A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems

Tracy D. Braun[†], Howard Jay Siegel[†], Noah Beck[†], Ladislau L. Bölöni[‡], Muthucumaru Maheswaran[§], Albert I. Reuther[†], James P. Robertson[*], Mitchell D. Theys[†], Bin Yao[†], Debra Hensgen[◇], and Richard F. Freund[¶]

[†]School of Electrical and Computer Engineering
[‡]Department of Computer Sciences
Purdue University
West Lafayette, IN 47907 USA
{tdbraun, hj, noah, reuther, theys, yaob}
@ecn.purdue.edu, boloni@cs.purdue.edu

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

[*]Motorola
6300 Bridgepoint Parkway
Bldg. #3, MD: OE71
Austin, TX 78730 USA
robertso@ibmoto.com

[◇]Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5118 USA
hensgen@cs.nps.navy.mil

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

Abstract

Heterogeneous computing (HC) environments are well suited to meet the computational demands of large, diverse groups of tasks (i.e., a meta-task). The problem of mapping (defined as matching and scheduling) these tasks onto the machines of an HC environment has been shown, in general, to be NP-complete, requiring the development of heuristic techniques. Selecting the best heuristic to use in a given environment, however, remains a difficult problem, because comparisons are often clouded by different underlying assumptions in the original studies of each heuristic. Therefore, a collection of eleven heuristics from the literature has been selected, implemented, and analyzed under one set of common assumptions. The eleven heuristics examined are Opportunistic Load Balancing, User-Directed Assignment, Fast Greedy, Min-min, Max-min, Greedy, Genetic Algorithm, Simulated Annealing, Genetic Simulated Annealing, Tabu, and A. This study provides one even basis for comparison and insights into circumstances where one technique will outperform another. The evaluation procedure is specified, the heuristics are defined, and then selected results are compared.*

This research was supported in part by the DARPA/ITO Quorum Program under NPS subcontract numbers N62271-98-M-0217 and N62271-98-M-0448. Some of the equipment used was donated by Intel.

1. Introduction

Mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different high-performance machines, interconnected with high-speed links to execute different computationally intensive applications that have diverse computational requirements [10, 18, 24]. The general problem of mapping (i.e., matching and scheduling) tasks to machines has been shown to be NP-complete [8, 15]. Heuristics developed to perform this mapping function are often difficult to compare because of different underlying assumptions in the original studies of each heuristic [3]. Therefore, a collection of eleven heuristics from the literature has been selected, implemented, and compared by simulation studies under one set of common assumptions.

To facilitate these comparisons, certain simplifying assumptions were made. Let a meta-task be defined as a collection of independent tasks with no data dependencies (a given task, however, may have subtasks and dependencies among the subtasks). For this case study, it is assumed that static (i.e., off-line or predictive) mapping of meta-tasks is being performed. (In some systems, all tasks and subtasks in a meta-task, as defined above, are referred to as just tasks.)

It is also assumed that each machine executes a single task at a time, in the order in which the tasks arrived. Because there are no dependencies among the tasks, scheduling is simplified, and thus the resulting solutions of the mapping heuristics focus more on finding an efficient matching of tasks to machines. It is also assumed that the size of the meta-task (number of tasks to execute), $\underline{t}$, and the number of machines in the HC environment, $\underline{m}$, are static and known *a priori*.

Section 2 defines the computational environment parameters that were varied in the simulations. Descriptions of the eleven mapping heuristics are found in Section 3. Section 4 examines selected results from the simulation study. A list of implementation parameters and procedures that could be varied for each heuristic is presented in Section 5.

This research was supported in part by the DARPA/ITO Quorum Program project called MSHN (Management System for Heterogeneous Networks) [13]. MSHN is a collaborative research effort among the Naval Postgraduate School, NOEMIX, Purdue University, and the University of Southern California. 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 requested qualities of service. The heuristics developed in this paper or their derivatives may be included in the Scheduling Advisor component of the MSHN prototype.

2. Simulation Model

The eleven static mapping heuristics were evaluated using simulated execution times for an HC environment. Because these are static heuristics, it is assumed that an accurate estimate of the expected execution time for each task on each machine is known prior to execution and contained within an ETC (expected time to compute) matrix. One row of the ETC matrix contains the estimated execution times for a given task on each machine. Similarly, one column of the ETC matrix consists of the estimated execution times of a given machine for each task in the meta-task. Thus, $ETC(i, j)$ is the estimated execution time for task i on machine j. (These times are assumed to include the time to move the executables and data associated with each task to the particular machine when necessary.) The assumption that these estimated expected execution times are known is commonly made when studying mapping heuristics for HC systems (e.g., [11, 16, 25]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [14, 24].)

For the simulation studies, characteristics of the ETC matrices were varied in an attempt to represent a variety of possible HC environments. The ETC matrices used were generated using the following method. Initially, a $t \times 1$ baseline column vector, $\underline{B}$, of floating point values is created. Let $\underline{\phi_b}$ be the upper-bound of the range of possible values within the baseline vector. The baseline column vector is generated by repeatedly selecting a uniform random number, $\underline{x_b^i} \in [1,\ \phi_b)$, and letting $B(i) = x_b^i$ for $0 \leq i < t$. Next, the rows of the ETC matrix are constructed. Each element $ETC(i, j)$ in row i of the ETC matrix is created by taking the baseline value, $B(i)$, and multiplying it by a uniform random number, $\underline{x_r^{i,j}}$, which has an upper-bound of $\underline{\phi_r}$. This new random number, $x_r^{i,j} \in [1,\ \phi_r)$, is called a row multiplier. One row requires m different row multipliers, $0 \leq j < m$. Each row i of the ETC matrix can be then described as $ETC(i, j) = B(i) \times x_r^{i,j}$, for $0 \leq j < m$. (The baseline column itself does not appear in the final ETC matrix.) This process is repeated for each row until the $m \times t$ ETC matrix is full. Therefore, any given value in the ETC matrix is within the range $[1,\ \phi_b \times \phi_r)$.

To evaluate the heuristics for different mapping scenarios, the characteristics of the ETC matrix were varied based on several different methods from [2]. The amount of variance among the execution times of tasks in the meta-task for a given machine is defined as task heterogeneity. Task heterogeneity was varied by changing the upper-bound of the random numbers within the baseline column vector. High task heterogeneity was represented by $\phi_b = 3000$ and low task heterogeneity used $\phi_b = 100$. Machine heterogeneity represents the variation that is possible among the execution times for a given task across all the machines. Machine heterogeneity was varied by changing the upper-bound of the random numbers used to multiply the baseline values. High machine heterogeneity values were generated using $\phi_r = 1000$, while low machine heterogeneity values used $\phi_r = 10$. These heterogeneous ranges are based on one type of expected environment for MSHN.

To further vary the ETC matrix in an attempt to capture more aspects of realistic mapping situations, different ETC matrix consistences were used. An ETC matrix is said to be consistent if whenever a machine j executes any task i faster than machine k, then machine j executes all tasks faster than machine k [2]. Consistent matrices were generated by sorting each row of the ETC matrix independently. In contrast, inconsistent matrices characterize the situation where machine j is faster than machine k for some tasks, and slower for others. These matrices are left in the unordered, random state in which they were generated. In between these two extremes, semi-consistent matrices represent a partial ordering among the machine/task execution times. For the semi-consistent matrices used here, the row elements in column positions $\{0, 2, 4, \ldots\}$ of row i are extracted, sorted, and replaced in order, while the row elements in column positions $\{1, 3, 5, \ldots\}$ remain unordered. (That is, the even columns are consistent and the odd columns are inconsistent.)

Sample *ETC* matrices for the four inconsistent heterogeneous permutations of the characteristics listed above are shown in Tables 1 through 4. (Other probability distributions for *ETC* values, including an exponential distribution and a truncated Gaussian [1] distribution, were also investigated, but not included in the results discussed here.) All results in this study used *ETC* matrices that were of size $t = 512$ tasks by $m = 16$ machines. While it was necessary to select some specific parameter values to allow implementation of a simulation, the characteristics and techniques presented here are completely general. Therefore, if these parameter values do not apply to a specific situation of interest, researchers may substitute in their own ranges, distributions, matrix sizes, etc., and the evaluation software of this study will still apply.

3. Heuristic Descriptions

The definitions of the eleven static meta-task mapping heuristics are provided below. First, some preliminary terms must be defined. Machine availability time, $mat(j)$, is the earliest time a machine j can complete the execution of all the tasks that have previously been assigned to it. Completion time, $ct(i, j)$, is the machine availability time plus the execution time of task i on machine j, i.e., $ct(i, j) = mat(j) + ETC(i, j)$. The performance criterion used to compare the results of the heuristics is the maximum value of $ct(i, j)$, for $0 \le i < t$ and $0 \le j < m$, for each mapping, also known as the makespan [19]. Each heuristic is attempting to minimize the makespan (i.e., finish execution of the meta-task as soon as possible).

The descriptions below implicitly assume that the machine availability times are updated after each task is mapped. For cases when tasks can be considered in an arbitrary order, the order in which the tasks appeared in the *ETC* matrix was used. Some of the heuristics listed below had to be modified from their original implementation to better handle the scenarios under consideration.

For many of the heuristics, there are control parameter values and/or control function specifications that can be selected for a given implementation. For the studies here, such values and specifications were selected based on experimentation and/or information in the literature. These parameters and functions are mentioned in Section 5.

OLB: Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected execution time on that machine [1, 9, 10].

UDA: In contrast to OLB, User-Directed Assignment (UDA) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability [1]. UDA is sometimes referred to as Limited Best Assignment (LBA), as in [1, 9].

Fast Greedy: Fast Greedy assigns each task, in arbitrary order, to the machine with the minimum completion time for that task [1].

Min-min: The Min-min heuristic begins with the set U of all unmapped tasks. Then, the set of minimum completion times, $M = \{m_i : m_i = \min_{0 \le j < m}(ct(i, j))$, for each $i \in U\}$, is found. Next, the task with the overall *minimum* completion time from M is selected and assigned to the corresponding machine (hence the name Min-min). Lastly, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., $U = \emptyset$) [1, 9, 15].

Intuitively, Min-min attempts to map as many tasks as possible to their first choice of machine (on the basis of completion time), under the assumption that this will result in a shorter makespan. Because tasks with shorter execution times are being mapped first, it was expected that the percentage of tasks that receive their first choice of machine would generally be higher for Min-min than for Max-min (defined next), and this was verified by data recorded during the simulations.

Max-min: The Max-min heuristic is very similar to Min-min. The Max-min heuristic also begins with the set U of all unmapped tasks. Then, the set of minimum completion times, $M = \{m_i : m_i = \min_{0 \le j < m}(ct(i, j))$, for each $i \in U\}$, is found. Next, the task with the overall *maximum* completion time from M is selected and assigned to the corresponding machine (hence the name Max-min). Lastly, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., $U = \emptyset$) [1, 9, 15].

The motivation behind Max-min is to attempt to minimize the penalties incurred by delaying the scheduling of long-running tasks. Assume that the meta-task being mapped has several tasks with short execution times, and a small quantity of tasks with very long execution times. Mapping the tasks with the longer execution times to their best machines first allows these tasks to be executed concurrently with the remaining tasks (with shorter execution times). This concurrent execution of long and short tasks can be more beneficial than a Min-min mapping where all of the shorter tasks would execute first, and then a few longer running tasks execute while several machines sit idle. The assumption here is that with Max-min the tasks with shorter execution times can be mixed with longer tasks and evenly distributed among the machines, resulting in better machine utilization and a better meta-task makespan.

Greedy: The Greedy heuristic is literally a combination of the Min-min and Max-min heuristics. The Greedy heuristic performs both of the Min-min and Max-min heuristics, and uses the better solution [1, 9].

GA: Genetic Algorithms (GAs) are a popular technique used for searching large solution spaces (e.g., [25, 27]). The version of the heuristic used for this study was adapted from [27] for this particular solution space. Figure 1 shows the steps in a general Genetic Algorithm.

The Genetic Algorithm implemented here operates on a population of 200 chromosomes (possible mappings) for a given meta-task. Each chromosome is a $t \times 1$ vector, where position i ($0 \leq i < t$) represents task i, and the entry in position i is the machine to which the task has been mapped. The initial population is generated using two methods: (a) 200 randomly generated chromosomes from a uniform distribution, or (b) one chromosome that is the Min-min solution and 199 random solutions (mappings). The latter method is called seeding the population with a Min-min chromosome. The GA actually executes eight times (four times with initial populations from each method), and the best of the eight mappings is used as the final solution.

After the generation of the initial population, all of the chromosomes in the population are evaluated (i.e., ranked) based on their fitness value (i.e., makespan), with a smaller fitness value being a better mapping. Then, the main loop in Figure 1 is entered and a rank-based roulette wheel scheme [26] is used for selection. This scheme probabilistically generates new populations, where better mappings have a higher probability of surviving to the next generation. Elitism, the property of guaranteeing the best solution remains in the population [20], was also implemented.

Next, the crossover operation selects a pair of chromosomes and chooses a random point in the first chromosome. For the sections of both chromosomes from that point to the end of each chromosome, crossover exchanges machine assignments between corresponding tasks. Every chromosome is considered for crossover with a probability of 60%.

After crossover, the mutation operation is performed. Mutation randomly selects a task within the chromosome, and randomly reassigns it to a new machine. Both of these random operations select values from a uniform distribution. Every chromosome is considered for mutation with a probability of 40%.

Finally, the chromosomes from this modified population are evaluated again. This completes one iteration of the GA. The GA stops when any one of three conditions are met: (a) 1000 total iterations, (b) no change in the elite chromosome for 150 iterations, or (c) all chromosomes converge. If no stopping criteria is met, the loop repeats, beginning with the selection of a new population. The stopping criteria that usually occurred in testing was no change in the elite chromosome in 150 iterations.

SA: Simulated Annealing (SA) is an iterative technique that considers only one possible solution (mapping) for each meta-task at a time. This solution uses the same representation for a solution as the chromosome for the GA.

SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space (e.g., [6, 17, 21]). This probability is based on a system temperature that decreases for each iteration. As the system temperature "cools," it is more difficult for currently poorer solutions to be accepted. The initial system temperature is the makespan of the initial mapping.

The specific SA procedure implemented here is as follows. The initial mapping is generated from a uniform random distribution. The mapping is mutated in the same manner as the GA, and the new makespan is evaluated. The decision algorithm for accepting or rejecting the new mapping is based on [6]. If the new makespan is better, the new mapping replaces the old one. If the new makespan is worse (larger), a uniform random number $z \in [0,\ 1)$ is selected. Then, z is compared with y, where

$$y = \frac{1}{1 + e^{\left(\frac{\text{old makespan}-\text{new makespan}}{\text{temperature}}\right)}}. \qquad (1)$$

If $z > y$ the new (poorer) mapping is accepted, otherwise it is rejected, and the old mapping is kept.

Notice that for solutions with similar makespans (or if the system temperature is very large), $y \to 0.5$, and poorer solutions are more easily accepted. In contrast, for solutions with very different makespans (or if the system temperature is very small), $y \to 1$, and poorer solutions will usually be rejected.

After each mutation, the system temperature is decreased by 10%. This defines one iteration of SA. The heuristic stops when there is no change in the makespan for 150 iterations or the system temperature reaches zero. Most tests ended with no change in the makespan for 150 iterations.

GSA: The Genetic Simulated Annealing (GSA) heuristic is a combination of the GA and SA techniques [4, 23]. In general, GSA follows procedures similar to the GA outlined above. GSA operates on a population of 200 chromosomes, uses a Min-min seed in four out of eight initial populations, and performs similar mutation and crossover operations. However, for the selection process, GSA uses the SA cooling schedule and system temperature, and a simplified SA decision process for accepting or rejecting a new chromosomes. GSA also used elitism to guarantee that the best solution always remained in the population.

The initial system temperature for the GSA selection process was set to the average makespan of the initial population, and decreased 10% for each iteration. When a new (post-mutation, post-crossover, or both) chromosome is compared with the corresponding original chromosome, if the new makespan is less than the old makespan plus the system temperature, then the new chromosome is accepted. That is, if

$$\text{new makespan} < (\text{old makespan} + \text{temperature}) \quad (2)$$

is true, the new chromosome becomes part of the population. Otherwise, the original chromosome survives to the next iteration. Therefore, as the system temperature decreases, it is again more difficult for poorer solutions (i.e., longer makespans) to be accepted. The two stopping criteria used were either (a) no change in the elite chromosome in 150 iterations or (b) 1000 total iterations. Again, the most common stopping criteria was no change in the elite chromosome in 150 iterations.

Tabu: Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas [7, 12]. A solution (mapping) uses the same representation as a chromosome in the GA approach.

The implementation of Tabu search used here begins with a random mapping, generated from a uniform distribution. Starting with the first task in the mapping, task $i = 0$, each possible pair of tasks is formed, $(i,\ j)$ for $0 \leq i < t - 1$ and $i < j < t$. As each pair of tasks is formed, they exchange machine assignments. This constitutes a short hop. The intuitive purpose of a short hop is to find the nearest local minimum solution within the solution space. After each exchange, the new makespan is evaluated. If the new makespan is an improvement, the new mapping is accepted (a successful short hop), and the pair generation-and-exchange sequence starts over from the beginning ($i = 0$) of the new mapping. Otherwise, the pair generation-and-exchange sequence continues from its previous state, $(i,\ j)$. New short hops are generated until 1200 successful short hops have been made or all combinations of task pairs have been exhausted with no further improvement.

At this point, the final mapping from the local solution space search is added to the tabu list. The tabu list is a method of keeping track of the regions of the solution space that have already been searched. Next, a new random mapping is generated, and it must differ from each mapping in the tabu list by at least half of the machine assignments (a successful long hop). The intuitive purpose of a long hop is to move to a new region of the solution space that has not already been searched. The final stopping criterion for the heuristic is a total of 1200 successful hops (short and long combined). Then, the best mapping from the tabu list is the final answer.

A*: The final heuristic in the comparison study is known as the A* heuristic. A* has been applied to many other task allocation problems (e.g., [5, 16, 21, 22]). The technique used here is similar to [5].

A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions (a subset of tasks are assigned to machines), and leaf nodes represent final solutions (all tasks are assigned to machines). The partial solution of a child node has one more task mapped than the parent node. Call this additional task $\underline{a}$. Each parent node generates m children, one for each possible mapping of a. After a parent node has done this, the parent node is removed and replaced in the tree by the m children. Based on experimentation and a desire to keep execution time of the heuristic tractable, the maximum number of nodes in the tree at any one time is limited in this study to $\underline{n_{max}} = 1024$.

Each node, $\underline{n}$, has a cost function, $\underline{f(n)}$, associated with it. The cost function is an estimated lower-bound on the makespan of the best solution that includes the partial solution represented by node n. Let $\underline{g(n)}$ represent the makespan of the task/machine assignments in the partial solution of node n, i.e., $g(n)$ is the maximum of the machine availability times $(mat(j))$ based on the set of tasks that have been mapped to machines in node n's partial solution. Let $\underline{h(n)}$ be a lower-bound estimate on the difference between the makespan of node n's partial solution and the makespan for the best complete solution that includes node n's partial solution. Then, the cost function for node n is computed as

$$f(n) = g(n) + h(n). \quad (3)$$

Therefore, $f(n)$ represents the makespan of the partial solution of node n plus a lower-bound estimate of the time to execute the rest of the (unmapped) tasks in the meta-task.

The function $h(n)$ is defined in terms of two functions, $\underline{h_1(n)}$ and $\underline{h_2(n)}$, which are two different approaches to deriving a lower-bound estimate. Recall that $M = \{m_i : m_i = \min_{0 \leq j < m}(ct(i,j))$, for each $i \in U\}$. For node n let $\underline{mmct(n)}$ be the overall maximum element of M over $\overline{\text{all } i \in U}$ (i.e., "the maximum minimum completion time"). Intuitively, $mmct(n)$ represents the best possible meta-task makespan by making the typically unrealistic assumption that each task in U can be assigned to the machine indicated in M without conflict. Thus, based on [5], $h_1(n)$ is defined as

$$h_1(n) = \max(0,\ (mmct(n) - g(n))). \tag{4}$$

Next, let $\underline{sdma(n)}$ be the sum of the differences between $g(n)$ and each machine availability time over all machines after executing all of the tasks in the partial solution represented by node n:

$$sdma(n) = \sum_{j=0}^{m-1}(g(n) - mat(j)). \tag{5}$$

Intuitively, $sdma(n)$ represents the amount of machine availability time remaining that can be scheduled without increasing the final makespan. Let $\underline{smet(n)}$ be defined as the sum of the minimum expected execution times (i.e., ETC values) for all tasks in U:

$$smet(n) = \sum_{i \in U}(\min_{0 \leq j < m}(ETC(i,j)) \tag{6}$$

This gives an estimate of the amount of remaining work to do, which could increase the final makespan. The function h_2 is then defined as

$$h_2(n) = \max(0,\ (smet(n) - sdma(n))/m), \tag{7}$$

where $(smet(n) - sdma(n))/m$ represents an estimate of the minimum increase in the meta-task makespan if the tasks in U could be "ideally" (but, in general, unrealistically) distributed among the machines. Using these definitions,

$$h(n) = \max(h_1(n), h_2(n)), \tag{8}$$

representing a lower-bound estimate on the time to execute the tasks in U.

Thus, beginning with the root, the node with the minimum $f(n)$ is replaced by its m children, until n_{max} nodes are created. From that point on, any time a node is added, the tree is pruned by deleting the node with the largest $f(n)$. This process continues until a leaf node (representing a complete mapping) is reached. Note that if the tree is not pruned, this method is equivalent to an exhaustive search.

These eleven heuristics were all implemented under the common simulation model described in Section 2. The results from experiments using these implementations are described in the next section. Suggestions for alternative heuristic implementations are given in Section 5.

4. Experimental Results

An interactive software application has been developed that allows simulation, testing, and demonstration of the heuristics examined in Section 3 applied to the meta-tasks defined by the ETC matrices described in Section 2. The software allows a user to specify t and m, to select which ETC matrices to use, and to choose which heuristics to execute. It then generates the specified ETC matrices, executes the desired heuristics, and displays the results, similar to Figures 2 through 13. The results discussed in this section were generated using portions of this software.

When comparing mapping heuristics, the execution time of the heuristics themselves is an important consideration. For the heuristics listed, the execution times varied greatly. The experimental results discussed below were obtained on a Pentium II 400 MHz processor with 1GB of RAM. Each of the simpler heuristics (OLB, UDA, Fast Greedy, and Greedy) executed in a few seconds for one ETC matrix with $t = 512$ and $m = 16$. For the same sized ETC matrix, SA and Tabu, both of which manipulate a single solution during an iteration, averaged less than 30 seconds. GA and GSA required approximately 60 seconds per matrix because they manipulate entire populations, and A* required about 20 minutes per matrix.

The resulting meta-task execution times (makespans) from the simulations for every case of consistency, task heterogeneity, and machine heterogeneity are shown in Figures 2 through 13. All experimental results represent the execution time of a meta-task (defined by a particular ETC matrix) based on the mapping found by the heuristic specified, averaged over 100 different ETC matrices of the same type (i.e., 100 mappings). For each heuristic, the range bars show the minimum and maximum meta-task execution times over the 100 mappings (100 ETC matrices) used to compute the average meta-task execution time. Tables 1 through 4 show sample subsections from the four types of inconsistent ETC matrices considered. Semi-consistent and consistent matrices of the same types could be generated from these matrices as described in Section 2. For the results described here, however, entirely new matrices were generated for each case.

For the four consistent cases, Figures 2 through 5, the UDA algorithm had the worst execution times by an order of magnitude. This is easy to explain. For the consistent cases, all tasks will have the lowest execution time on one machine, and all tasks will be mapped to this particular machine. This corresponds to results found in [1]. Because of this poor performance, the UDA results were not included in Figures 2 through 5. OLB, Max-min, and SA had the next poorest results. GA performed the best for the consistent cases. This was due in part to the good performance of the Min-min heuristic. The best GA solution always came from one of the populations that had been seeded with the Min-min solution. As is apparent in the figures, Min-min performed very well on its own, giving the second best results. The mutation, crossover, and selection operations of the GA were always able to improve on this solution, however. GSA, which also used a Min-min seed, did not always improve upon the Min-min solution. Because of the probabilistic procedure used during selection, GSA would sometimes accept poorer intermediate solutions. These poorer intermediate solutions never led to better final solutions, thus GSA gave the third best results. The performance of A* was hindered because the estimates made by $h_1(n)$ and $h_2(n)$ are not as accurate for consistent cases as they are for inconsistent and semi-consistent cases. For consistent cases, $h_1(n)$ underestimates the competition for machines and $h_2(n)$ underestimates the "workload" distributed to each machine.

These results suggest that if the best overall solution is desired, the GA should be employed. However, the improvement of the GA solution over the Min-min solution was never more than 10%. Therefore, the Min-min hueristic may be more appropriate in certain situations, given the difference in execution times of the two heuristics.

For the four inconsistent test cases in Figures 6 through 9, UDA performs much better and the performance of OLB degrades. Because there is no pattern to the consistency, OLB will assign more tasks to poor or even worst-case machines, resulting in poorer schedules. In contrast, UDA improves because the "best" machines are distributed across the set of machines, thus task assignments will be more evenly distributed among the set of machines avoiding load imbalance. Similarly, Fast Greedy and Min-min performed very well, and slightly outperformed UDA, because the machines providing the best task completion times are more evenly distributed among the set of machines. Min-min was also better than Max-min for all of the inconsistent cases. The advantages Min-min gains by mapping "best case" tasks first outweighs the savings in penalties Max-min has by mapping "worst case" tasks first.

Tabu gave the second poorest results for the inconsistent cases, at least 16% poorer than the other heuristics. Inconsistent matrices generated more successful short hops than the associated consistent matrices. Therefore, fewer long hops were generated and less of the solution space was searched, resulting in poorer solutions. The increased number of successful short hops for inconsistent matrices can be explained as follows. The pairwise comparison procedure used by the short hop procedure will assign machines with better performance first, early in the search procedure. For the consistent cases, these machines will always be from the same set of machines. For inconsistent cases, these machines could be any machine. Thus, for consistent cases, the search becomes somewhat ordered, and the successful short hops get exhausted faster. For inconsistent cases, the lack of order means there are more successful short hops, resulting in fewer long hops.

GA and A* had the best average makespans, and were usually within a small constant factor of each other. The random approach employed by these methods was useful and helped overcome the difficulty of locating good mappings within inconsistent matrices. GA again benefited from having the Min-min initial mapping. A* did well because if the tasks get more evenly distributed among the machines, this more closely matches the lower-bound estimates of $h_1(n)$ and $h_2(n)$.

Finally, consider the semi-consistent cases in Figures 10 through 13. For semi-consistent cases with high machine heterogeneity, the UDA heuristic again gave the worst results. Intuitively, UDA is suffering from the same problem as in the consistent cases: half of all tasks are getting assigned to the same machine. OLB does poorly for high machine heterogeneity cases because worst case matchings will have higher execution times for high machine heterogeneity. For low machine heterogeneity, the worst case matchings have a much lower penalty. The best heuristics for the semi-consistent cases were Min-min and GA. This is not surprising because these were two of the best heuristics from the consistent and inconsistent tests, and semi-consistent matrices are a combination of consistent and inconsistent matrices. Min-min was able to do well because it searched the entire row for each task and assigned a high percentage of tasks to their first choice. GA was robust enough to handle the consistent components of the matrices, and did well for the same reasons mentioned for inconsistent matrices.

5. Alternative Implementations

The experimental results in Section 4 show the performance of each heuristic under the assumptions presented. For several heuristics, specific control parameter values and control functions had to be selected. In most cases, control parameter values and control functions were based on the references cited or experiments conducted. However, for these heuristics, different, valid implementations are possible using different control parameters and control functions.

GA, SA, GSA: Several parameter values could be varied among these techniques, including (where appropriate) population size, crossover probability, mutation probability, stopping criteria, number of runs with different initial populations per result, and the system temperature. The specific procedures used for the following actions could also be modified (where appropriate) including initial population "seed" generation, mutation, crossover, selection, elitism, and the accept/reject new mapping procedure.

Tabu: The short hop method implemented was a "first descent" (take the first improvement possible) method. "Steepest descent" methods (where several short hops are considered simultaneously, and the one with the most improvement is selected) are also used in practice [7]. Other techniques that could be varied are the long hop method, the order of the short hop pair generation-and-exchange sequence, and the stopping condition. Two possible alternative stopping criteria are when the tabu list reaches a specified number of entries, or when there is no change in the best solution in a specified number of hops.

A*: Several variations of the A* method that was employed here could be implemented. Different functions could be used to estimate the lower-bound $h(n)$. The maximum size of the search tree could be varied, and several other techniques exist for tree pruning (e.g., [21]).

In summary, for the GA, SA, GSA, Tabu, and A* heuristics there are a great number of possible valid implementations. An attempt was made to use a reasonable implementation of each heuristic for this study. Future work could examine other implementations.

6. Conclusions

The goal of this study was to provide a basis for comparison and insights into circumstances where one technique will out perform another for eleven different heuristics. The characteristics of the *ETC* matrices used as input for the heuristics and the methods used to generate them were specified. The implementation of a collection of eleven heuristics from the literature was described. The results of the mapping heuristics were discussed, revealing the best heuristics to use in certain scenarios. For the situations, implementations, and parameter values used here, GA was the best heuristic for most cases, followed closely by Min-min, with A* also doing well for inconsistent matrices.

A software tool was developed that allows others to compare these heuristics for many different types of *ETC* matrices. These heuristics could also be the basis of a mapping toolkit. If this toolkit were given an *ETC* matrix representing an actual meta-task and an actual HC environment, the toolkit could analyze the *ETC* matrix, and utilize the best mapping heuristic for that scenario. Depending on the overall situation, the execution time of the mapping heuristic itself may impact this decision. For example, if the best mapping available in less than one minute was desired and if the characteristics of a given *ETC* matrix most closely matched a consistent matrix, Min-min would be used; if more time was available for finding the best mapping, GA and A* should be considered.

The comparisons of the eleven heuristics and twelve situations provided in this study can be used by researchers as a starting point when choosing heuristics to apply in different scenarios. They can also be used by researchers for selecting heuristics to compare new, developing techniques against.

Acknowledgments – The authors thank Shoukat Ali and Taylor Kidd for their comments.

References

[1] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," *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*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Sept. 1997 (D. Hensgen, advisor.)

[3] T. D. Braun, H. J. Siegel, N. Beck, L. 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," *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1998, pp. 330–335

(included in the Proceedings of the *7th IEEE Symposium on Reliable Distributed Systems*, 1998).

[4] H. Chen, N. S. Flann, and D. W. Watson, "Parallel genetic simulated annealing: A massively parallel SIMD approach," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 9, No. 2, Feb. 1998, pp. 126–136.

[5] K. Chow and B. Liu, "On mapping signal processing algorithms to a heterogeneous multiprocessor system," *1991 International Conference on Acoustics, Speech, and Signal Processing - ICASSP 91*, Vol. 3, May 1991, pp. 1585–1588.

[6] M. Coli and P. Palazzari, "Real time pipelined system design through simulated annealing," *Journal of Systems Architecture*, Vol. 42, No. 6-7, Dec. 1996, pp. 465–475.

[7] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, "Improving search by incorporating evolution principles in parallel tabu search," *1994 IEEE Conference on Evolutionary Computation*, Vol. 2, 1994, pp. 823–828.

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

[9] 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.

[10] R. F. Freund and H. J. Siegel, "Heterogeneous processing," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 13–17.

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

[12] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, MA, June 1997.

[13] D. A. Hensgen, T. Kidd, M. C. Schnaidt, D. St. John, H. J. Siegel, T. D. Braun, M. Maheswaran, 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 Workshop on Heterogeneous Computing Systems (HCW '99)*, Apr. 1999, to appear.

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

[15] 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.

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

[17] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671–680.

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

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

[20] G. Rudolph, "Convergence analysis of canonical genetic algorithms," *IEEE Transactions on Neural Networks*, Vol. 5, No. 1, Jan. 1994, pp. 96–101.

[21] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[22] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion," *IEEE Transactions on Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 197–203.

[23] P. Shroff, D. Watson, N. Flann, and R. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, April 1996, pp. 98–104.

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

[25] 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.

[26] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *IEEE Computer*, Vol. 27, No. 6, June 1994, pp. 17–26.

[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," *Journal of Parallel and Distributed Computing*, Vol. 47, No. 1, Nov. 1997, pp. 1–15.

Biographies

Tracy D. Braun is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received a Bachelor of Science in Electrical Engineering with Honors and High Distinction from the University of Iowa in 1995. In 1997, he received an MSEE from the School of Electrical and Computer Engineering at Purdue University. He received a Benjamin Meisner Fellowship from Purdue University for the 1995-1996 academic year. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. He is an active member of the Beta Chapter of Eta Kappa Nu at Purdue University, and has held several offices during his studies at Purdue, including chapter President. He has also been employed at Norand Data Systems and Silicon Graphics Inc./Cray Research. His research interests include parallel algorithms, heterogeneous computing, computer security, and software design.

H. J. 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.

Noah Beck is a Research Assistant and MSEE student at Purdue University in the School of Electrical and Computer Engineering. He received a Bachelor of Science in Computer Engineering from Purdue University in 1997, and is an active member of the Beta chapter of the Eta Kappa Nu honorary society. He has also been employed at Intel Corporation, and his research interests include microprocessor architecture, parallel computing, and heterogeneous computing.

Ladislau L. Bölöni is a PhD student and Research Assistant in the Computer Sciences Department at Purdue University. He received a Diploma Engineer degree in Computer Engineering with Honors from the Technical University of Cluj-Napoca, Romania in 1993. He received a fellowship from the Hungarian Academy of Sciences for the 1994-95 academic year. He is a member of ACM and the Upsilon Pi Epsilon honorary society. His research interests include distributed object systems, autonomous agents and parallel computing.

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.

Albert I. Reuther is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received his Bachelor of Science in Computer and Electrical Engineering with Highest Distinction in 1994 and received a Masters of Science in Electrical Engineering in 1996, both at Purdue. He was a Purdue Andrews Fellowship recipient in the 1994-95 and 1995-96 academic years. He is a member of IEEE, IEEE Computer Society, ACM, and Eta Kappa Nu honorary society and has been employed by General Motors and Hewlett-Packard. His research interests include multimedia systems, heterogeneous computing, parallel processing, and educational multimedia.

James P. Robertson currently works for Motorola's PowerPC System Performance and Modeling group. He received a Bachelor of Science in Computer Engineering with Honors from the school of Electrical and Computer Engineering at Purdue University in 1996. As an undergraduate student he received an NSF undergraduate research scholarship. In 1998 he received an MSEE from Purdue University. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. While attending Purdue

University he was an active member of the Beta Chapter of Eta Kappa Nu, having held several offices including chapter Treasurer.

Mitchell D. Theys is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received a Bachelor of Science in Computer and Electrical Engineering in 1993 with Highest Distinction, and a Master of Science in Electrical Engineering in 1996, both from Purdue. He received support from a Benjamin Meisner Fellowship from Purdue University, an Intel Graduate Fellowship, and an AFCEA Graduate Fellowship. He is a member of the Eta Kappa Nu honorary society, IEEE, and IEEE Computer Society. He was elected President of the Beta Chapter of Eta Kappa Nu at Purdue University and has held several various offices during his stay at Purdue. He has held positions with Compaq Computer Corporation, S&C Electic Company, and Lawrence Livermore National Laboratory. His research interests include design of single chip parallel machines, heterogeneous computing, parallel processing, and software/hardware design.

Bin Yao is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received Bachelor of Science in Electrical Engineering from Beijing University in 1996. He received Andrews Fellowship from Purdue University for the academic years 1996-1998. He is a student member of the IEEE. His research interests include distributed algorithms, fault tolerant computing, and heterogeneous computing.

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. Dr. 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.

```
initial population generation;
evaluation;
while (stopping criteria not met) {
          selection;
          crossover;
          mutation;
          evaluation;
}
```

Figure 1. General procedure for a Genetic Algorithm, based on [26].

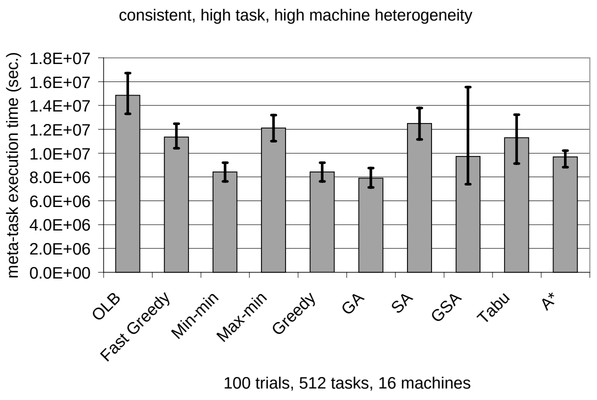

Figure 2. Consistent, high task, high machine heterogeneity.

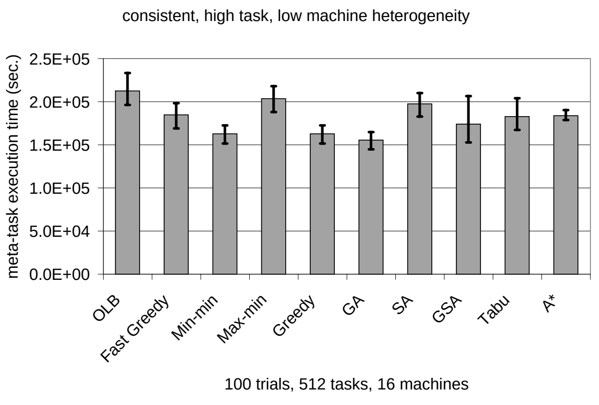

Figure 3. Consistent, high task, low machine heterogeneity.

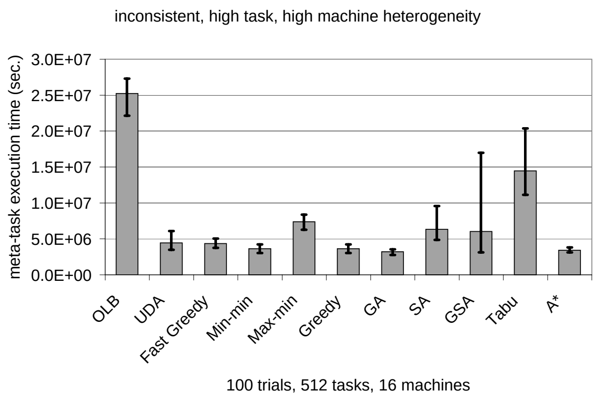

Figure 6. Inconsistent, high task, high machine heterogeneity.

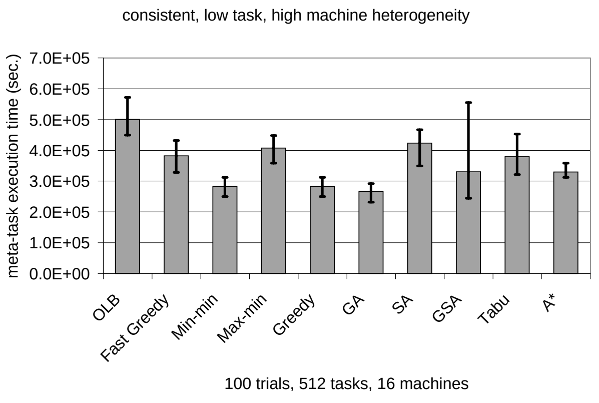

Figure 4. Consistent, low task, high machine heterogeneity.

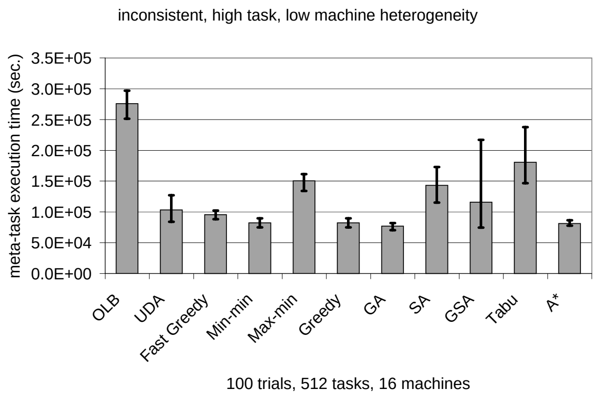

Figure 7. Inconsistent, high task, low machine heterogeneity.

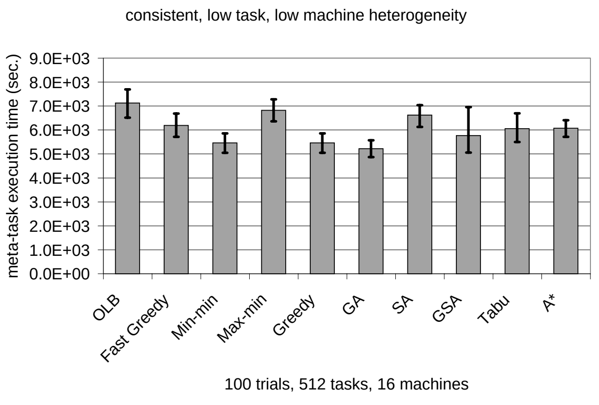

Figure 5. Consistent, low task, low machine heterogeneity.

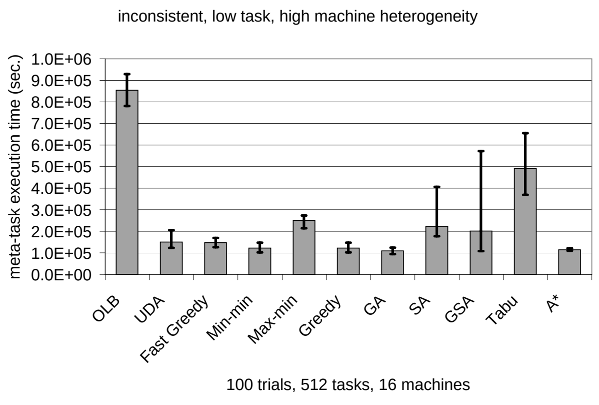

Figure 8. Inconsistent, low task, high machine heterogeneity.

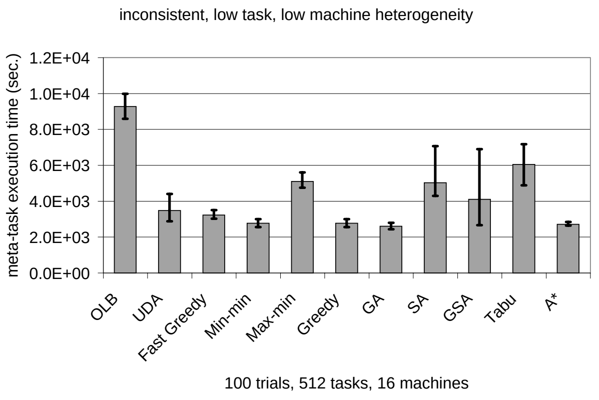

Figure 9. Inconsistent, low task, low machine heterogeneity.

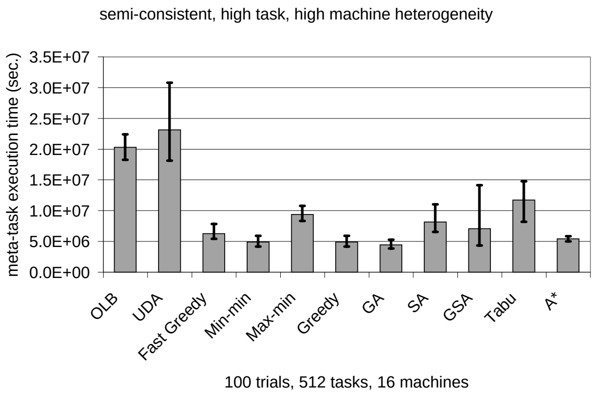

Figure 10. Semi-consistent, high task, high machine heterogeneity.

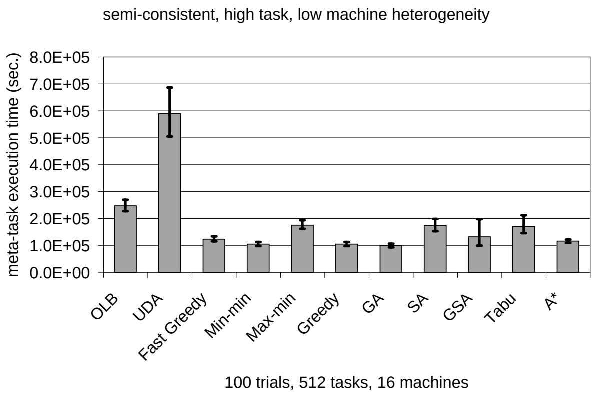

Figure 11. Semi-consistent, high task, low machine heterogeneity.

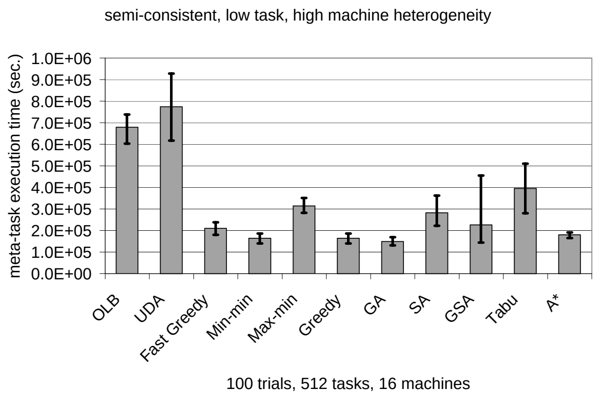

Figure 12. Semi-consistent, low task, high machine heterogeneity.

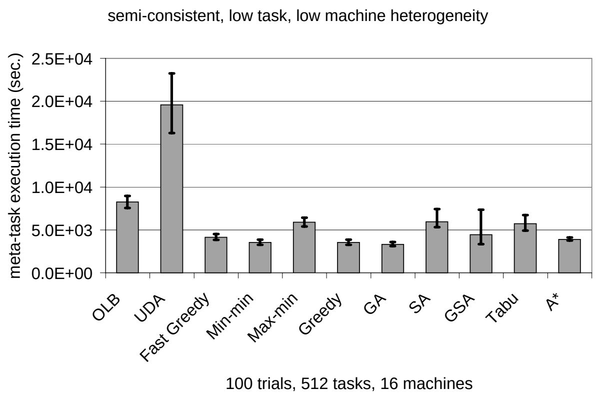

Figure 13. Semi-consistent, low task, low machine heterogeneity.

	machines							
t	436,735.9	815,309.1	891,469.0	1,722,197.6	1,340,988.1	740,028.0	1,749,673.7	251,140.1
a	950,470.7	933,830.1	2,156,144.2	2,202,018.0	2,286,210.0	2,779,669.0	220,536.3	1,769,184.5
s	453,126.6	479,091.9	150,324.5	386,338.1	401,682.9	218,826.0	242,699.6	11,392.2
k	1,289,078.2	1,400,308.1	2,378,363.0	2,458,087.0	351,387.4	925,070.1	2,097,914.2	1,206,158.2
s	646,129.6	576,144.9	1,475,908.2	424,448.8	576,238.7	223,453.8	256,804.5	88,737.9
	1,061,682.3	43,439.8	1,355,855.5	1,736,937.1	1,624,942.6	2,070,705.1	1,977,650.2	1,066,470.8
	10,783.8	7,453.0	3,454.4	23,720.8	29,817.3	1,143.7	44,249.2	5,039.5
	1,940,704.5	1,682,338.5	1,978,545.6	788,342.1	1,192,052.5	1,022,914.1	701,336.3	1,052,728.3

Table 1. Sample 8×8 excerpt from ETC with inconsistent, high task, high machine heterogeneity.

	machines							
t	21,612.6	13,909.7	6,904.1	3,621.5	3,289.5	8,752.0	5,053.7	14,515.3
a	578.4	681.1	647.9	477.1	811.9	619.5	490.9	828.7
s	122.8	236.9	61.3	143.6	56.0	313.4	283.5	241.9
k	1,785.7	1,528.1	6,998.8	4,265.3	3,174.6	3,438.0	7,168.4	2,059.3
s	510.8	472.0	358.5	461.4	1,898.7	1,535.4	1,810.2	906.6
	22,916.7	18,510.0	11,932.7	6,088.3	9,239.7	15,036.4	18,107.7	12,262.6
	5,985.3	2,006.5	1,546.4	6,444.6	2,640.0	7,389.3	5,924.9	1,867.2
	16,192.4	3,088.9	16,532.5	13,160.6	10,574.2	7,136.3	15,353.4	2,150.6

Table 2. Sample 8×8 excerpt from ETC with inconsistent, high task, low machine heterogeneity.

	machines							
t	16,603.2	71,369.1	39,849.0	44,566.1	55,124.3	9,077.3	87,594.5	31,530.5
a	738.3	2,375.0	5,606.2	804.9	1,535.8	4,772.3	994.2	1,833.9
s	1,513.8	45.1	1,027.3	2,962.1	2,748.2	2,406.3	19.4	969.9
k	2,219.9	5,989.2	2,747.0	88.2	2,055.1	665.0	356.3	2,404.9
s	12,654.7	10,483.7	10,601.5	6,804.6	134.3	10,532.8	12,341.5	5,046.3
	4,226.0	48,152.2	11,279.3	35,471.1	30,723.4	24,234.0	6,366.9	22,926.9
	20,668.5	28,875.9	29,610.1	7,363.3	24,488.0	31,077.3	8,705.0	11,849.4
	52,953.2	14,608.1	58,137.2	16,685.5	36,571.3	35,888.8	38,147.0	15,167.5

Table 3. Sample 8×8 excerpt from ETC with inconsistent, low task, high machine heterogeneity.

	machines							
t	512.9	268.0	924.9	494.4	611.2	606.9	921.6	209.6
a	8.5	16.8	23.4	19.2	27.9	22.7	19.6	8.3
s	228.8	238.5	107.2	180.0	334.6	88.2	192.8	125.7
k	345.1	642.4	136.8	206.2	559.5	349.5	640.2	664.2
s	117.3	235.9	149.9	71.5	136.6	363.6	182.8	359.5
	240.7	412.0	259.1	319.8	237.5	338.3	178.5	537.7
	462.8	93.3	574.9	449.4	421.8	559.6	487.7	298.7
	119.5	36.7	224.2	194.2	176.5	156.8	182.7	192.0

Table 4. Sample 8×8 excerpt from ETC with inconsistent, low task, low machine heterogeneity.