on Table 3) in which the Sufferage gives a shorter makespan than the Min-min (bar heights are proportional to task execution times).

first execution of the **for** statement so that $T(S) = O(Sm)$. Let ω be a number quantifying the extent of contention among the tasks for the different machines. The complexity of the Sufferage heuristic can then be given as $O(\omega Sm)$, where $1 \leqslant \omega \leqslant S$. It can be seen that ω is equal to S in the worst case and is 1 in the best case; these values of ω are numerically equal to the number of iterations of the **do** loop on Line (4), for the worst and the best case, respectively.

The batch mode heuristics can cause some tasks to be starved of machines. Let H_i be a subset of meta-task M_i consisting of tasks that were mapped (as part of M_i) at the mapping event i at time τ_i but did not begin execution by the next mapping event at τ_{i+1}. H_i is the subset of M_i that is included in M_{i+1}. Due to the expected heterogeneous nature of the tasks, the meta-task M_{i+1} may be so mapped that some or all of the tasks arriving between τ_i and τ_{i+1} may begin executing before the tasks in set H_i do. It is possible that some or all of the tasks in H_i may be included in H_{i+1}. This probability increases as the number of new tasks arriving between τ_i and τ_{i+1} increases. In general, some tasks may be remapped at each successive mapping event without actually beginning execution (i.e., the task is *starving* for a machine). This impacts the response time the user sees (this is examined as a "sharing penalty" in [17]).

To reduce starvation, aging schemes are implemented. The *age* of a task is set to zero when it is mapped for the first time and incremented by one each time the task is remapped. Let σ be a constant that can be adjusted empirically to change the extent to which aging affects the operation of the heuristic. An *aging factor*, $\zeta = (1 + \text{age}/\sigma)$, is then computed for each task. For the experiments in this study, σ is arbitrarily set to 10 (so that the aging factor for a task increases by one after every 10 remappings of the task). The aging factor is used to enhance the probability of an "older" task beginning before the tasks that would otherwise begin first. In the Min-min heuristic, for each task, the completion time obtained in Line (5) of Fig. 1 is multiplied by the corresponding value for $1/\zeta$. As the age of a task increases, its age-compensated expected completion time (i.e., one used to determine the mapping) gets increasingly smaller than its original expected completion time. This increases its probability of being selected in Line (6) in Fig. 1.

For the Max-min heuristic, the completion time of a task is multiplied by ζ. In the Sufferage heuristic, the sufferage value computed in Line (8) in Fig. 2 is multiplied by ζ.

4. SIMULATION PROCEDURE

The mappings are simulated using a discrete event simulator (e.g., [5, 14, 22]). The task arrivals are modeled by a Poisson random process. The simulator contains an *ETC* (expected time to compute) matrix that contains the expected execution times of a task on all machines, for all the tasks that can arrive for service. The ETC matrix entries used in the simulation studies represent the e_{ij} values (in seconds) that the heuristic would use in its operation. The actual execution time of a task can be different from the value given by the ETC matrix. This variation is modeled by generating a *simulated actual execution time* for each task by sampling a truncated Gaussian probability density function with variance equal to three times the expected execution time of the task and mean equal to the expected execution time of the task (e.g., [2, 20]). If the sampling results in a negative value, the value is discarded and the same probability density function is sampled again. This process is repeated until a positive value is returned by the sampling process.

In an ETC matrix, the numbers along a row indicate the estimated expected execution times of the corresponding task on different machines. The average variation along the rows is referred to as the *machine heterogeneity* [2]. Similarly, the numbers along a column of the ETC matrix indicate the estimated expected execution times of the machine for different tasks. The average variation along the columns is referred to as the *task heterogeneity* [2]. One classification of heterogeneity is to divide it into high heterogeneity and low heterogeneity. Based on the above idea, four categories were proposed for the ETC matrix in [2]: (a) high task heterogeneity and high machine heterogeneity (HiHi), (b) high task heterogeneity and low machine heterogeneity (HiLo), (c) low task heterogeneity and high machine heterogeneity (LoHi), and (d) low task heterogeneity and low machine heterogeneity (LoLo).

The ETC matrix can be further classified into two classes, consistent and inconsistent, which are orthogonal to the previous classifications. For a *consistent* ETC matrix, if machine m_x has a lower execution time than machine m_y for task t_k, then the same is true for any task t_i. The ETC matrices that are not consistent are *inconsistent* ETC matrices. Inconsistent ETC matrices occur in practice when (1) there is a variety of different machine architectures in the HC suite (e.g., parallel machines, superscalars, workstations) and (2) there is a variety of different computational needs among the tasks (e.g., readily parallelizable tasks, difficult to parallelize tasks, tasks that are floating point intensive, simple text formatting tasks). Thus, the way in which a task's needs correspond to a machine's capabilities may differ for each possible pairing of tasks to machines.

In addition to the consistent and inconsistent classes, a *semiconsistent* class could also be defined. A semiconsistent ETC matrix is characterized by a consistent submatrix. In the semiconsistent ETC matrices used here, 50% of the tasks and 25% of the machines define a consistent submatrix. Furthermore, it is assumed that for a particular task the execution times that fall within the consistent submatrix are smaller than those that fall out. This assumption is justified because one way for some machines to perform consistently better than others for some tasks is to be very much faster for those tasks than the other machines.

Let an ETC matrix have $t_{\max}$ rows and $m_{\max}$ columns. Random ETC matrices that belong to the different categories are generated in the following manner:

1. Let Γ_t be an arbitrary constant quantifying task heterogeneity, being smaller for low task heterogeneity. Let N_t be a number picked from the uniform random distribution with range $[1, \Gamma_t]$.

2. Let Γ_m be an arbitrary constant quantifying machine heterogeneity, being smaller for low machine heterogeneity. Let N_m be a number picked from the uniform random distribution with range $[1, \Gamma_m]$.

3. Sample N_t $t_{\max}$ times to get a vector $q[0..(t_{\max}-1)]$.

4. Generate the ETC matrix, $e[0..(t_{\max}-1), 0..(m_{\max}-1)]$ by the following algorithm:

```
for $t_i$ from 0 to $(t_{\max}-1)$
    for $m_j$ from 0 to $(m_{\max}-1)$
        pick a new value for $N_m$
        e[i, j] = q[i]*$N_m$.
    endfor
endfor
```

From the raw ETC matrix generated above, a semiconsistent matrix could be generated by sorting the execution times across a random subset of the machines for each task in a random subset of tasks. An inconsistent ETC matrix could be obtained simply by leaving the raw ETC matrix as such. Consistent ETC matrices were not considered in this study because they are least likely to arise in the current intended MSHN environment.

In the experiments described here, the values of Γ_t for low and high task heterogeneities are 1000 and 3000, respectively. The values of Γ_m for low and high machine heterogeneities are 10 and 100, respectively. These heterogeneous ranges are based on one type of expected environment for MSHN.

5. EXPERIMENTAL RESULTS AND DISCUSSION

5.1. *Overview*

The experimental evaluation of the heuristics is performed in three parts. In the first part, the immediate mode heuristics are compared using various metrics. The second part involves a comparison of the batch mode heuristics. The third part is the comparison of the batch mode and the immediate mode heuristics. Unless stated otherwise, the following are valid for the experiments described here. The number of machines is held constant at 20, and the experiments are performed for $|K| = \{1000, 2000\}$. All heuristics are evaluated in a HiHi heterogeneity environment, both for the inconsistent and the semiconsistent cases, because these correspond to some of the currently expected MSHN environments.

For each value of $|K|$, tasks are mapped under two different Poisson arrival rates, λ_h and λ_l, such that $\lambda_h > \lambda_l$. The value of λ_h is chosen empirically to be high enough to allow at most 50% tasks to have completed when the last task in the set arrives. That is, for λ_h, when at least 50% of the tasks execute no new tasks are arriving. This may correspond to a situation in which tasks are submitted during the day but not at night.

In contrast, λ_l is chosen to be low enough to allow at least 90% of the tasks to have completed when the last task in the set arrives. That is, for λ_l, when at most 10% of the tasks execute no new tasks are arriving. This may correspond more closely than λ_h to a situation where tasks arrive continuously. The difference between λ_h and λ_l can also be considered to represent a difference in burstiness.

Some experiments were also performed at a third arrival rate λ_t, where λ_t was high enough to ensure that only 20% of the tasks have completed when the last task in the set arrived. The MCT heuristic was used as a basis for these percentages. Unless otherwise stated, the task arrival rate is set to λ_h.

Example comparisons are discussed in Subsections 5.2 to 5.4. Each data point in the comparison charts is an average over 50 trials, where for each trial the simulated actual task execution times are chosen independently. The makespan for each trial for each heuristic has been normalized with respect to the benchmark heuristic, which is the MCT for immediate mode heuristics and the Min-min for the batch heuristics. The Min-min serves as a benchmark also for the experiments where batch heuristics are compared with immediate mode heuristics. Each bar (except the one for the benchmark heuristic) in the comparison charts gives a 95% confidence interval (shown as an "I" on the top of bars) for the mean of the normalized value. Occasionally upper bound, lower bound, or the entire confidence interval is not distinguishable from the mean value, for the scale used in the graphs here. More general conclusions about the heuristics' performance are in Section 6.

5.2. Comparisons of the Immediate Mode Heuristics

Unless otherwise stated, the immediate mode heuristics are investigated under the following conditions. In the KPB heuristic, k is equal to 20%. This particular value of k was found to give the lowest makespan for the KPB heuristic under the conditions of the experiments. For the SA, the lower threshold and the upper threshold for the load balance index are 0.6 and 0.9, respectively. Once again these values were found to give optimum values of makespan for the SA.

In Fig. 4, immediate mode heuristics are compared based on a normalized makespan for inconsistent HiHi heterogeneity. From Fig. 4, it can be noted that the KPB heuristic completes the execution of the last finishing task earlier than the other heuristics (however, it is only slightly better than the MCT). For $k = 20\%$ and $m = 20$, the KPB heuristic forces a task to choose a machine from a subset of four machines. These four machines have the lowest execution times for the given task.

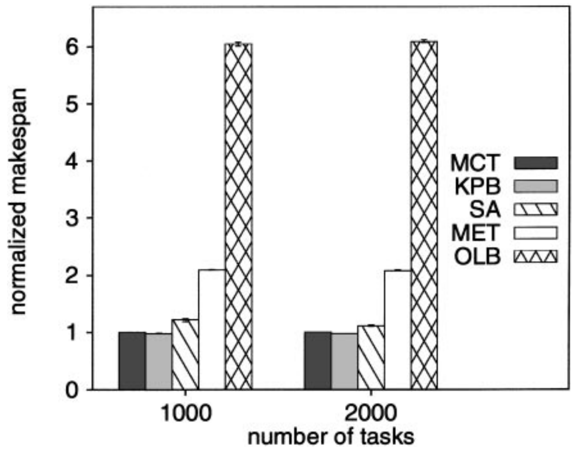

FIG. 4. Makespan for the immediate mode heuristics for inconsistent HiHi heterogeneity.

The chosen machine would give the smallest completion time as compared to other machines in the set.

Figure 5 compares the normalized makespans of the different immediate mode heuristics for semiconsistent HiHi heterogeneity. As shown in Figs. 4 and 5, the relative performance of the different immediate mode heuristics is impacted by the degree of consistency of the ETC matrices. However, the KPB still performs best, closely followed by the MCT.

For the semiconsistent type of heterogeneity, machines within a particular subset perform tasks that lie within a particular subset faster than other machines. From Fig. 5, it can be observed that for semiconsistent ETC matrices, the MET heuristic performs the worst. For the semiconsistent matrices used in these simulations, the MET heuristic maps half of the tasks to the same machine, considerably increasing the load imbalance. Although the KPB also considers only the fastest four machines for each task for the particular value of *k* used here (which happen to be the same four machines for half of the tasks), the performance does not differ much from the

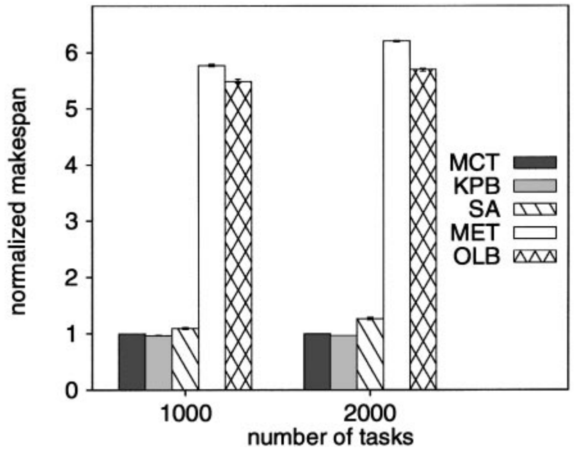

FIG. 5. Makespan of the immediate mode heuristics for semi-consistent HiHi heterogeneity.

inconsistent HiHi case. Additional experiments have shown that the KPB performance is quite insensitive to values of k as long as k is larger than the minimum value (where the KPB heuristic is reduced to the MET heuristic). For example, when k is doubled from its minimum value of 5%, the makespan decreases by a factor of about five. However a further doubling of k brings down the makespan by a factor of only about 1.2.

5.3. Comparisons of the Batch Mode Heuristics

Figure 6 compares the batch mode heuristics based on normalized makespan. In these comparisons, unless otherwise stated, the regular time interval strategy is employed to schedule meta-task mapping events. The time interval is set to 10 s. This value was empirically found to optimize makespan over other values. From Fig. 6, it can be noted that the Sufferage heuristic outperforms the Min-min and the Max-min heuristics based on makespan (although, it is only slightly better than the Min-min). The Sufferage heuristic considers the "loss" in completion time of a task if it is not assigned to its first choice in making the mapping decisions. By assigning their first choice machines to the tasks that have the highest sufferage values among all contending tasks, the Sufferage heuristic reduces the overall completion time.

Furthermore, it can be noted that the makespan given by the Max-min is much larger than the makespans obtained by the other two heuristics. Using reasoning similar to that given in Subsection 3.3 for explaining better expected performance for the Min-min, it can be seen that the Max-min assignments change a given machine's ready time status by a larger amount than the Min-min assignments do. If tasks t_i and t_k are contending for a particular machine m_j, then the Max-min assigns m_j to the task (say t_i) that will increase the ready time of m_j more. This decreases the probability that t_k will still have its earliest completion time on m_j and shall be assigned to it. Experimental data shows that the percentage of tasks assigned their minimum execution time machine is likely to be lower for the Max-min than for other batch mode heuristics [17]. It might be expected that a larger

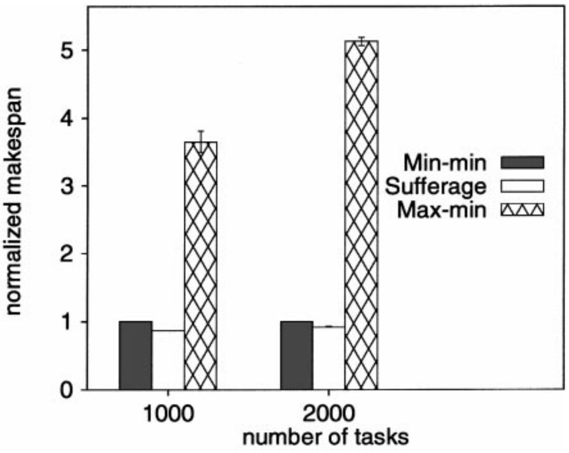

FIG. 6. Makespan of the batch mode heuristics for the regular time interval strategy and inconsistent HiHi heterogeneity.

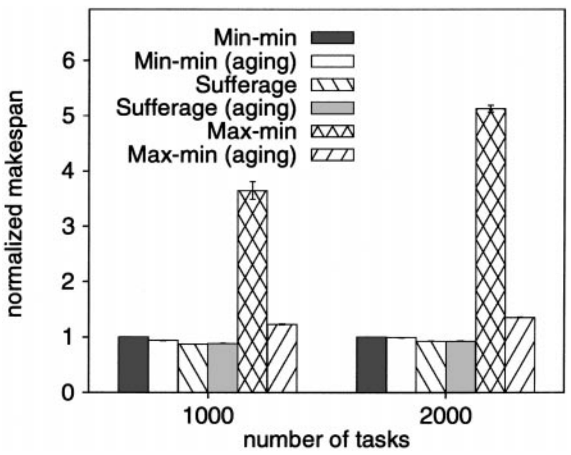

FIG. 7. Makespan for the batch mode heuristics for the regular time interval strategy with and without aging for inconsistent HiHi heterogeneity.

makespan will result if a larger number of tasks is assigned to the machines that do not have the best execution times for those tasks. Although not shown here, the results for makespan for semiconsistent HiHi are similar to those for inconsistent HiHi.

The impact of aging on batch mode heuristics is shown in Fig. 7. The Min-min without aging is used here to normalize the performance of the other heuristics. The Max-min benefits most from the aging scheme. Recall that the Min-min performs much better than the Max-min when there is no aging. Aging modifies the Max-min's operation so that tasks with smaller completion times can be scheduled prior to those with larger completion times, thus reducing the negative aspects of that technique. This is discussed further in [17].

Figure 8 shows the result of repeating the above experiments with a fixed count strategy for a batch size of 40. This particular batch size was found to give an optimum value of the makespan for the Min-min heuristic. The Min-min with

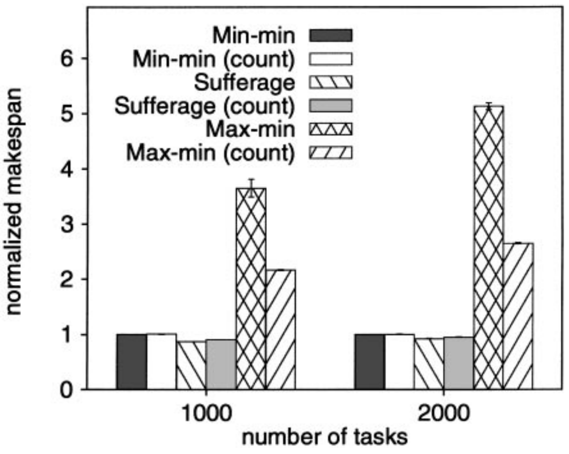

FIG. 8. Comparison of the makespans given by the regular time interval strategy and the fixed count strategy for inconsistent HiHi heterogeneity.

regular time interval strategy (interval of 10 s) is used here to normalize the performance of the other heuristics. Figure 8 compares regular time interval strategy and fixed count strategy on the basis of normalized makespans given by different heuristics for inconsistent HiHi heterogeneity. It can be seen that the fixed count approach gives similar results for the Min-min and the Sufferage heuristics. The Max-min heuristic, however, benefits considerably from the fixed count approach; makespan drops to about 60% for $|K| = 1000$ and to about 50% for $|K| = 2000$, as compared to the makespan given by the regular time interval strategy. A possible explanation lies in a conceptual element of similarity between the fixed count approach and the aging scheme. The value of $\kappa = 40$ used here resulted in batch sizes that were smaller than those using the 10 s regular time interval strategy. Thus, small tasks waiting to execute will have fewer tasks to compete with and, hence, less chance of being delayed by a larger task. Although not shown here, the results for the semiconsistent case show that as compared to the inconsistent case, the regular time interval approach gives slightly better results than the fixed count approach for the Sufferage and the Min-min. For the Max-min, however, the above two cases gave very similar results.

It should be noted that all the results given here are for inconsistent HiHi heterogeneity. For other types of heterogeneity the results might be different. For example, for inconsistent LoLo heterogeneity, the performance of the Max-min is almost identical to that of the Min-min [17].

5.4. Comparing Immediate and Batch Mode Heuristics

In Fig. 9, two immediate mode heuristics, the MCT and the KPB, are compared with two batch mode heuristics, the Min-min and the Sufferage. The comparison is performed with Poisson arrival rate set to λ_h. It can be noted that for this "high" arrival rate and $|K| = 2000$, batch heuristics are superior to immediate mode heuristics. This is because the number of tasks waiting to begin execution is likely to be larger in the above circumstances than in any other considered here, which in turn means

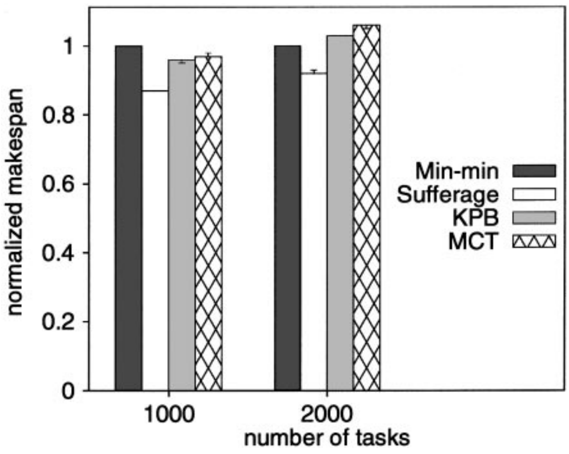

FIG. 9. Comparison of the makespan given by batch mode heuristics (regular time interval strategy) and immediate mode heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_h.

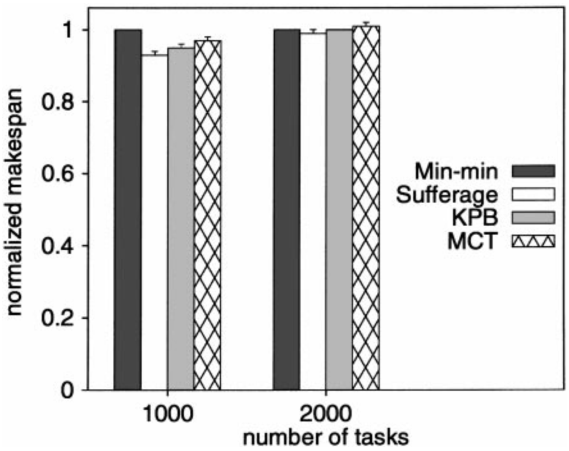

FIG. 10. Comparison of the makespan given by batch mode heuristics (regular time interval strategy) and immediate mode heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_1.

that rescheduling is likely to improve many more mappings in such a system. The immediate mode heuristics consider only one task when they try to optimize machine assignment, and they do not reschedule. Recall that the mapping heuristics use a combination of expected and actual task execution times to compute machine ready times. The immediate mode heuristics are likely to approach the performance of the batch ones at low task arrival rates, because then both classes of heuristics have comparable information about the actual execution times of the tasks. For example, at a certain low arrival rate, the 100th arriving task might find that 70 previously arrived tasks have completed. At a higher arrival rate, only 20 tasks might have completed when the 100th task arrives. The above observation is supported by the graph in Fig. 10, which shows that the relative performance difference between immediate mode and batch mode heuristics decreases with a decrease in

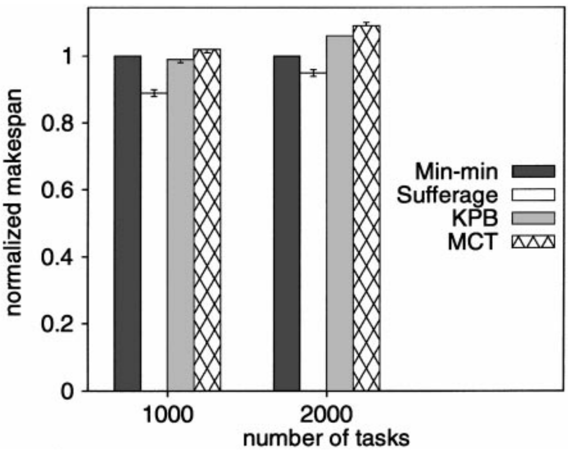

FIG. 11. Comparison of the makespan given by batch mode heuristics (regular time interval strategy) and immediate mode heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_t.

arrival rate. Given the observation that the KPB and the Sufferage perform almost similarly at this low arrival rate, it might be better to use the KPB heuristic because of its smaller computational complexity.

Figure 11 shows the performance difference between immediate mode and batch mode heuristics at an even faster arrival rate of λ_t. It can be seen that for $|K| = 2000$ batch mode heuristics outperform immediate mode heuristics with a larger margin here. Although not shown in the results here, the makespan values for all heuristics are larger for lower arrival rate. This is attributable to the fact that at lower arrival rates, there typically is more machine idle time.

6. CONCLUSIONS

New and previously proposed dynamic matching and scheduling heuristics for mapping independent tasks onto HC systems were compared under a variety of simulated computational environments. Five immediate mode heuristics and three batch mode heuristics were studied.

In the immediate mode, for both the semiconsistent and the inconsistent types of HiHi heterogeneity, the KPB heuristic outperformed the other heuristics (however, the KPB was only slightly better than the MCT). The relative performance of the OLB and the MET with respect to the makespan reversed when the heterogeneity was changed from the inconsistent to the semiconsistent. The OLB did better than the MET for the semiconsistent case.

In the batch mode, for the semiconsistent and the inconsistent types of HiHi heterogeneity, the Sufferage performed the best (though, the Sufferage was only slightly better than the Min-min). The batch mode heuristics were shown to give a smaller makespan than the immediate mode ones for large $|K|$ and high task arrival rate. For smaller values of $|K|$ and lower task arrival rates, the improvement in makespan offered by batch mode heuristics is was shown to be nominal.

This study quantifies how the relative performance of these dynamic mapping heuristics depends on (a) the consistency property of the ETC matrix and (b) the arrival rate of the tasks. Thus, the choice of the heuristic that is best to use in a given heterogeneous environment will be a function of such factors. Therefore, it is important to include a set of heuristics in a resource management system for HC and then use the heuristic that is most appropriate for a given situation (as will be done in the Scheduling Advisor for MSHN).

Researchers can build on the evaluation techniques and results presented here in future efforts by considering other nonpreemptive dynamic heuristics, as well as preemptive ones. Furthermore in future studies, tasks can be characterized in more complex ways (e.g., inter-task communications, deadlines, priorities [3]) and using other environmental factors (e.g., task arrival rates, degrees of heterogeneity, number of machines in the HC suite, impact of changing the variance when simulating actual task execution times). Thus, the studies given in this paper illustrate some evaluation techniques, examine important heuristics, and provide comparisons, as well as act as a framework for future research.

ACKNOWLEDGMENTS

A preliminary version of portions of this paper was presented at the 8th IEEE Heterogeneous Computing Workshop (HCW '99), April 1999. The authors thank Taylor Kidd, Surjamukhi Chatterjea, and Tracy D. Braun for their valuable comments and suggestions.

REFERENCES

1. R. Amstrong, D. Hensgen, and T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predications, *in* "7th IEEE Heterogeneous Computing Workshop (HCW '98)," pp. 79–87, 1998.
2. R. Amstrong, "Investigation of Effect of Different Run-Time Distributions on SmartNet Performance," Thesis, Department of Computer Science, Naval Postgraduate School, 1997. [D. Hensgen, Advisor]
3. T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems, *in* "1998 IEEE Symposium on Reliable Distributed Systems," pp. 330–335, 1998.
4. T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, R. F. Freund, and D. Hensgen, A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems, *in* "8th IEEE Heterogeneous Computing Workshop (HCW '99)," pp. 15–29, 1999.
5. A. H. Buss, A tutorial on discrete-event modeling with simulation graphs, *in* "1995 Winter Simulation Conference (WSC '95)," pp. 74–81, 1995.
6. M. M. Eshaghian, Ed., "Heterogeneous Computing," Artech House, Norwood, MA, 1996.
7. I. Foster and C. Kesselman, Eds., "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufmann, San Fransisco, CA, 1999.
8. R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, Scheduling resources in multiuser, heterogeneous, computing environments with SmartNet, *in* "7th IEEE Heterogeneous Computing Workshop (HCW '98)," pp. 184–199, 1998.
9. R. F. Freund and H. J. Siegel, Heterogeneous processing, *IEEE Comput*. **26**, 6 (June 1993), 13–17.
10. A. Ghafoor and J. Yang, Distributed heterogeneous supercomputing management system, *IEEE Comput*. **26**, 6 (June 1993), 78–86.
11. D. A. Hensgen, T. Kidd, D. St. John, M. C. Schnaidt, H. J. Siegel, T. D. Braun, M. Maheswaran, S. Ali, J.-K. Kim, C. Irvine, T. Levin, R. F. Freund, M. Kussow, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, An overview of MSHN: The management system for heterogeneous networks, *in* "8th IEEE Heterogeneous Computing Workshop (HCW '99)," pp. 184–198, 1999.
12. O. H. Ibarra and C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, *J. ACM* **24**, 2 (Apr. 1977), 280–289.
13. M. A. Iverson and F. Ozguner, Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment, *in* "7th IEEE Heterogeneous Computing Workshop (HCW '98)," pp. 70–78, 1998.
14. R. Jain, "The Art of Computer Systems Performance Analysis," Wiley, New York, 1991.
15. M. Kafil and I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, *IEEE Concurrency* **6**, 3 (July–Sep. 1998), 42–51.
16. C. Leangsuksun, J. Potter, and S. Scott, Dynamic task mapping algorithms for a distributed heterogeneous computing environment, *in* "4th IEEE Heterogeneous Computing Workshop (HCW '95)," pp. 30–34, 1995.

17. M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "A Comparison of Dynamic Strategies for Mapping a Class of Independent Tasks onto Heterogeneous Computing Systems," Technical Report, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, in preparation.

18. M. Maheswaran, T. D. Braun, and H. J. Siegel, Heterogeneous distributed computing, *in* "Encyclopedia of Electrical and Electronics Engineering" (J. G. Webster, Ed.), Wiley, New York, Vol. 8, pp. 679–690, 1999.

19. R. Mirchandaney, D. Towsley, and J. A. Stankovic, Adaptive load sharing in heterogeneous distributed systems, *J. Parallel Distrib. Computing* **9**, 4 (Aug. 1990), 331–346.

20. A. Papoulis, "Probability, Random Variables, and Stochastic Processes," McGraw–Hill, New York, 1984.

21. M. Pinedo, "Scheduling: Theory, Algorithms, and Systems," Prentice Hall, Englewood Cliffs, NJ, 1995.

22. U. W. Pooch and J. A. Wall, "Discrete Event Simulation: A Practical Approach," CRC Press, Boca Raton, FL, 1993.

23. H. G. Rotithor, Taxonomy of dynamic task scheduling schemes in distributed computing systems, *IEE Proc. Comp. Digital Techn.* **141**, 1 (Jan. 1994), 1–10.

24. H. Singh and A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, *in* "5th IEEE Heterogeneous Computing Workshop (HCW '96)," pp. 86–97, 1996.

25. V. Suresh and D. Chaudhuri, Dynamic rescheduling—A survey of research, *Internat. J. Production Econom.* **32**, 1 (Aug. 1993), 53–63.

26. P. Tang, P. C. Yew, and C. Zhu, Impact of self-scheduling on performance of multiprocessor systems, *in* "3rd International Conference on Supercomputing," pp. 593–603, 1988.

27. 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. Parallel Distrib. Comput.* **47**, 1 (Nov. 1997), 8–22.

MUTHUCUMARU MAHESWARAN is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. In 1990, he received a B.Sc. degree in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an M.S.E.E. degree in 1994 and a Ph.D. degree in 1998, both from the School of Electrical and Computer Engineering at Purdue University. He held a Fulbright scholarship during his tenure as an M.S.E.E. student at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, Internet and world wide web systems, metacomputing, mobile programs, network computing, parallel computing, resource management systems for metacomputing, and scientific computing. He has authored or coauthored 15 technical papers in these and related areas. He is a member of the Eta Kappa Nu honorary society.

SHOUKAT ALI is pursuing a Ph.D. from the School of Electrical and Computer Engineering at Purdue University, where he is currently a Research Assistant. His main research topic is dynamic mapping of meta-tasks in heterogeneous computing systems. He has held teaching positions at the Aitchison College and the Keynesian Institute of Management and Sciences, both in Lahore, Pakistan. He was also a Teaching Assistant at Purdue. Shoukat received his B.S. degree in electrical and electronic engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1996. He received his M.S.E.E. from the School of Electrical and Computer Engineering at Purdue University in 1999. His research interests include computer architecture, parallel computing, and heterogeneous computing.

HOWARD JAY SIEGEL is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received B.S. degrees in both electrical engineering and management from MIT, and the M.A., M.S.E. and Ph.D. degrees from the Department of Electrical Engineering and Computer Science at Princeton University. Professor Siegel has coauthored over 250 technical papers and coedited seven volumes, and he wrote the book "Interconnection Networks for Large-Scale Parallel Processing." He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing* and was on the Editorial Boards of the *IEEE Transactions on*

Parallel and Distributed Systems and the *IEEE Transactions on Computers*. 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.

DEBRA HENSGEN is an Associate Professor in the Computer Science Department at The Naval Postgraduate School. She received her Ph.D. in the area of Distributed Operating Systems from the University of Kentucky. She is currently a Principal Investigator of the DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) and a co-investigator of the DARPA-sponsored Server and Active Agent Management (SAAM) Next Generation Internet project. Her areas of interest include active modeling in resource management systems, network rerouting to preserve quality of service guarantees, visualization tools for performance debugging of parallel and distributed systems, and methods for aggregating sensor information. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems.

RICHARD F. FREUND is a founder and CEO of NOEM1X, a San Diego based start-up company to commercialize distributed computing technology. Freund is also one of the early pioneers in the field of distributed computing, in which he has written or co-authored a number of papers. In addition he is a founder of the Heterogeneous Computing Workshop, held each year in conjunction with the International Parallel and Distributed Processing Symposium. Freund won a Meritorious Civilian Service Award during his former career as a government scientist.