Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems

Muthucumaru Maheswaran[†], Shoukat Ali[‡], Howard Jay Siegel[‡],
Debra Hensgen[◇], and Richard F. Freund[⋆]

[†]Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2, Canada
Email: maheswar@cs.umanitoba.ca

[‡] School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285 USA
Email: {alis,hj}@ecn.purdue.edu

[◇]Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940 USA
Email: hensgen@cs.nps.navy.mil

[⋆]NOEMIX Inc.
1425 Russ Blvd., Ste. T-110
San Diego, CA 92101 USA
Email: rffreund@noemix.com

Abstract

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: on-line and batch mode heuristics. Three new heuristics, one for batch and two for on-line, are introduced as part of this research. Simulation studies are performed to compare these heuristics with some existing ones. In total, five on-line heuristics and three batch heuristics are examined. The on-line heuristics consider, to varying degrees and in different ways, task affinity for different machines and machine ready times. The batch heuristics consider these factors, as well as aging of tasks waiting to execute. The simulation results reveal that the choice of mapping heuristic depends on parameters such as: (a) the structure of the heterogeneity among tasks and machines, (b) the optimization requirements, and (c) the arrival rate of the tasks.

1. Introduction

An emerging trend in computing is to use distributed heterogeneous computing (HC) systems constructed by networking various machines to execute a set of tasks [5, 14]. These 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 assigning tasks to machines.

In a general HC system, dynamic schemes are necessary to assign tasks to machines (matching), and to compute the execution order of the tasks assigned to each machine (scheduling) [3]. In the HC system considered here, the tasks are assumed to be independent, i.e., no communications between the tasks are needed. 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 (matching and scheduling) heuristics investigated in this study are non-preemptive, and assume that the tasks have no deadlines or priorities associated with them.

The mapping heuristics can be grouped into two categories: on-line mode and batch-mode heuristics. In the on-line 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 on-line 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 between on-line 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 [9]. For the heuristics discussed in this paper, maximization of the throughput is the primary objective. This performance metric is the most common one in the production oriented environments. However, the performance of the heuristics is examined us-

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. Some of the equipment used was donated by Intel.

ing other metrics as well.

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

Section 2 describes some related work. In Section 3, the optimization criterion and another performance metric are defined. Section 4 discusses the mapping approaches studied here. The simulation procedure is given in Section 5. Section 6 presents the simulation results.

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

2. Related Work

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., [10, 12, 18, 20]).

Some of the heuristics examined for batch-mode mapping in this paper are based on the static heuristics given in [9]. The heuristics presented in [9] 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 several heuristics are designed. Worst case performance bounds are obtained for the heuristics. The Min-min heuristic that is used here as a benchmark for batch mode mapping is based on the ideas presented in [9], and implemented in SmartNet [6].

In [10], a dynamic matching and scheduling scheme based on a distributed policy for mapping tasks onto HC systems is provided. A task can have several subtasks, and the subtasks can have data dependencies among them. In the scheme presented in [10], the subtasks in an application receive information about the subtasks in other applications only in terms of load estimates on the machines. Each application uses an algorithm that uses a weighting factor to determine the mapping for the subtasks. The weighting factor for a subtask is derived by considering the length of the critical path from the subtask to the end of the directed acyclic graph (DAG) that represents the application. If each application is an independent task with no subtasks, as is the case in this paper, then the scheme presented in [10] is not suitable, because the mapping criterion is designed to exploit information available in a DAG. Therefore, the scheme provided in [10] is not compared to the heuristics presented in this paper.

Two dynamic mapping approaches, one using a centralized policy and the other using a distributed policy, are developed in [12]. The centralized heuristic referred to therein as the global queue equalization algorithm is similar to the minimum completion time heuristic that is used as a benchmark in this paper and described in Section 4. 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 simulation results provided in [12] show that the centralized heuristic always performs better than the distributed heuristic. The heuristics in [12] are very similar to the minimum completion time heuristic used as a benchmark in this paper. Hence, they are not experimentally compared with the heuristics presented here.

In [18], a survey of dynamic scheduling heuristics for distributed computing systems is provided. Most of the heuristics featured in [18] perform load sharing to schedule the tasks on different machines, not considering any task-machine affinities while making the mapping decisions for HC systems. In contrast to [18], these affinities are considered to varying degrees in all but one of the heuristics examined in this paper.

A survey of dynamic scheduling heuristics for job-shop environments is provided in [20]. It classifies the dynamic scheduling algorithms into three approaches: conventional approach, knowledge-based approach, and distributed problem solving approach. The class of heuristics grouped under the conventional approach are similar to the minimum completion time heuristic considered in this paper, however, the problem domains considered in [20] and here differ. Furthermore, some of the heuristics featured in [20] use priorities and deadlines to determine the task scheduling order whereas priorities and deadlines are not considered here.

In distributed computer systems, load balancing schemes have been a popular strategy for mapping tasks onto the machines (e.g., [15, 18]). In [15], the performance characteristics of simple load balancing heuristics for HC distributed

systems are studied. The heuristics presented in [15] do not consider task execution times when making their decisions.

SmartNet [6] is an RMS for HC systems that employs various heuristics to map tasks to machines considering resource and task heterogeneity. In this paper, some appropriate selected SmartNet heuristics are included in the comparative study.

3. Performance Metrics

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 the 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 arrival time of the task t_i be a_i, and let the begin time of t_i be b_i. From the above definitions, $c_{ij} = b_i + e_{ij}$. Let c_i be c_{ij}, where machine j is assigned to execute task i. The makespan for the complete schedule is then defined as $max_{t_i \in K}(c_i)$ [17]. Makespan is a measure of the throughput of the HC system, and does not measure the quality of service imparted to an individual task.

Recall from Section 1, in on-line 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 $i \geq 0$. 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 did not start executing, i.e., $M_k = \{t_j \mid \tau_{k-1} \leq a_j < \tau_k\} \cup \{t_j \mid a_j < \tau_{k-1}, b_j > \tau_k\}$. The waiting time for task t_j is defined as $b_j - a_j$. Let $\bar{c}_j$ be the completion time of task t_j if it is the only task that is executing on the system. The sharing penalty (ρ_j) for the task t_j is defined as $(c_j - \bar{c}_j)$. The average sharing penalty for the tasks in the set K is given by $[\sum_{t_j \in K} \rho_j] / \mid K \mid$. The average sharing penalty for a set of tasks mapped by a given heuristic is an indication of the heuristic's ability to minimize the effects of contention among different tasks in the set. It therefore indicates quality of service provided to an individual task, as gauged by the wait incurred by the task before it begins and the time to perform the actual computation. Other performance metrics are considered in [13].

4. Mapping Heuristics

4.1. Overview

In the on-line 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, 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 on-line mode so that a task need not wait until the next mapping event to begin its execution.

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, because the 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 the heuristic is negligibly small as compared to the average task execution time.

Both on-line 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.,[7, 11, 19]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [14].) These estimates can be supplied before a task is submitted for execution, or at the time it is submitted. (The use of some of the heuristics studied here in a static environment is discussed in [4].)

The ready time of a machine is quantified by the earliest time that machine is going to be ready after completing the execution of the tasks that are currently assigned to it. It is assumed that each time a task t_i completes on a machine m_j a report is sent to the mapper. Because the heuristics presented here are dynamic, the expected machine ready times are based on a combination of actual task execution times and estimated expected task execution times. The experiments presented in Section 6 model this situation using simulated actual values for the execution times of the tasks that have already finished their execution. Also, all heuristics examined here operate in a centralized fashion on a dedicated suite of machines; i.e., the mapper controls the execution of all jobs on all machines in the suite. It is also assumed that the mapping heuristic is being run on a separate machine.

4.2. On-line mode mapping heuristics

The MCT (minimum completion time) 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 [6]. The MCT heuristic is used as a benchmark for the on-line 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 MET (minimum execution time) 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) [6]). 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 SA (switching algorithm) heuristic 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 MET at the expense of load balance until a given threshold and then use 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 ready time over all machines in the suite be r_{max}, and the minimum 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 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 reaches π_l, the SA heuristic switches back to using the MCT heuristic for mapping the tasks and the cycle continues.

As an example of functioning of the SA with lower and upper limits of 0.6 and 0.9, respectively, for $|K| = 1000$, the SA switched between the MET and the MCT two times, assigning 715 tasks using the MCT. For $|K| = 2000$, 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 an MET assignment in a highly loaded system will bring a smaller decrease in load balance index than when the same assignment is made in a lightly loaded system. Therefore many more MET assignments can be made in a highly loaded system before the load balance index falls below the lower threshold.

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 KPB (k-percent best) heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the $(km/100)$ best machines based on the execution times for the task, where $100/m \leq k \leq 100$. 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 as much as matching of 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 lacks 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 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 SA have 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 task, the KPB will cause much machine idle time compared to the MCT, and can result in much poorer performance.

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

$O(\frac{km}{100})$ time, i.e., $O(m)$ time to determine the machine assignment. Overall the heuristic takes $O(m \log m)$ time.

The OLB (opportunistic load balancing) heuristic assigns the task to the machine that becomes ready next. It does not consider the execution time of the task when mapping it onto a 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 [21].

4.3. Batch mode mapping heuristics

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 except when all machines are busy. When all machines are busy, all scheduled mapping events that precede the one before the expected ready time of the machine that finishes earliest are canceled.

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 $\mid M_i \mid$ larger than or equal to a predetermined arbitrary number κ, or (b) all tasks 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.

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. In the complexity analysis, 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 Figure 1 is from SmartNet [6]. In Figure 1, let r_j denote the expected time 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 rows of the c matrix. 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 with 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 that any assignment could. 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. 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.

```
(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
```

Figure 1. The Min-min heuristic.

The initialization of the c matrix in Line (1) to Line (3) 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 given in Figure 1. It is also from SmartNet [6]. 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. The matrix c and vector r are updated and the above process is repeated with tasks that have not yet been assigned a machine. 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 the cases where we have many more shorter tasks than the long 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 these cases. 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.

The Sufferage heuristic 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.

Figure 2 shows the Sufferage heuristic. The initialization phase in Lines (1) to (3) 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 the each machine assigned a new task. Perform the next iteration of the **do** loop beginning on Line (4) until all tasks have been mapped.

Table 1 shows a scenario in which the Sufferage will outperform the Min-min. Table 1 shows the expected execution time values for four tasks on four machines (all initially idle). In this particular case, the Min-min heuristic gives a makespan of 9.3 and the Sufferage heuristic gives a makespan of 7.8. Figure 3 gives a pictorial representation of the assignments made for the case in Table 1.

From the pseudo code given in Figure 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

	m_0	m_1	m_2	m_3
t_0	4	4.8	13.4	5
t_1	5	8.2	8.8	8.9
t_2	5.5	6.8	9.4	9.3
t_3	5.2	6	7.8	10.8

Table 1. An example expected execution time matrix that illustrates the situation where the Sufferage heuristic outperforms the Min-min heuristic.

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 task are assigned in the first execution of the **for** statement so that $T(S) = O(Sm)$. Let $\underline{\omega}$ be a number quantifying the extent of contention among the tasks for the different machines. The running time of Sufferage heuristic can then be given as $O(\omega Sm)$ time, where $1 \leq \omega \leq S$. 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).

```
(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 an arbitrary order)
(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
```

Figure 2. The Sufferage heuristic.

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).

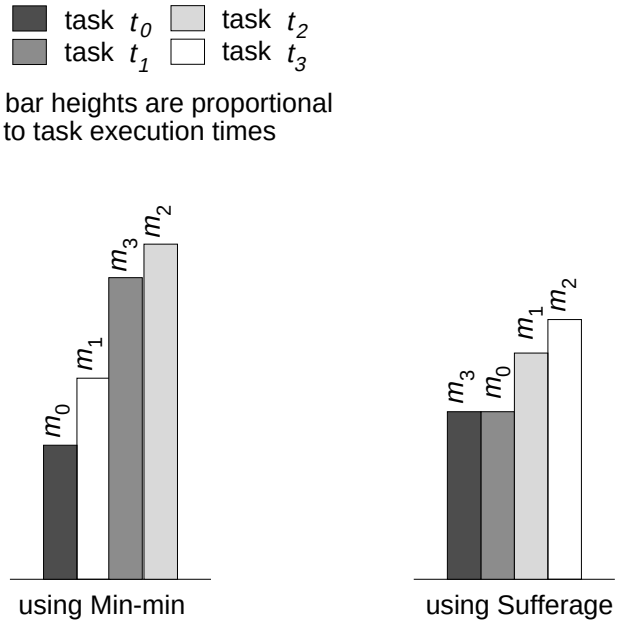

Figure 3. An example scenario (based on Table 1) where the Sufferage gives a shorter makespan than the Min-min.

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 set to 10. 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 Figure 1 is multiplied by the corresponding value for $\frac{1}{\zeta}$. 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 Figure 1.

Similarly, 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 Figure 2 is multiplied by ζ.

5. Simulation Procedure

The mappings are simulated using a discrete event simulator. 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 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 [2, 16]. 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 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 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. In addition to the consistent and inconsistent classes, a semi-consistent class could also be defined. A semi-consistent ETC matrix is characterized by a consistent sub-matrix. In the semi-consistent ETC matrices used here, 50% of the tasks and 25% of the machines define a consistent sub-matrix. Furthermore, it is assumed that for a

particular task the execution times that fall within the consistent sub-matrix are smaller than those that fall out. This assumption is justified because the machines that perform consistently better than the others for some tasks are more likely to be very much faster for those tasks than very much slower.

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

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

2. Let $\underline{\Gamma_m}$ be an arbitrary constant quantifying machine heterogeneity, being smaller for low machine heterogeneity. Let $\underline{N_m}$ 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 $\underline{q}[0..(t_{max}-1)]$.

4. Generate the ETC matrix, $\underline{e}[0..(t_{max}-1), 0..(m_{max}-1)]$ 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 semi-consistent matrix could be generated by sorting the execution times for a random subset of the tasks on a random subset of machines. 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.

6. Experimental Results and Discussion

6.1. Overview

The experimental evaluation of the heuristics is performed in three parts. In the first part, the on-line 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 on-line 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 semi-consistent cases, because these correspond to some of the currently expected MSHN environments. A Poisson distribution is used to generate the task arrivals. For each value of $|K|$, tasks are mapped under two different 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 complete when the last task in the set arrives. Similarly, λ_l is chosen to be low enough to allow at least 90% of the tasks to complete when the last task in the set arrives. The MCT heuristic is used in this standardization. Unless otherwise stated, the task arrival rate is set to λ_h. λ_l is more likely to represent an HC system where the task arrival is characterized by little burstiness; no particular group of tasks arrives in a much shorter span of time than some other group having same number of tasks. λ_h is supposed to characterize the arrivals in an HC system where a large group of tasks arrives in a much shorter time than some other group having same number of tasks; e.g., in this case a burst of $|K|$ tasks.

Example comparisons are discussed in Subsections 6.2 to 6.4. 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. More general conclusions about the heuristics' performance is in Section 7. Comparisons for a larger set of performance metrics are given in [13].

6.2. Comparisons of the on-line mode heuristics

Unless otherwise stated, the on-line 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 Figure 4, on-line mode heuristics are compared based on makespan for inconsistent HiHi heterogeneity. From Figure 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. The chosen machine would give the smallest completion time as compared to other machines in the set.

Figure 5 compares the on-line mode heuristics using average sharing penalty. Once again, the KPB heuristic per-

forms best. However, the margin of improvement is smaller than that for the makespan. It is evident that the KPB provides maximum throughput (system oriented performance metric) and minimum average sharing penalty (application oriented performance metric).

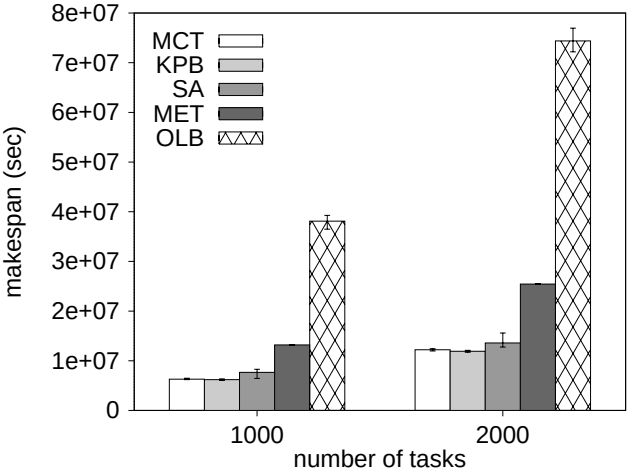

Figure 4. Makespan for the on-line heuristics for inconsistent HiHi heterogeneity.

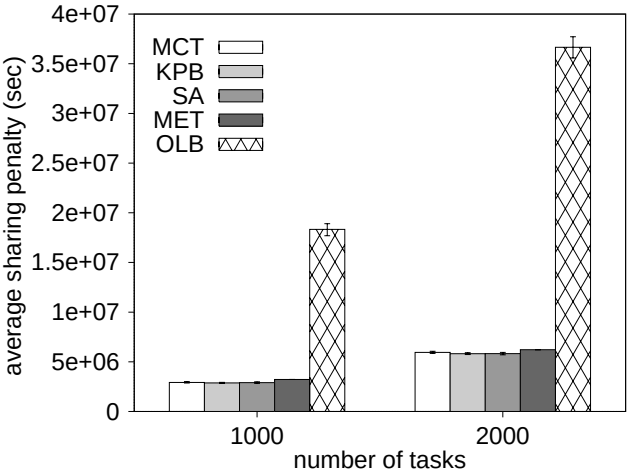

Figure 5. Average sharing penalty of the on-line heuristics for inconsistent HiHi heterogeneity.

Figure 6 compares the makespans of the different on-line heuristics for semi-consistent HiHi ETC matrices. Figure 7 compares the average sharing penalties of the different on-line heuristics. As shown in Figures 4 and 6 the relative performance of the different on-line heuristics is impacted by the degree of consistency of the ETC matrices.

For the semi-consistent type of heterogeneity, machines within a particular subset perform tasks that lie within a particular subset faster than other machines. From Figure 6, it can be observed that for semi-consistent ETC matrices, the MET heuristic performs the worst. For the semi-consistent 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 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 5. However a further doubling of k brings down the makespan by a factor of only about 1.2.

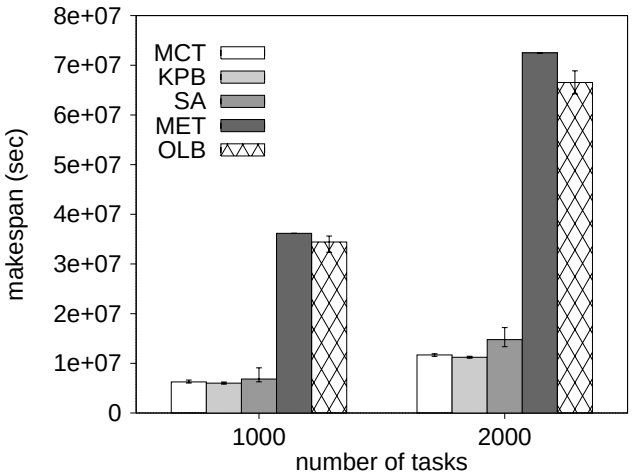

Figure 6. Makespan of the on-line heuristics for semi-consistent HiHi heterogeneity.

6.3. Comparisons of the batch mode heuristics

Figures 8 and 9 compare the batch mode heuristics based on makespan and average sharing penalty, respectively. 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 seconds. This value was empirically found to optimize makespan over other values. From Figure 8, 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). However, for average sharing penalty, the Min-min heuristic outperforms the other heuristics (Figure 9). 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.

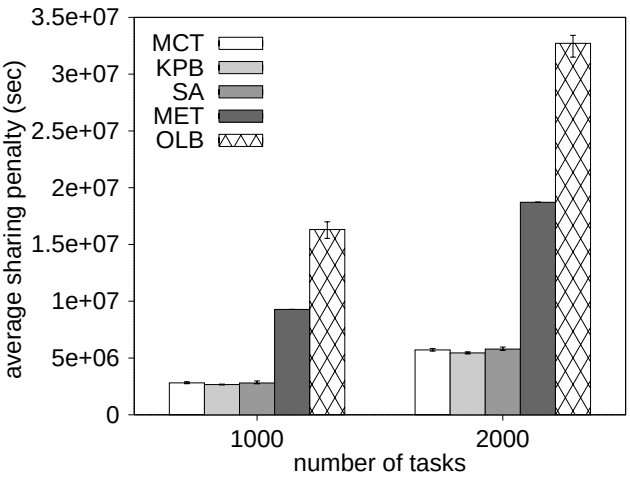

Figure 7. Average sharing penalty of the on-line heuristics for semi-consistent HiHi heterogeneity.

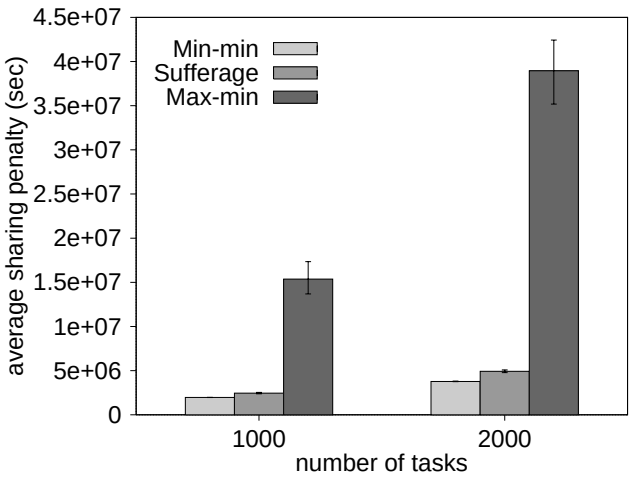

Figure 9. Average sharing penalty of the batch heuristics for the regular time interval strategy and inconsistent HiHi heterogeneity.

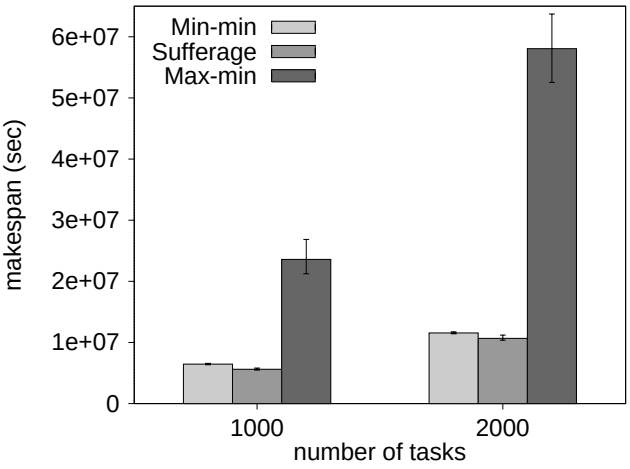

Figure 8. Makespan of the batch heuristics for the regular time interval strategy and inconsistent HiHi heterogeneity.

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 4.3 for explaining better expected performance for the Min-min, it can be seen that the Max-min assignments change the machine ready time status by a larger amount than the Min-min assignments do. (The Sufferage also does not necessarily schedule the tasks that finish later first.) 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. In general, the percentage of tasks assigned their first choice is likely to be lower for the Max-min than for other batch mode heuristics. It might be expected that a larger 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.

Figure 10 compares the makespan of the batch mode heuristics for semi-consistent HiHi heterogeneity. The comparison of the same heuristics for the same parameters is shown in Figure 11 with respect to average sharing penalty. Results for both average sharing penalty and makespan for semi-consistent HiHi are similar to those for inconsistent HiHi.

The impact of aging on batch mode heuristics is shown in Figures 12 and 13. From Figures 12 and 13, three observations are in order. First, the Max-min heuristic benefits most from the aging scheme. Second, the makespan and the average sharing penalty given by the Sufferage heuristic change negligibly when aging scheme is applied. Third, even though aging schemes are meant to reduce starvation of tasks (as gauged by average sharing penalty), they also reduce the makespan.

The fact that the Max-min benefits most from the aging scheme can be explained using the reasoning given in the discussion on starvation in Subsection 4.3. The larger the number (say N_{new}) of newly arriving tasks between the mapping events τ_i and τ_{i+1}, the larger the probability that some of the tasks mapped at mapping event τ_i, or earlier, will be starved (due to more competing tasks). The Max-min heuristic schedules tasks that finish later first. As mapping events are not scheduled if machines are busy, two successive mapping events in the Max-min are likely to be separated by a larger time duration than those in the Sufferage or the Min-min. The value of N_{new} is therefore likely to

be larger in the Max-min schedules, and starvation is more likely to occur. Consequently, aging schemes would make greater difference to the Max-min schedules: the tasks that finish sooner are much more likely to be scheduled before the tasks that finish later in the Max-min with aging than in the Max-min without aging. In contrast to the Max-min (or the Min-min) operation, the Sufferage heuristic optimizes a machine assignment only over the tasks that are contending for that particular machine. This reduces the probability of competition between the "older" tasks and the new arrivals, which in turn reduces the need for an aging scheme, or the improvement in schedule in case aging is implemented.

Figures 14, 15, 16, and 17 show the results of repeating the above experiments with a batch count mapping strategy for a batch size of 40. This particular batch size was found to give an optimum value of the makespan. Figure 14 compares regular time interval strategy and fixed count strategy on the basis of makespans given by different heuristics for inconsistent HiHi heterogeneity. In Figure 15, the average sharing penalties of the same heuristics for the same parameters are compared. It can be seen that the fixed count approach gives essentially the same 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 $\mid K \mid = 1000$, and to about 50% for $\mid K \mid = 2000$ 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. A "good" value of κ in fixed count strategy is neither too small to allow only a limited optimization of machine assignment nor too large to subject the tasks carried over from the previous mapping events to a possibly defeating competition with the new or recent arrivals. Figures 16 and 17 show the makespan and the average sharing penalty given for the semi-consistent case. These results show that, for the Sufferage and the Min-min, the regular time interval approach gives slightly better results than the fixed count approach. For the Max-min, however, the fixed count approach gives better performance.

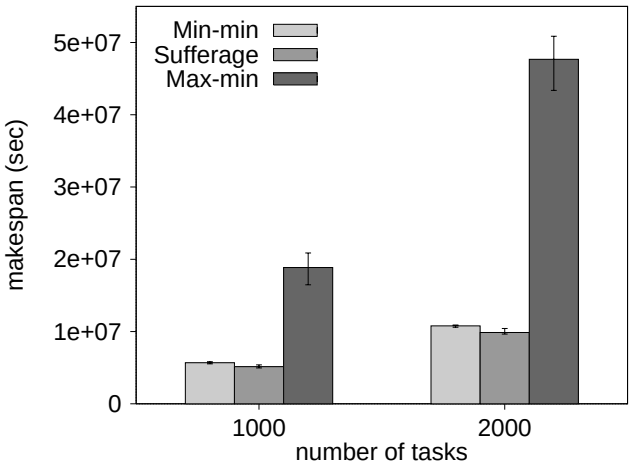

Figure 10. Makespan of the batch heuristics for the regular time interval strategy and semi-consistent HiHi heterogeneity.

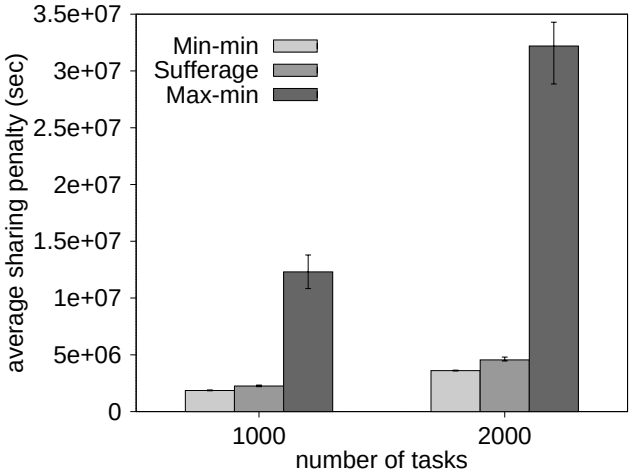

Figure 11. Average sharing penalty of the batch heuristics for the regular time interval strategy and semi-consistent HiHi heterogeneity.

6.4. Comparing on-line and batch heuristics

In Figure 18, two on-line 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 the higher arrival rate and larger $\mid K \mid$, batch heuristics are superior to on-line heuristics. This is because the number of tasks waiting to begin execution is likely to be larger in above circumstances than in any other, which in turn means that rescheduling is likely to improve many more mappings in such a system. The on-line heuristics consider only one task when they try to optimize machine assignment, and do not reschedule. Recall that the mapping heuristics use a combination of expected and actual task execution times to compute machine ready times. The on-line 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 100-th arriving task might find that 70 previously arrived tasks have completed. At a higher arrival rate, only 20 tasks might have completed when the 100-th task arrived. The above observation is borne out in Figure 19, which shows that the relative performance difference between on-line and batch heuristics decreases with a decrease in arrival rate. Given the observation that the KPB and the Sufferage per-

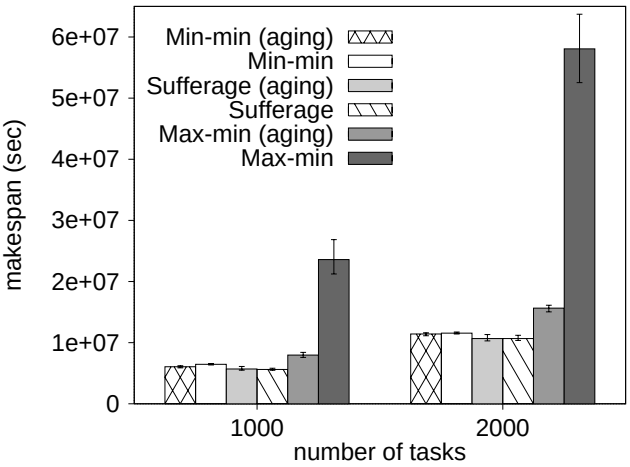

Figure 12. Makespan for the batch heuristics for the regular time interval strategy with and without aging for inconsistent HiHi heterogeneity.

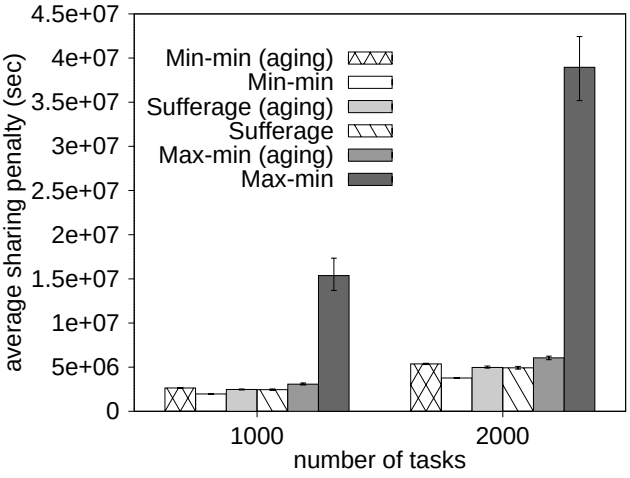

Figure 13. Average sharing penalty of the batch heuristics for the regular time interval strategy with and without aging for inconsistent HiHi heterogeneity.

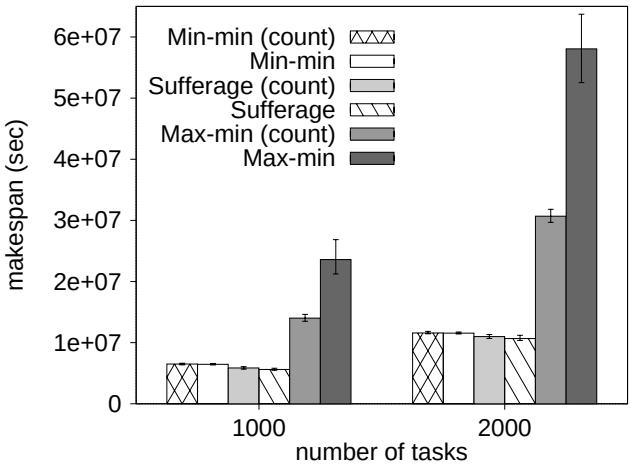

Figure 14. Comparison of the makespans given by the fixed count mapping strategy and the regular time interval strategy for inconsistent HiHi heterogeneity.

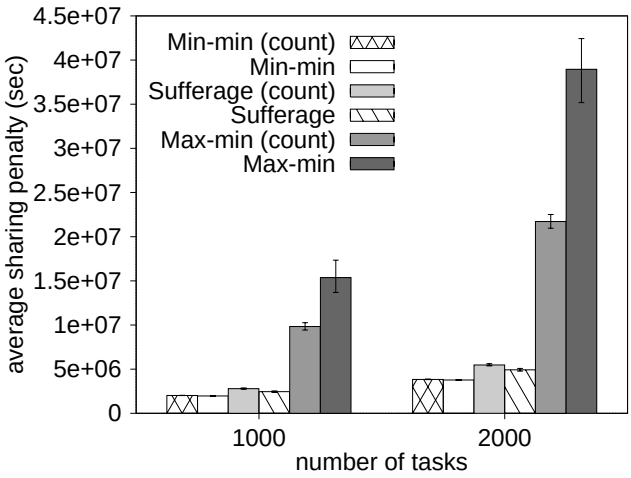

Figure 15. Comparison of the average sharing penalty given by the fixed count mapping strategy and the regular time interval strategy for inconsistent HiHi heterogeneity.

form almost similarly at this low arrival rate, it might be better to use the KPB heuristic because of its smaller computation time. Moreover, Figures 18 and 19 show that the makespan values for all heuristics are larger for lower arrival rate. This is attributable to the fact that at lower arrival rates, a larger fraction of a task's completion time is determined by its beginning time.

7. 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 on-line mode heuristics and three batch mode heuristics were studied.

In the on-line mode, for both the semi-consistent and the inconsistent types of HiHi heterogeneity, the KPB heuristic outperformed the other heuristics on all performance metrics (however, the KPB was only slightly better than the MCT). The average sharing penalty gains were smaller than the makespan ones. The KPB can provide good system oriented performance (e.g., minimum makespan) and at the same time provide good application oriented performance (e.g., low average sharing penalty). The relative performance of the OLB and the MET with respect to the makespan reversed when the heterogeneity was changed

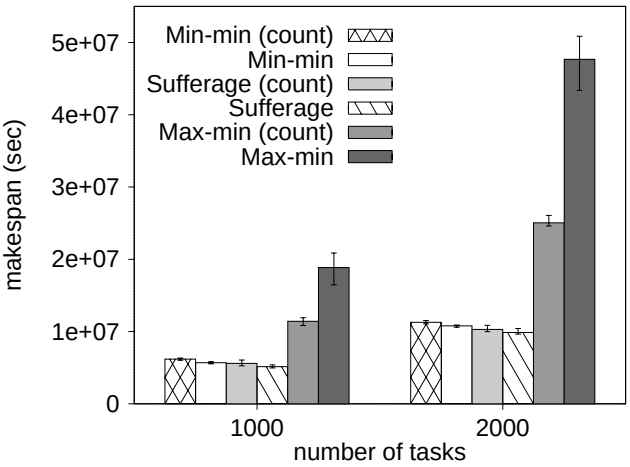

Figure 16. Comparison of the makespan given by the fixed count mapping strategy and the regular time interval strategy for semi-consistent HiHi heterogeneity.

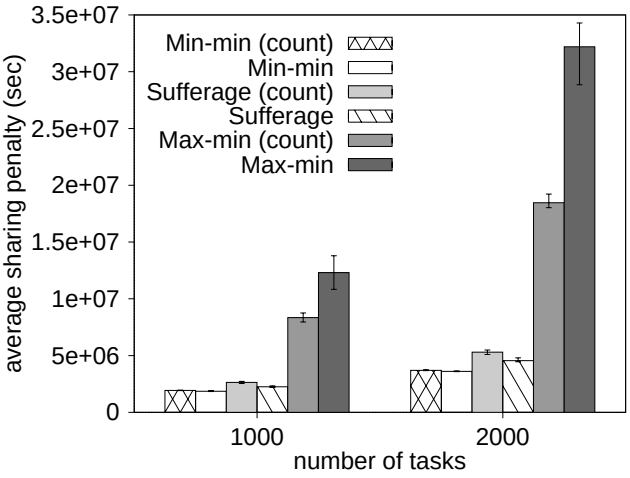

Figure 17. Comparison of the average sharing penalty given by the fixed count mapping strategy and the regular time interval strategy for semi-consistent HiHi heterogeneity.

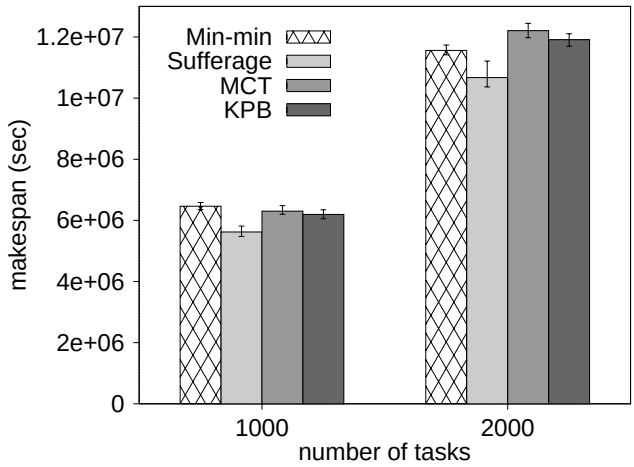

Figure 18. Comparison of the makespan given by batch heuristics (regular time interval strategy) and on-line heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_h.

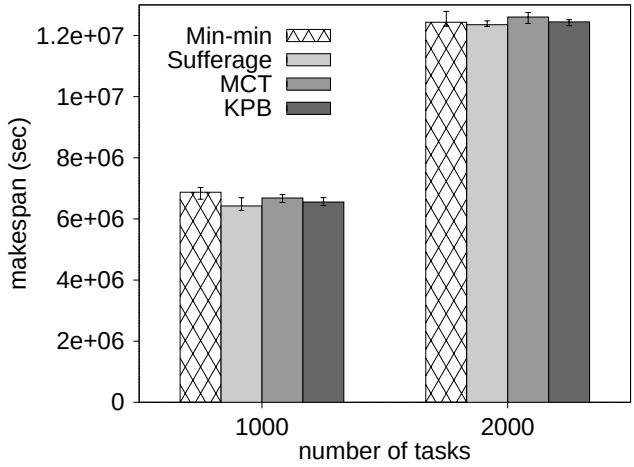

Figure 19. Comparison of the makespan given by batch heuristics (regular time interval strategy) and on-line heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_l.

from the semi-consistent to the inconsistent. The OLB did better than the MET for the semi-consistent case.

In the batch mode, for the semi-consistent and the inconsistent types of HiHi heterogeneity, the Min-min heuristic outperformed the Sufferage and Max-min heuristics in the average sharing penalty. However, the Sufferage performed the best with respect to makespan for both the semi-consistent and the inconsistent types of HiHi heterogeneity (though, the Sufferage was only slightly better than the Min-min).

The batch heuristics are likely to give a smaller makespan than the on-line 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 heuristics is likely 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, (b) the requirement to optimize system oriented or application oriented performance metrics (e.g., optimizing makespan versus optimizing average sharing penalty), and (c) the arrival rate of the tasks. Thus, the choice of the heuristic which is best to use will be a function of such factors. Therefore, it is important to include a set of heuristics in a resource

management system for HC environments, and then use the heuristic that is most appropriate for a given situation (as will be done in the Scheduling Advisor for MSHN).

Acknowledgments: The authors thank Taylor Kidd, Surjamukhi Chatterjea, and Tracy D. Braun for their valuable comments and suggestions.

References

[1] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predications," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 79–87.

[2] R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance*, Master's thesis, Department of Computer Science, Naval Postgraduate School, 1997 (D. Hensgen, advisor).

[3] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, 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," *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1998, pp. 330–335 (included in the proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, 1998).

[4] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, 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," *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, Apr. 1999, to appear.

[5] I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Fransisco, CA, 1999.

[6] 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 multi-user, heterogeneous, computing environments with SmartNet," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 184–199.

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

[8] D. Hensgen, T. Kidd, M. C. Schnaidt, D. St. John, H. J. Siegel, T. D. Braun, M. Maheshwaran, S. Ali, J-K. Kim, C. Irvine, T. Levin, R. Wright, R. F. Freund, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, "An overview of MSHN: A Management System for Heterogeneous Networks," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, to appear.

[9] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, Vol. 24, No. 2, Apr. 1977, pp. 280–289.

[10] M. A. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 70–78.

[11] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, Vol. 6, No. 3, July-Sep. 1998, pp. 42–51.

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

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. 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, in preparation, 1999.

[14] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," in *Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster, ed., John Wiley, New York, NY, scheduled to appear in 1999.

[15] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Adaptive load sharing in heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, Vol. 9, No. 4, Aug. 1990, pp. 331–346.

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

[17] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[18] H. G. Rotithor, "Taxonomy of dynamic task scheduling schemes in distributed computing systems," *IEE Proceedings on Computer and Digital Techniques*, Vol. 141, No. 1, Jan. 1994, pp. 1–10.

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

[20] V. Suresh and D. Chaudhuri, "Dynamic rescheduling–A survey of research," *International Journal of Pro-*

duction Economics, Vol. 32, No. 1, Aug. 1993, pp. 53–63.

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

Biographies

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. In 1990, he received a BSc degree in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an MSEE degree in 1994 and a PhD 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 MSEE 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 an MSEE degree 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 Aitchison College and Keynesian Institute of Management and Sciences, both in Lahore, Pakistan. He was also a Teaching Assistant at Purdue. Shoukat received his BS degree in electrical and electronic engineering from the University of Engineering and Technology, Lahore, Pakistan in 1996. 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 BS degrees in both electrical engineering and management from MIT, and the MA, MSE, and PhD degrees 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*. 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 PhD 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 re-routing 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 NOEMIX, a San Diego based startup 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 IPPS/SPDP. Freund won a Meritorious Civilian Service Award during his former career as a government scientist.