Journal of Parallel and Distributed Computing **61**, 810–837 (2001)
doi:10.1006/jpdc.2000.1714, available online at http://www.idealibrary.com on IDEAL®

A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems[1]

Tracy D. Braun, Howard Jay Siegel,[2] and Noah Beck

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

Ladislau L. Bölöni

CPlane Inc., 897 Kifer Road, Sunnyvale, California 94086
E-mail: boloni@cplane.com

Muthucumaru Maheswaran

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

Albert I. Reuther

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

James P. Robertson

Motorola, 6300 Bridgepoint Parkway, Bldg. #3, MD:OE71, Austin, Texas 78730
E-mail: robertso@ibmoto.com

Mitchell D. Theys

Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois 60607-7053
E-mail: mtheys@uic.edu

Bin Yao

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

[1] This research was supported in part by the DARPA/ITO Quorum Program under NPS Subcontracts N62271-98-M-0217 and N62271-98-M-0448, and under the GSA Subcontract GS09K99BH0250. Some of the equipment used was donated by Intel and Microsoft.

[2] Address as of August 2001: Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, Colorado 80523. E-mail: hj@colostate.edu.

Debra Hensgen

OpenTV, 401 East Middlefield Road, Mountain View, California 94043
E-mail: dhensgen@opentv.com

and

Richard F. Freund

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

Received October 15, 1998; revised June 18, 2000; accepted November 15, 2000

Mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different high-performance machines, interconnected with high-speed links, to perform different computationally intensive applications that have diverse computational requirements. HC environments are well suited to meet the computational demands of large, diverse groups of tasks. The problem of optimally mapping (defined as matching and scheduling) these tasks onto the machines of a distributed 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 study of each heuristic. Therefore, a collection of 11 heuristics from the literature has been selected, adapted, implemented, and analyzed under one set of common assumptions. It is assumed that the heuristics derive a mapping statically (i.e., off-line). It is also assumed that a metatask (i.e., a set of independent, noncommunicating tasks) is being mapped and that the goal is to minimize the total execution time of the metatask. The 11 heuristics examined are Opportunistic Load Balancing, Minimum Execution Time, Minimum Completion Time, Min–min, Max–min, Duplex, 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 comparison results are discussed. It is shown that for the cases studied here, the relatively simple Min–min heuristic performs well in comparison to the other techniques. © 2001 Academic Press

Key Words: A*; Genetic Algorithm; heterogeneous computing; mapping heuristics; metatasks; simulated annealing; static matching; Tabu search.

1. INTRODUCTION

Mixed-machine *heterogeneous computing* (*HC*) environments utilize a distributed suite of different high-performance machines, interconnected with high-speed links, to perform different computationally intensive applications that have diverse computational requirements [1, 14, 16, 18, 30, 37]. The matching of tasks to machines and scheduling the execution order of these tasks is referred to as *mapping*. The general problem of optimally mapping tasks to machines in an HC suite has been shown to be NP-complete [15, 23]. Heuristics developed to perform this mapping function are often difficult to compare because of different underlying assumptions in the original study of each heuristic [6]. Therefore, a collection of 11 heuristics from the literature has been selected, adapted, implemented, and compared by simulation studies under one set of common assumptions.

To facilitate these comparisons, certain simplifying assumptions were made. For these studies, let a *metatask* be defined as a collection of independent tasks with no intertask data dependencies. The mapping of the metatasks is being performed *statically* (i.e., off-line, or in a predictive manner). The goal of this mapping is to minimize the total execution time of the metatask.

Metatasks composed of independent tasks occur in many situations. For example, all of the jobs submitted to a supercomputer center by different users would constitute a metatask. Another example of a metatask would be a group of image processing applications all operating on different images.

Static mapping is utilized in many different types of analyses and environments. The most common use of static mapping is for predictive analyses (e.g., to plan the work for the next day and/or to meet a deadline). For example, assume a NASA center knows it will have a 2-hour communication window with a probe tomorrow. In those 2 hours, NASA center will have to analyze the data the probe sends back and determine if the probe needs to be adjusted before communications blackout. Therefore, the NASA center will want to plan the most efficient way to handle the data *a priori* and determine if the deadline can be met. Another use of static mapping is for "what if" simulation studies. For example, a system administrator may need to justify the benefits of purchasing another machine for an HC suite. Static mapping is also used for post-mortem analyses. For example, a static mapper can be used *ex post facto* to evaluate how well an on-line (i.e., dynamic) mapper is performing. Future high-powered computational grids [16] will also be able to utilize static mapping techniques to distribute resources and computational power. The wide applicability of static mapping makes it an important area for ongoing research.

It is also assumed that each machine executes a single task at a time (i.e., no multitasking), in the order in which the tasks are assigned. The size of the metatask (i.e., the number of tasks to execute), τ, and the number of machines in the HC suite, μ, are static and known beforehand.

This study provides one even basis for comparison and insights into circumstances where one mapping technique will out-perform another. The evaluation procedure is specified, the heuristics are defined, and then comparison results are shown. For the cases studied here, the relatively simple Min–min heuristic (defined in Section 3) performs well in comparison to other, more complex techniques investigated.

The remainder of this paper is organized as follows. Section 2 defines the computational environment parameters that were varied in the simulations. Descriptions of the 11 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) [21]. MSHN was a collaborative research effort among the Naval Postgraduate School, NOEMIX, Purdue University, and the University of Southern California. One objective of MSHN was to design and evaluate mapping heuristics for different types of HC environments.

2. SIMULATION MODEL

The 11 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 a $\tau \times \mu$ *ETC* (expected time to compute) *matrix*. The assumption that these estimated expected execution times are known is commonly made when studying mapping heuristics for HC systems (e.g., [19, 26, 40]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [27, 30, 37].)

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 metatask. Thus, for an arbitrary task t_i and an arbitrary machine m_j, $ETC(t_i, m_j)$ is the estimated execution time of t_i on m_j.

The $ETC(t_i, m_j)$ entry could be assumed to include the time to move the executables and data associated with task t_i from their known source to machine m_j. For cases when it is impossible to execute task t_i on machine m_j (e.g., if specialized hardware is needed), the value of $ETC(t_i, m_j)$ is set to infinity.

For the simulation studies, characteristics of the ETC matrices were varied in an attempt to represent a range of possible HC environments. The ETC matrices used were generated using the following method. Initially, a $\tau \times 1$ *baseline* column vector, B, of floating point values is created. Let ϕ_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, $x_b^i \in [1, \phi_b)$, and letting $B(i) = x_b^i$ for $0 \leqslant i < \tau$. Next, the rows of the ETC matrix are constructed. Each element $ETC(t_i, m_j)$ in row i of the ETC matrix is created by taking the baseline value, $B(i)$, and multiplying it by a uniform random number, $x_r^{i,j}$, which has an upper bound of ϕ_r. This new random number, $x_r^{i,j} \in [1, \phi_r)$, is called a *row multiplier*. One row requires μ different row multipliers, $0 \leqslant j < \mu$. Each row i of the ETC matrix can then be described as $ETC(t_i, m_j) = B(i) \times x_r^{i,j}$, for $0 \leqslant j < \mu$. (The baseline column itself does not appear in the final ETC matrix.) This process is repeated for each row until the $\tau \times \mu$ ETC matrix is full. Therefore, any given value in the ETC matrix is within the range $[1, \phi_b \times \phi_r)$ [29].

To evaluate the heuristics for different mapping scenarios, the characteristics of the ETC matrix were varied based on several different methods from [4]. The amount of variance among the execution times of tasks in the metatask 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

TABLE 1

Sample 8 × 8 Excerpt from One of the 512 × 16 ETC Matrices with Consistent, High Task, High Machine Heterogeneity Used in Generating Fig. 3

	machines							
t	25,137.5	52,468.0	150,206.8	289,992.5	392,348.2	399,562.1	441,485.5	518,283.1
a	30,802.6	42,744.5	49,578.3	50,575.6	58,268.1	58,987.9	85,213.2	87,893.0
s	242,727.1	661,498.5	796,048.1	817,745.8	915,235.9	925,875.6	978,057.6	1,017,448.1
k	68,050.1	303,515.9	324,093.1	643,133.7	841,877.3	856,312.9	861,314.8	978,066.3
s	6,480.2	42,396.7	98,105.4	166,346.8	240,319.5	782,658.5	871,532.6	1,203,339.8
	175,953.8	210,341.9	261,825.0	306,034.2	393,292.2	412,085.4	483,691.9	515,645.9
	116,821.4	240,577.6	241,127.9	406,791.4	1,108,758.0	1,246,430.8	1,393,067.0	1,587,743.1
	36,760.6	111,631.5	150,926.0	221,390.0	259,491.1	383,709.7	442,605.7	520,276.8

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. The ranges were chosen to reflect the fact that in real situations there is more variability across execution times for different tasks on a given machine than the execution time for a single task across different machines.

To further vary the characteristics of the ETC matrices in an attempt to capture more aspects of realistic mapping situations, different ETC matrix consistencies were used. An ETC matrix is said to be *consistent* if whenever a machine m_j executes any task t_i faster than machine m_k, then machine m_j executes all tasks faster than machine m_k [4]. Consistent matrices were generated by sorting each row of the ETC matrix independently, with machine m_0 always being the fastest and machine $m_{(\mu-1)}$ the slowest. In contrast, *inconsistent* matrices characterize the situation where machine m_j may be faster than machine m_k for some tasks and slower for others. These matrices are left in the unordered, random state in which they were generated (i.e., no consistency is enforced). *Partially-consistent* matrices are inconsistent matrices that include a consistent submatrix of a predefined size. For the partially-consistent matrices used here, the row elements in column positions $\{0, 2, 4, ...\}$ of row i are extracted, sorted, and replaced in order, while the row

TABLE 2

Sample 8 × 8 Excerpt from One of the 512 × 16 ETC Matrices with Inconsistent, High Task, High Machine Heterogeneity Used in Generating Fig. 5

	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 3

Sample 8 × 8 Excerpt from One of the 512 × 16 ETC Matrices with Partially-Consistent, High Task, High Machine Heterogeneity Used in Generating Fig. 7

	machines							
t	1,003,569.7	910,811.9	1,085,529.8	1,646,242.8	1,087,655.5	2,121,084.5	1,141,898.7	749,952.3
a	27,826.6	409,936.4	168,341.7	858,511.3	353,691.8	270,449.8	420,799.6	152,786.0
s	8,415.4	101,202.5	16,453.7	64,152.5	29,172.8	36,738.5	61,114.5	142,411.2
k	17,050.5	195,067.8	79,175.8	787,263.3	173,239.2	438,599.0	378,563.4	747,305.4
s	32,275.4	434,445.7	135,989.1	496,326.8	221,097.9	463,577.7	244,747.3	431,704.5
	28,850.0	138,449.0	32,730.9	93,025.9	90,044.4	223,827.9	96,715.5	129,979.1
	145,038.5	350,917.4	210,957.4	265,590.5	486,217.7	317,915.2	728,732.4	625,365.5
	11,763.0	460,975.2	214,456.3	821,904.1	296,960.4	459,109.0	350,026.7	54,926.4

elements in column positions $\{1, 3, 5, \ldots\}$ remain unordered (i.e., the even columns are consistent and the odd columns are, in general, inconsistent).

Twelve combinations of ETC matrix characteristics were used in this study: high or low task heterogeneity, high or low machine heterogeneity, and one type of consistency (consistent, inconsistent, or partially consistent). Three sample ETC matrices from these 12 possible permutations of the characteristics are shown in Tables 1 through 3. (Examples of all 12 ETC matrices can be found in [5].) The results in this study (see Section 4) used ETC matrices that had $\tau = 512$ tasks and $\mu = 16$ machines. These results were taken as the average of 100 ETC matrices for each case.

While it was necessary to select some specific parameter values for τ, μ, and the ETC entries to allow implementation of a simulation, the 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 values and the evaluation software of this study will still apply. For example, an alternative method for generating ETC matrices that could be used with this evaluation software is described in [2].

3. HEURISTIC DESCRIPTIONS

3.1. Introduction

The definitions of the 11 static metatask mapping heuristics are provided below. First, some preliminary terms must be defined. *Machine availability time*, $mat(m_j)$, is the earliest time machine m_j can complete the execution of all the tasks that have previously been assigned to it (based on the ETC entries for those tasks). The *completion time* for a new task t_i on machine m_j, $ct(t_i, m_j)$, is the machine availability time for m_j plus the execution time of task t_i on machine m_j, i.e., $ct(t_i, m_j) = mat(m_j) + ETC(t_i, m_j)$. The performance criterion used to compare the results of the heuristics is the maximum value of $ct(t_i, m_j)$, for $0 \leqslant i < \tau$ and $0 \leqslant j < \mu$. The maximum $ct(t_i, m_j)$ value, over $0 \leqslant i < \tau$ and $0 \leqslant j < \mu$, is the *metatask execution time*, and is called the *makespan* [32]. Each heuristic is attempting to minimize the makespan, i.e., finish execution of the metatask as soon as possible.

The descriptions below implicitly assume that the machine availability times are updated after each task is mapped. For heuristics where the tasks are considered in an arbitrary order, the order in which the tasks appeared in the ETC matrix was used. Most of the heuristics discussed here had to be adapted for this problem domain.

For many of the heuristics, there are control parameters 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. Some of these parameters and functions are mentioned in Section 5.

3.2. Heuristics

OLB: *Opportunistic Load Balancing* (*OLB*) assigns each task, in arbitrary order, to the next machine that is expected to be available, regardless of the task's expected execution time on that machine [3, 17, 18]. The intuition behind OLB is to keep all machines as busy as possible. One advantage of OLB is its simplicity, but because OLB does not consider expected task execution times, the mappings it finds can result in very poor makespans.

MET: In contrast to OLB, *Minimum Execution Time* (*MET*) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability [3, 17]. The motivation behind MET is to give each task to its best machine. This can cause a severe load imbalance across machines. In general, this heuristic is obviously not applicable to HC environments characterized by consistent ETC matrices.

MCT: *Minimum Completion Time* (*MCT*) assigns each task, in arbitrary order, to the machine with the minimum expected completion time for that task [3]. This causes some tasks to be assigned to machines that do not have the minimum execution time for them. The intuition behind MCT is to combine the benefits of OLB and MET, while avoiding the circumstances in which OLB and MET perform poorly.

Min–min: The *Min–min* heuristic begins with the set U of all unmapped tasks. Then, the set of minimum completion times, $M = \{\min_{0 \leqslant j < \mu}(ct(t_i, m_j))$, for each $t_i \in U\}$, 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). Last, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., U is empty) [3, 17, 23]. Min–min is based on the minimum completion time, as is MCT. However, Min–min considers all unmapped tasks during each mapping decision and MCT only considers one task at a time.

Min–min maps the tasks in the order that changes the machine availability status by the least amount that any assignment could. Let t_i be the first task mapped by Min–min onto an empty system. The machine that finishes t_i the earliest, say m_j, is also the machine that executes t_i the fastest. For every task that Min–min maps after t_i, the Min–min heuristic changes the availability status of m_j by the least possible amount for every assignment. Therefore, the percentage of tasks assigned to their first choice (on the basis of execution time) is likely to be higher for Min–min than for Max–min (defined next). The expectation is that a smaller makespan can be obtained if more tasks are assigned to the machines that complete them the earliest and also execute them the fastest.

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, 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). Last, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., U is empty) [3, 17, 23].

Intuitively, Max–min attempts to minimize the penalties incurred from performing tasks with longer execution times. Assume, for example, that the metatask being mapped has many tasks with very short execution times and one task with a very long execution time. Mapping the task with the longer execution time to its best machine first allows this task to be executed concurrently with the remaining tasks (with shorter execution times). For this case, this would be a better mapping than a Min–min mapping, where all of the shorter tasks would execute first, and then the longer running task would execute while several machines sit idle. Thus, in cases similar to this example, the Max–min heuristic may give a mapping with a more balanced load across machines and a better makespan.

Duplex: The *Duplex* heuristic is literally a combination of the Min–min and Max–min heuristics. The Duplex heuristic performs both of the Min–min and Max–min heuristics and then uses the better solution [3, 17]. Duplex can be performed to exploit the conditions in which either Min–min or Max–min performs well, with negligible overhead.

GA: *Genetic Algorithms* (*GAs*) are a technique used for searching large solution spaces (e.g., [22, 31, 40, 42, 44]). The version of the heuristic used for this study was adapted from [44] for this particular problem domain. Figure 1 shows the steps in a general GA.

The GA implemented here operates on a *population* of 200 *chromosomes* (possible mappings) for a given metatask. Each chromosome is a $\tau \times 1$ vector, where position i ($0 \leqslant i < \tau$) represents task t_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* (i.e., mapping for the metatask) and 199 random solutions. The latter method is called *seeding* the population with a

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

FIG. 1. General procedure for a Genetic Algorithm, based on [41].

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.

Each chromosome has a *fitness value*, which is the makespan that results from the matching of tasks to machines within that chromosome. After the generation of the initial population, all of the chromosomes in the population are evaluated based on their fitness value, with a smaller fitness value being a better mapping. Then, the main loop in Fig. 1 is entered and a rank-based roulette wheel scheme [41] is used for *selection.* This scheme probabilistically duplicates some chromosomes and deletes others, where better mappings have a higher probability of being duplicated in the next generation. *Elitism*, the property of guaranteeing the best solution remains in the population [33], was also implemented. The population size stays fixed at 200.

Next, the *crossover* operation selects a random 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 chromosome, then randomly selects a task within the chromosome, and randomly reassigns it to a new machine. Every chromosome is considered for mutation with a probability of 40%. For both crossover and mutation, the random operations select values from a uniform distribution.

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 to the same mapping. Until the stopping criterium is met, the loop repeats, beginning with the selection step. The stopping criterium 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 metatask at a time. This solution uses the same representation as the chromosome for the GA. The initial implementation of SA was evaluated and then modified and refined to give a better final version. Both the initial and final implementations are described below.

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., [10, 28, 31, 34, 45]). This probability is based on a *system temperature* that decreases for each iteration. As the system temperature "cools," it is more difficult for poorer solutions to be accepted. The initial system temperature is the makespan of the initial (random) mapping.

The initial SA procedure implemented here is as follows. The first 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 [10]. 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

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

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

For solutions with similar makespans (or if the system temperature is very large), $y \to 0.5$, and poorer solutions are accepted with approximately a 50% probability. 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 reduced to 90% of its current value. (This percentage is defined as the *cooling rate*.) This completes one iteration of SA. The heuristic stops when there is no change in the makespan for 150 iterations or the system temperature approaches zero. Most tests ended when the system temperature approached zero (approximated by 10^{-200}).

Results from preliminary studies using the initial implementation described above showed that the GA usually found the best mappings of all 11 heuristics. However, the execution time of the SA heuristic was much shorter than that of the GA. Therefore, to provide a fairer comparison, the following changes were made to the final SA implementation. First, the stopping conditions were modified so that the number of unchanged iterations was raised to 200 and two different cooling rates were used, 80 and 90%. Next, SA was allowed to execute eight times for each cooling rate, using the best solution from all 16 runs as the final mapping. Last, four of the eight runs for each cooling rate were seeded with the Min–min solution, just as with the GA. The modifications gave SA an execution time as long as GA.

Even with the additional execution time and Min–min seedings, SA still found poorer solutions than Min–min or GA. Because SA allows poorer solutions to be accepted at intermediate stages, it allows some very poor solutions in the initial stages, from which it can never recover (see Section 4).

GSA: The *Genetic Simulated Annealing* (*GSA*) heuristic is a combination of the GA and SA techniques [7, 36]. In general, GSA follows procedures similar to the GA outlined above. 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 chromosome.

Specifically, the initial system temperature was set to the average makespan of the initial population and reduced to 90% of its current value for each iteration. Whenever a mutation or crossover occurs, the new chromosome is compared with the corresponding original chromosome. If the new makespan is less than the original makespan plus the system temperature, then the new chromosome is accepted [7, 36]. Otherwise, the original chromosome survives to the next iteration. Therefore, as the system temperature decreases, it is again more difficult for poorer solutions 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. 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 [12, 20, 31]. 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 as the initial solution, generated from a uniform distribution. To manipulate the current solution and move through the solution space, a *short hop* is performed. The intuitive purpose of a short hop is to find the nearest local minimum solution within the solution space. The basic procedure for performing a short hop is to consider, for each possible pair of tasks, each possible pair of machine assignments, while the other $\tau-2$ assignments are unchanged. This is done for every possible pair of tasks. The pseudocode for the short hop procedure is given in Fig. 2.

Let the tasks in the pair under consideration be denoted ti and tj in Fig. 2. (The machine assignments for the other $\tau-2$ tasks are held fixed.) The machines to which tasks ti and tj are remapped are mi and mj, respectively. For each possible pair of tasks, each possible pair of machine assignments is considered. Lines 1 through 4 set the boundary values of the different loops. Line 6 or 8 is where each new solution (mapping) is evaluated, and line 9 is where the new solution is considered for acceptance. Each of these new solutions is a short hop. If the new makespan is an improvement, the new solution is saved, replacing the current solution. (This is defined as a *successful short hop*.) When ti and tj represent the same task (ti = tj), a special case occurs (line 5). In these situations, all machines for that one task are considered.

```
0   LOOP:  /* begin short hop procedure */
1          for ti = 0 to $\tau$ -1          /* first task in pair */
2              for mi = 0 to $\mu$-1          /* first machine in pair */
3                  for tj = ti to $\tau$-1          /* second task in pair */
4                      for mj = 0 to $\mu$-1          /* second machine in pair */
5                          if (ti == tj)
6                              evaluate new solution
                               with task tj on machine mj;
7                          else
8                              evaluate new solution with
                               task ti on machine mi and
                               task tj on machine mj;
9                          if (new solution is better) {
10                             replace old solution with new solution;
11                             successful_hops = successful_hops + 1;
12                             goto LOOP;          /* restart from inital state */
                           }
13                         if (successful_hops == $limit_{hops}$)
14                             goto END;          /* end all searching */
15                     end for
16                 end for
17             end for
18         end for
19  END:
```

FIG. 2. Pseudocode describing the short hop procedure used in Tabu search.

When any new solution is found to be an improvement (lines 9 to 12), the procedure breaks out of the for loops, and starts searching from the beginning again. The short hop procedure ends when (1) every pair-wise remapping combination has been exhausted with no improvement (i.e., the bounds of all four for loops in Fig. 2 have been reached), or (2) the limit on the total number of successful hops ($limit_{hops}$) is reached.

When the short hop procedure ends, 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. After each successful long hop, the short hop procedure is repeated.

The stopping criterion for the entire heuristic is when the sum of the total number of successful short hops and successful long hops equals $limit_{hops}$. Then, the best mapping from the tabu list is the final answer.

Similar to SA, some parameters of Tabu were varied in an attempt to make Tabu more competitive with GA, while also trying to provide a fairer comparison between Tabu and GA. To this end, the value used for $limit_{hops}$ was varied depending on the type of consistency of the matrix being considered, as explained below.

Because of the implementation of the short hop procedure described above, the execution time of the Tabu search depended greatly on the type of consistency of the ETC matrix. Each time a new task is considered for remapping in the short hop procedure, it is first considered on machine m_0, then m_1, etc. For consistent matrices, these will be the fastest machines. Therefore, once a task gets reassigned to a fast machine, the remaining permutations of the short hop procedure will be unsuccessful. That is, because the short hop procedure begins searching sequentially from the best machines, there will be a larger number of unsuccessful hops performed for each successful hop for consistent ETC matrices. Thus, the execution time of Tabu will increase.

Therefore, to keep execution times fair and competitive with GA, $limit_{hops}$ was set to 1000 for consistent ETC matrices, 2000 for partially-consistent matrices, and 2500 for inconsistent matrices. When most test cases had stopped, the percentage of successful short hops was high (90% or more) relative to the percentage of successful long hops (10% or less). But because there were always long hops being performed, every pair-wise combination of short hops was being exhausted, and new, different regions of the solution space were being searched.

A*: The final heuristic in the comparison study is the A* heuristic. A* has been applied to many other task allocation problems (e.g., [9, 26, 31, 34, 35]). The technique used here is similar to [9].

*A** is a search technique based on a μ-ary tree, beginning at a root node that is a null solution. As the tree grows, nodes represent partial mappings (a subset of tasks is assigned to machines). The partial mapping (solution) of a child node has one more task mapped than the parent node. Call this additional task t_a. Each parent node generates μ children, one for each possible mapping of t_a. After a parent node has done this, the parent node becomes inactive. To keep execution time of the heuristic tractable, there is a pruning process to limit the maximum number of active nodes in the tree at any one time (in this study, to 1024).

Each node, n, has a *cost function*, $f(n)$, 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 $g(n)$ 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 ($\max_{0 \leqslant j < \mu} mat(m_j)$) based on the set of tasks that have been mapped to machines in node n's partial solution. Let $h(n)$ 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

$$f(n) = g(n) + h(n). \tag{2}$$

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 metatask (the set U).

The function $h(n)$ is defined in terms of two functions, $h_1(n)$ and $h_2(n)$, which are two different approaches to deriving a lower-bound estimate. Recall that $M = \{\min_{0 \leqslant j < \mu}(ct(t_i, m_j))$, for each $t_i \in U\}$. For node n let $mmct(n)$ be the overall maximum element of M (i.e., the maximum minimum completion time). Intuitively, $mmct(n)$ represents the best possible metatask 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 [9], $h_1(n)$ is defined as

$$h_1(n) = \max(0, (mmct(n) - g(n))). \tag{3}$$

Next, let $sdma(n)$ 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:

$$sdma(n) = \sum_{j=0}^{m-1} (g(n) - mat(m_j)). \tag{4}$$

Intuitively, $sdma(n)$ represents the collective amount of machine availability time remaining that can be mapped without increasing the final makespan.

Let $smet(n)$ be defined as the sum over all tasks in U of the minimum expected execution time (i.e., ETC value) for each task in U:

$$smet(n) = \sum_{t_i \in U} (\min_{0 \leqslant j < \mu} (ETC(t_i, m_j)). \tag{5}$$

This gives a lower bound on the amount of remaining work to do, which could increase the final makespan. The function h_2 is then defined as

$$h_2(n) = \max(0, (smet(n) - sdma(n))/\mu), \tag{6}$$

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

$$h(n) = \max(h_1(n), h_2(n)), \tag{7}$$

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

Thus, after the root node generates μ nodes for t_0 (each node mapping t_0 to a different machine), the node with the minimum $f(n)$ generates its μ children, and so on, until 1024 nodes are created. From that point on, any time a node is added, the tree is pruned by deactivating the leaf 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.

3.3. *Concluding Remarks*

This set of 11 static mapping heuristics is not exhaustive, nor is it meant to be. It is simply a representative set of several different approaches, including iterative, noniterative, greedy, and biologically-inspired techniques. Several other types of static mapping heuristics exist. For example, other techniques that have been or could be used as static mappers for heterogeneous computing environments include the following: neural networks [8], linear programming [11], the Mapping Heuristic (MH) algorithm [13], the Cluster-M technique [14], the Levelized Min Time (LMT) algorithm [24], the *k*-percent best (KPB) and Sufferage heuristics [29], the Dynamic Level Scheduling (DLS) algorithm [38], recursive bisection [39], and the Heterogeneous Earliest-Finish-Time (HEFT) and Critical-Path-on-a-Processor (CROP) techniques [43]. The 11 heuristics examined here were selected because they seemed among the most appropriate for the static mapping of metatasks and covered a wide range of techniques.

4. EXPERIMENTAL RESULTS

4.1. *Introduction*

An interactive software application has been developed that allows simulation, testing, and demonstration of the heuristics examined in Section 3, applied to the metatasks defined by the *ETC* matrices described in Section 2. The software allows a user to specify τ and μ, to select which type of *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 Figs. 3 through 8. The results discussed in this section were generated using this software.

The makespans (i.e., the time it would take for a given metatask to complete on the heterogeneous environment) from the simulations for selected cases of consistency, task heterogeneity, and machine heterogeneity are shown in Figs. 3 through 8. Tables 1 through 3 show sample 8×8 subsections from three different types of 512×16 ETC matrices. Results for all cases are in [5]. A representative set of cases and results are discussed in this section. As indicated below, omitted cases are similar to those discussed in the rest of this section.

All experimental results represent the average makespan for a metatask of the defined type of ETC matrix. For each heuristic and each type of ETC matrix, the results were averaged over 100 different ETC matrices of the same type (i.e., 100 mappings).

The range bars for each heuristic (in Figs. 3 through 8) show the 95% confidence interval [25] (min, max) for the average makespan. This interval represents the likelihood that makespans of mappings for that type of heterogeneity and heuristic fall within the specified range. That is, if another ETC matrix (of the same type) was generated, and the specified heuristic generated a mapping, then the makespan of the mapping would be within the given confidence interval with 95% certainty.

4.2. *Heuristic Execution Times*

When comparing mapping heuristics, the execution time of the heuristics themselves is an important consideration. For the 11 heuristics that were compared, the execution times varied greatly. The experimental results discussed below were obtained on a Pentium II 400 MHz processor with 1 Gbyte of RAM. The heuristic execution times are the average time each heuristic took to compute a mapping for a single 512 task $\times$ 16 machine ETC matrix, averaged over 100 different matrices (all for inconsistent, high task, high machine heterogeneity).

The first three heuristics described, OLB, MET, and MCT, each of which has asymptotic complexity of $O(\mu\tau)$, executed for less than 1 μs per ETC matrix. Next, the Min–min, Max–min, and Duplex heuristics, each with asymptotic complexity $O(\mu\tau^2)$, executed for an average of 200 ms. The GA, which provided the best results (in terms of makespan), had an average execution time of 60 s. GSA, which uses many procedures similar to the GA, had an average execution time of 69 s. As described in the previous section, SA and Tabu were adapted to provide a fairer comparison with the results of the GA, so their average execution times were also approximately 60 s per ETC matrix. Finally A*, which has exponential complexity, executed in an average of over 20 min (1200 s).

4.3. *Consistent Heterogeneity*

The results for the metatask execution times for two consistent cases are shown in Figs. 3 and 4. Results for both cases of high machine heterogeneity (i.e., high and low task heterogeneity) were similar. Results for both cases of low machine heterogeneity were also similar. Therefore only one figure from each type of machine heterogeneity is shown. The differences in magnitude on the y-axis among the graphs are from the different ranges of magnitude used in generating the different types of ETC matrices.

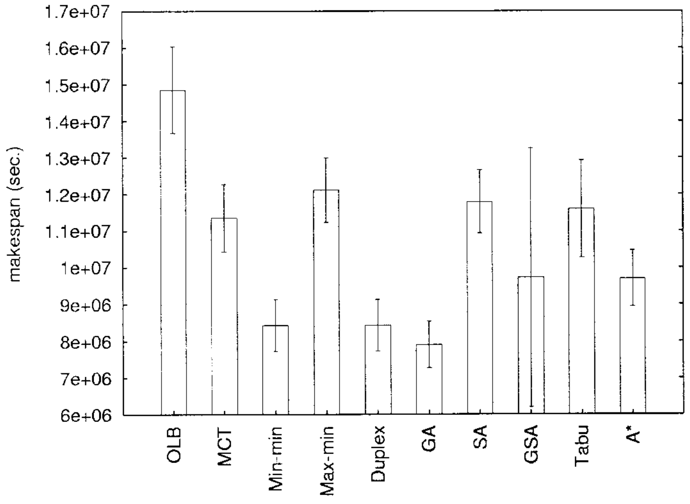

FIG. 3. Consistent, high task, high machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

For the two high machine heterogeneity cases, the relative performance order of the heuristics from best to worst was: (1) GA, (2) Min–min, (3) Duplex, (4) A*, (5) GSA, (6) MCT, (7) Tabu, (8) SA, (9) Max–min, (10) OLB, and (11) MET. For both cases of low machine heterogeneity, the relative performance order of the heuristics from best to worst was: (1) GA, (2) Min–min, (3) Duplex, (4) GSA, (5) A*, (6) Tabu, (7) MCT, (8) SA, (9) Max–min, (10) OLB, and (11) MET. For consistent ETC matrices, the MET algorithm mapped all tasks to the same machine, resulting in the worst performance by an order of magnitude. Therefore, MET is not included in the figures for the consistent cases. The performance of the heuristics will be discussed in the order in which they appear in the figures.

For all four consistent cases, OLB gave the second worst results (after MET). In OLB, the first μ tasks get assigned, one each, to the μ idle machines. Because of the consistent ETC matrix, there will be some very poor initial mappings (tasks $\mu - 2$

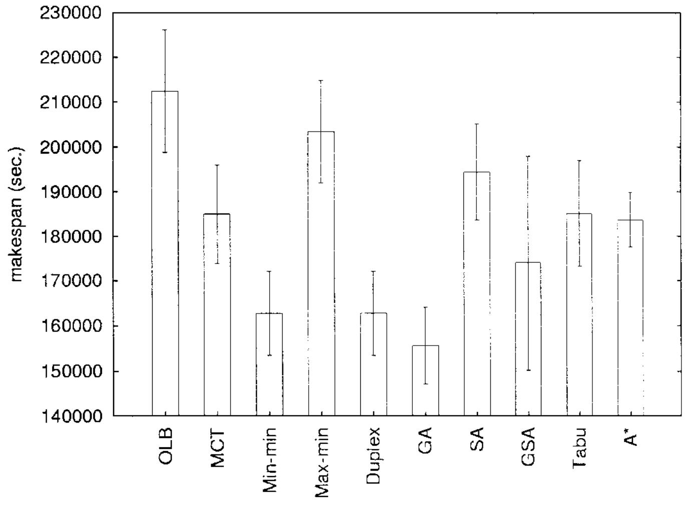

FIG. 4. Consistent, high task, low machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

and $\mu - 1$, for example, get their worst machines). Because task execution times are not considered, OLB may continue to assign tasks to machines where they execute slowly, hence the poor makespans for OLB.

MCT always performed around the median of the heuristics, giving the sixth best (low machine heterogeneity) or seventh best (high machine heterogeneity) results. MCT only makes one iteration through the ETC matrix, assigning tasks in the order in which they appear in the ETC matrix; hence it can only make mapping decisions of limited scope, and it cannot make globally intelligent decisions like Min–min or A*.

The Min–min heuristic performed very well for consistent ETC matrices, giving the second best result in each case. Not only did Min–min always give the second best mapping, but the Min–min mapping was always within 10% of the best mapping found (which was with GA, discussed below). Min–min is able to make globally intelligent decisions to minimize task completion times, which also results in good machine utilization and good makespans. Similar arguments hold for the Duplex heuristic.

In contrast, the Max–min heuristic always performed poorly, giving only the ninth best mapping. Consider the state of the machine ready times during the execution of the Min–min and Max–min heuristics. Min–min always makes the assignment that changes the machine ready times by the least amount. In general, the assignment made by Max–min will change the machine ready times by a larger amount. Therefore, the values of the machine ready times for each machine will remain closer to each other when using the Min–min heuristic than when using the Max–min heuristic. Both Min–min and Max–min will assign a given task to the machine that gives the best *completion* time. However, if the machine ready times remain close to each other, then Min–min gives each task a better chance of being assigned to the machine that gives the task its best *execution* time. In contrast, with Max–min, there is a higher probability of there being relatively greater differences among the machine ready times. This results in a load balancing effect, and each task has a lower chance of being assigned to the machine that gives the task its best *execution* time.

For the heterogeneous environments considered in this study, the type of special case where Max–min may outperform Min–min (as discussed in Section 3) never occurs. Min–min found a better mapping than Max–min every time (i.e., in each of the 100 trials for each type of heterogeneity). Thus, Max–min performed poorly in this study. As a direct result, the Duplex heuristic always selected the Min–min solution, giving Duplex a tie for the second best solution. (Because Duplex always relied on the Min–min solution, it is listed in third place.)

GA provided the best mappings for the consistent cases. This was due in large 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. However, the additional searching capabilities afforded to GA by performing crossover and mutation were beneficial, as the GA was always able to improve upon this solution by 5 to 10%.

SA, which manipulates a single solution, ranked eighth for both types of machine heterogeneity. For this type of heterogeneous environment, this heuristic (as implemented here) did not perform as well as the GA, which had similar execution time, and Min–min, which had a faster execution time. While the SA procedure is iterative (like the GA procedure), it appears that the crossover operation and selection procedure of the GA are advantageous for this problem domain.

The mapping found by GSA was either the fourth best (low machine heterogeneity) or the fifth best (high machine heterogeneity) mapping found, alternating with A*. GSA does well for reasons similar to those described for GA. The average makespan found by GSA could have been slightly better, but the results were hindered by a few very poor mappings that were found. These very poor mappings result in the large confidence intervals found in the figures for GSA. Thus, for these heterogeneous environments, the selection method from GA does better than the method from GSA.

Tabu was always the sixth or seventh best mapping (alternating with MCT). As noted in the previous section, because of the short hop procedure implemented and the structure of the consistent matrices, Tabu finds most of the successful short hops right away and must then perform a large number of unsuccessful short hops (recall machine m_i outperforms machine m_{i+1} for the consistent cases). Because the stopping criterium is determined by the number of successful hops, and because each short hop procedure has few successful hops, more successful long hops are generated, and more of the solution space is searched. Thus, Tabu performs better for consistent matrices than for inconsistent ones.

Considering the order of magnitude difference in execution times between A* and the other heuristics, the quality of the mappings found by A* (as implemented here) was disappointing. The A* mappings alternated between fourth and fifth best with GSA. 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 partially-consistent cases. For consistent cases, $h_1(n)$ underestimates the competition for machines and $h_2(n)$ overestimates the number of tasks that can be assigned to their best machine.

4.4. *Inconsistent Heterogeneity*

Two inconsistent test cases are shown in Figs. 5 and 6. For the two high machine heterogeneity cases (i.e., high and low task heterogeneity), the relative performance order of the heuristics from best to worst was: (1) GA, (2) A*, (3) Min–min, (4) Duplex, (5) MCT, (6) MET, (7) SA, (8) GSA, (9) Max–min, (10) Tabu, and (11) OLB. For both cases of low machine heterogeneity, the relative performance order of the heuristics from best to worst was: (1) GA, (2) A*, (3) Min–min, (4) Duplex, (5) MCT, (6) MET, (7) GSA, (8) SA, (9) Tabu, (10) Max–min, and (11) OLB.

MET performs much better than in the consistent cases, while the performance of OLB degrades. The reason OLB does better for consistent than inconsistent matrices is as follows. Consider, for example, machines m_0 and m_1 in the consistent case. By definition, all tasks assigned to m_0 will be on their best machine, and all tasks assigned to m_1 will be on their second best machine. However, OLB ignores direct consideration of the execution times of tasks on machines. Thus, for the

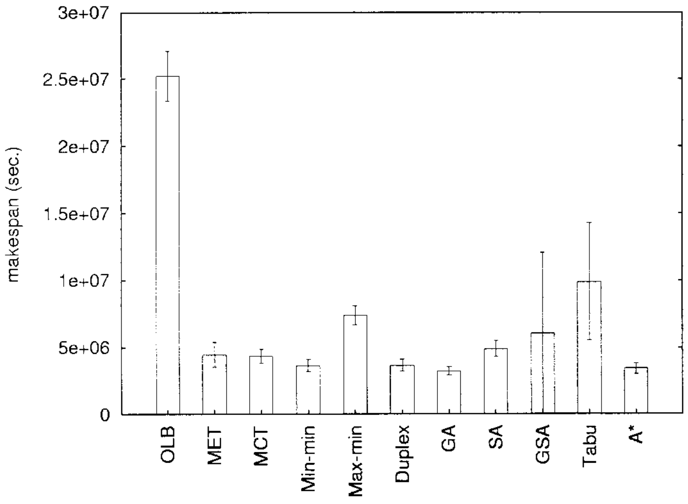

FIG. 5. Inconsistent, high task, high machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

inconsistent case, none of the tasks assigned to m_0 may be on their best machine, and none of the tasks assigned to m_1 may be on their second best machine, etc. Therefore, for each of the inconsistent cases, it is more likely that OLB will assign more tasks to poor machines, resulting in the worst mappings. In contrast, MET improves and finds the sixth best mappings 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, MCT can also exploit the fact that the machines providing the best task completion times are more evenly distributed among the set of machines. Thus, by assigning each task, in the order specified by the ETC matrix, to the machine that completes it the soonest, there is a better chance of assigning a task to a machine that executes it well, decreasing the overall makespan.

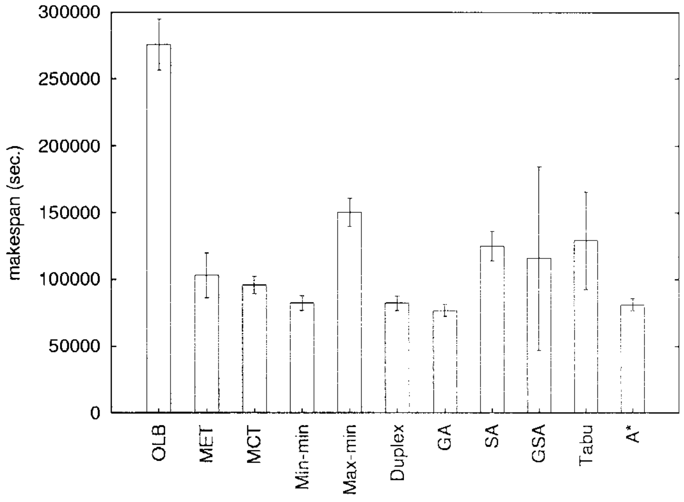

FIG. 6. Inconsistent, high task, low machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

Min–min continued to give better results than Max–min (which ranked ninth or tenth), by a factor of about two for all of the inconsistent cases. In fact, Min–min was again one of the best of all 11 heuristics, giving the third best mappings, which produced makespans that were still within 12% of the best makespans found. As noted earlier, Duplex selected the Min–min solution in every case and so ranked fourth.

GA provided the best mappings for the inconsistent cases. GA was again able to benefit from the performance of Min–min, as the best solution always came from one of the populations seeded with the Min–min solution. GA has provided the best solution in all consistent and inconsistent cases examined, and its execution time is largely independent of any of the heterogeneity characteristics. This makes it a good general-purpose heuristic, when mapper execution time is not a critical issue.

SA and GSA had similar results, alternating between the seventh and eighth best mappings. For the high machine heterogeneity cases, SA found mappings that were better by about 25%. For the low machine heterogeneity cases, GSA found the better mappings, but only by 3 to 11%.

Tabu performs very poorly (ninth or tenth best) for inconsistent matrices when compared to its performance for consistent matrices (sixth or seventh best). The sequential procedure for generating short hops, combined with the inconsistent structure of the ETC matrices, results in Tabu finding more successful short hops and performing fewer unsuccessful short hops. Many more intermediate solutions of marginal improvement exist within an inconsistent ETC matrix. Therefore, the hop limit is reached faster because of all the successful short hops (even though the hop limit is higher). Thus, less of the solution space is searched, and the result is a poor solution. That is, for the inconsistent case, the ratio of successful short hops to successful long hops increases, as compared to the consistent case, and fewer areas in the search space are examined.

A* had the second best average makespans, behind GA, and both of these methods produced results that were usually within a small factor of each other. A* did well because if the machines with the fastest execution times for different tasks are more evenly distributed, the lower-bound estimates of $h_1(n)$ and $h_2(n)$ are more accurate.

4.5. *Partially-Consistent Heterogeneity*

Finally, consider the partially-consistent cases in Figs. 7 and 8. For the high task, high machine heterogeneity cases, the relative performance order of the heuristics from best to worst was: (1) GA, (2) Min–min, (3) Duplex, (4) A*, (5) MCT, (6) GSA, (7) SA, (8) Tabu, (9) Max–min, (10) OLB, and (11) MET. The rankings for low task, high machine heterogeneity were similar to high task, high machine heterogeneity, except GSA and SA are switched in order. For both cases of low machine heterogeneity (i.e., high and low task heterogeneity), the relative performance order of the heuristics from best to worst was: (1) GA, (2) Min–min, (3) Duplex, (4) A*, (5) MCT, (6) GSA, (7) Tabu, (8) SA, (9) Max–min, (10) OLB, and (11) MET.

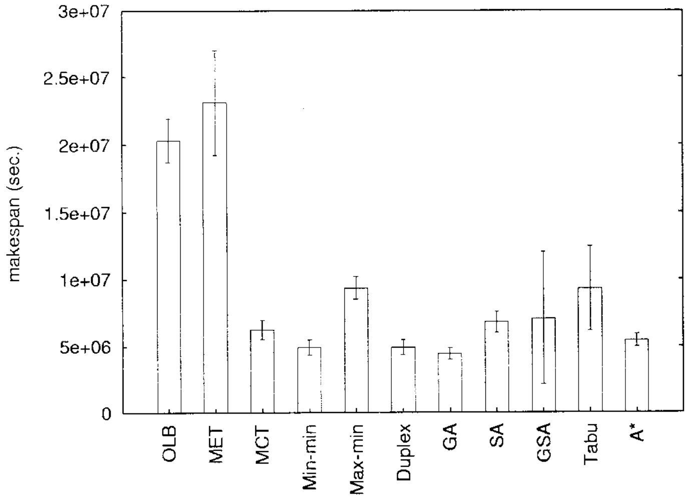

FIG. 7. Partially-consistent, high task, high machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

The MET performed the worst for every partially-consistent case. Intuitively, MET 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 bad assignments will have higher execution times for high machine heterogeneity. For low machine heterogeneity, the bad assignments have a much lower penalty. In all four cases, OLB was the second worst approach.

MCT again performs relatively well (fifth best) because the machines providing the best task completion times are more evenly distributed than the consistent case among the set of machines. Max–min continued to do poorly and ranked ninth. The Duplex solutions were the same as the Min–min solutions and tied for second

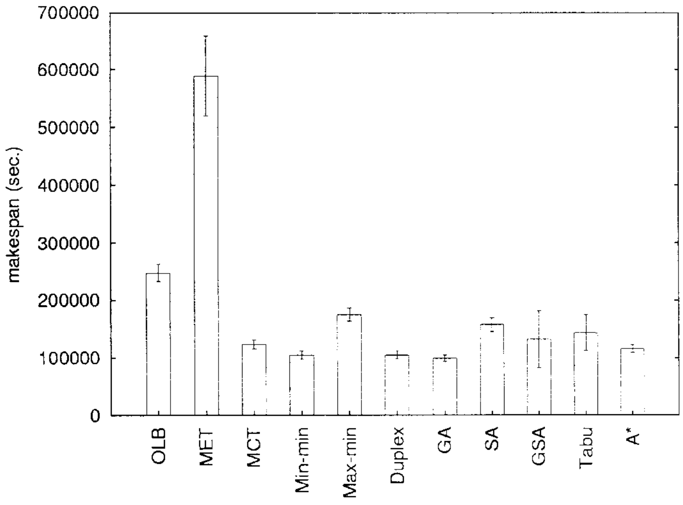

FIG. 8. Partially-consistent, high task, low machine heterogeneity ($\tau = 512$) and ($\mu = 16$).

best. The rankings for SA, GSA, and Tabu were approximately the averages of what they were for the consistent and inconsistent cases, as might be expected.

The best heuristics for the partially-consistent cases were GA (best) and Min–min (second best), followed closely by A* (fourth best, after Duplex). This is not surprising because these were among the best heuristics from the consistent and inconsistent cases, and partially-consistent matrices are a combination of consistent and inconsistent matrices. Min–min was able to do well because its approach assigned a high percentage of tasks to their first choice of machines. A* was robust enough to handle the consistent components of the matrices and did well for the same reasons mentioned for inconsistent matrices. GA maintained its position as best heuristic. The execution time and performance of GA is largely independent of heterogeneity characteristics. The additional regions of the solution space that are searched by the GA mutation and crossover operations are beneficial, as they were always able to improve on the Min–min solution by 5 to 10%.

4.6. *Summary*

To summarize the findings of this section, for consistent ETC matrices, GA gave the best results, Min–min the second best, and MET gave the worst. When the ETC matrices were inconsistent, OLB provided the poorest mappings while the mappings from GA and A* performed the best. For the partially-consistent cases, GA still gave the best results, followed closely by Min–min and A*, while MET had the slowest. All results were for metatasks with $\tau = 512$ tasks executing on $\mu = 16$ machines, averaged over 100 different trials.

For the situations considered in this study, the relative performance of the mapping heuristics varied based on the characteristics of the HC environments. The GA always gave the best performance. If mapper execution time is also considered, Min–min gave excellent performance (within 12% of the best) and had a very small execution time. The confidence intervals derived from the mappings for these two heuristics were among the best (smallest) of any of the 11 heuristics. GA was always within $\pm 9\%$ of its mean and Min–min was always within $\pm 13\%$ of the mean for all 12 cases. This means that, for any future metatask to be mapped, these two heuristics will generate a good makespan (within the confidence interval) 95% of the time.

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 and/or preliminary experiments that were conducted. However, for these heuristics, several different, valid implementations are possible using different control parameters and control functions. Some of these control parameters and control functions are listed below for selected heuristics.

GA: Several control parameter values could be varied in the GA, including population size, crossover probability, mutation probability, stopping criteria, and number of initial populations considered per result. Specific functions within GA controlling the progress of the search that could be changed are initial population seed generation, mutation, crossover, selection, and elitism.

SA: Parameter values with SA that could be modified are system temperature, cooling rate, stopping criteria, and the number of runs per result. Adaptable control procedures in SA include the initial population seed generation, mutation, and the equation for deciding when to accept a poorer solution.

GSA: Like the two heuristics it is based upon, GSA also has several parameters that could be varied, including population size, crossover probability, mutation probability, stopping criteria, cooling rate, 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: initial population seed generation, mutation, crossover, selection, and the equation for deciding when to accept a poorer solution.

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 [12]. 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., [34]).

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

Static mapping is useful in predictive analyses, impact studies, and post-mortem analyses. The goal of this study was to provide a basis for comparison and insights into circumstances when one static technique will out-perform another for 11 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 11 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 consistently gave the best results. The average performance of the relatively simple Min–min heuristic was always within 12% of the GA heuristic.

The comparisons of the 11 heuristics and 12 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 against which to compare new, developing techniques.

ACKNOWLEDGMENTS

The authors thank Shoukat Ali and the anonymous referees for their comments. A preliminary version of portions of this document appear in the proceedings of the *8th IEEE Workshop on Heterogeneous Computing*.

REFERENCES

1. S. Ali, T. D. Braun, H. J. Siegel, and A. A. Maciejewski, Heterogeneous computing, *in* "Encyclopedia of Distributed Computing" (J. Urbana and P. Dasgupta, Eds.), Kluwer Academic, Norwell, MA, to appear, 2001.
2. S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, Modeling task execution time behavior in heterogeneous computing systems, *Tamkang J. Science and Engineering* **3**, (Nov. 2000), 195–207.
3. R. Armstrong, D. Hensgen, and T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, *in* "7th IEEE Heterogeneous Computing Workshop (HCW '98)," pp. 79–87, 1998.
4. R. Armstrong, "Investigation of Effect of Different Run-Time Distributions on SmartNet Performance" (D. Hensgen, advisor), Master's Thesis, Department of Computer Science, Naval Postgraduate School, 1997.
5. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison Study of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," Technical Report TR-ECE-00-4, School of Electrical and Computer Engineering, Purdue University, Mar. 2000.
6. T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems, *in* "17th IEEE Symposium on Reliable Distributed Systems," pp. 330–335, 1998.
7. H. Chen, N. S. Flann, and D. W. Watson, Parallel genetic simulated annealing: A massively parallel SIMD approach, *IEEE Trans. Parallel Distrib. Comput.* **9**, 2 (Feb. 1998), 126–136.
8. R.-M. Chen and Y.-M. Huang, Multiconstraint task scheduling in multi-processor systems by neural networks, *in* "10th IEEE Conference on Tools with Artificial Intelligence," pp. 288–294, 1998.
9. K. Chow and B. Liu, On mapping signal processing algorithms to a heterogeneous multiprocessor system, *in* "1991 International Conference on Acoustics, Speech, and Signal Processing (ICASSP '91)," Vol. 3, pp. 1585–1588, 1991.
10. M. Coli and P. Palazzari, Real time pipelined system design through simulated annealing, *J. Systems Architecture* **42,** 6–7 (Dec. 1996), 465–475.
11. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," MIT Press, Cambridge, MA, 1992.
12. I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, Improving search by incorporating evolution principles in parallel tabu search, *in* "1994 IEEE Conference on Evolutionary Computation," Vol. 2, pp. 823–828, 1994.

13. H. El-Rewini and T. G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, *J. Parallel Distrib. Comput.* **9**, 2 (June 1990), 138–153.

14. M. M. Eshaghian, Ed., “Heterogeneous Computing,” Artech House, Norwood, MA, 1996.

15. D. Fernandez-Baca, Allocating modules to processors in a distributed system, *IEEE Trans. Software Engrg.* **15**, 11 (Nov. 1989), 1427–1436.

16. I. Foster and C. Kesselman, “The Grid: Blueprint for a New Computing Infrastructure,” Morgan Kaufman, New York, 1998.

17. 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, *in* “7th IEEE Heterogeneous Computing Workshop (HCW ’98),” pp. 184–199, 1998.

18. R. F. Freund and H. J. Siegel, Heterogeneous processing, *IEEE Comput.* **26**, 6 (June 1993), 13–17.

19. A. Ghafoor and J. Yang, Distributed heterogeneous supercomputing management system, *IEEE Comput.* **26**, 6 (June 1993), 78–86.

20. F. Glover and M. Laguna, “Tabu Search,” Kluwer Academic, Boston, MA, 1997.

21. 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, *in* “8th IEEE Workshop on Heterogeneous Computing Systems (HCW ’99),” pp. 184–198, 1999.

22. J. H. Holland, “Adaptation in Natural and Artificial Systems,” University of Michigan Press, Ann Arbor, MI, 1975.

23. O. H. Ibarra and C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, *J. Assoc. Comput. Mach.* **24**, 2 (Apr. 1977), 280–289.

24. M. Iverson, F. Ozguner, and G. Follen, Parallelizing existing application in a distributed heterogeneous environment, *in* “4th IEEE Heterogeneous Computing Workshop (HCW ’95),” pp. 93–100, 1995.

25. R. Jain, “The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling,” Wiley, New York, 1991.

26. M. Kafil and I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, *IEEE Concurrency* **6**, 3 (July–Sept. 1998), 42–51.

27. A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. L. Wang, Heterogeneous computing: Challenges and opportunities, *IEEE Comput.* **26**, 6 (June 1993), 18–27.

28. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by simulated annealing, *Science* **220**, 4598 (May 1983), 671–680.

29. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, *J. Parallel Distrib. Comput.* **59**, 2 (Nov. 1999), 107–121.

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

31. Z. Michalewicz and D. B. Fogel, “How to Solve It: Modern Heuristics,” Springer-Verlag, New York, 2000.

32. M. Pinedo, “Scheduling: Theory, Algorithms, and Systems,” Prentice–Hall, Englewood Cliffs, NJ, 1995.

33. G. Rudolph, Convergence analysis of canonical genetic algorithms, *IEEE Trans. Neural Networks* **5**, 1 (Jan. 1994), 96–101.

34. S. J. Russell and P. Norvig, “Artificial Intelligence: A Modern Approach,” Prentice–Hall, Englewood Cliffs, NJ, 1995.

35. C.-C. Shen and W.-H. Tsai, A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion, *IEEE Trans. Comput.* **34**, 3 (Mar. 1985), 197–203.

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

37. 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.), pp. 1886–1909, CRC Press, Boca Raton, FL, 1997.

38. G. C. Sih and E. A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, *IEEE Trans. Parallel Distrib. Systems* **4** (Feb. 1993), 175–186.

39. H. D. Simon and S.-H. Teng, How good is recursive bisection? *SIAM J. Sci. Comput.* **18**, 5 (Sept. 1997), 1436–1445.

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

41. M. Srinivas and L. M. Patnaik, Genetic algorithms: A survey, *IEEE Comput.* **27**, 6 (June 1994), 17–26.

42. Y. G. Tirat-Gefen and A. C. Parker, MEGA: An approach to system-level design of application specific heterogeneous multiprocessors, *in* "5th IEEE Heterogeneous Computing Workshop (HCW '96)," pp. 105–117, 1996.

43. H. Topcuoglu, S. Hariri, and M.-Y. Wu, Task scheduling algorithms for heterogeneous processors, *in* "8th IEEE Heterogeneous Computing Workshop (HCW '99)," pp. 3–14, 1999.

44. L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, *J. Parallel Distrib. Comput.* **47**, 1 (Nov. 1997), 1–15.

45. A. Y. Zomaya and R. Kazman, Simulated annealing techniques, *in* "Algorithms and Theory of Computation Handbook" (M. J. Atallah, Ed.), pp. 37-1–37-19, CRC Press, Boca Raton, FL, 1999.

TRACY D. BRAUN is a Ph.D. 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 the 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 Silicon Graphics Inc./Cray Research. His research interests include heterogeneous computing, algorithm design, computer security, and biological simulation. Beginning April 2001, he will be a senior research scientist at NOEMIX.

HOWARD JAY SIEGEL is a professor in the School of Electrical and Computer Engineering at Purdue University; beginning August 2001, he will hold the endowed chair position of Abell Distinguished Professor of Electrical and Computer Engineering at Colorado State University. He is a fellow of the IEEE and a fellow of the ACM. He received two BS degrees from MIT, and an MA, MSE, and Ph.D. from Princeton University. Professor Siegel has coauthored over 280 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 an UltraSPARC Verification Engineer at the Sun Microsystems Boston Design Center. He received a Bachelor of Science in computer engineering in 1997 and a Master of Science in electrical engineering in 1999 from Purdue University. His research interests include microprocessor architecture and verification, parallel computing, and heterogeneous computing.

LADISLAU L. BÖLÖNI is a Ph.D. 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 the 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 in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an MSEE in 1994 and a Ph.D. 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 Ph.D. 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 the 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 a 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 currently an assistant professor at the University of Illinois at Chicago in the Electrical Engineering and Computer Science Department. Dr. Theys received a Ph.D. in 1999 from the School of Electrical and Computer Engineering at Purdue University. In addition, he received a Master of Science in electrical engineering in 1996, and a Bachelor of Science in computer and electrical engineering in 1993, both from Purdue University. His current research interests include distributed computing, heterogeneous computing, parallel processing, VLSI design, and computer architecture. Dr. Theys has published several journal papers and also had several documents reviewed and accepted at conferences such as the International Conference on Parallel Processing and the Heterogeneous Computing Workshop. Dr. Theys has received support from Defense Advanced Research Projects Agency (DARPA), Intel, Microsoft, and the Armed Forces Communications and Electronics Association (AFCEA). Dr. Theys is a member of the IEEE, IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

BIN YAO is a Ph.D. student and research assistant in the School of Electrical and Computer Engineering at Purdue University. He receive Bachelor of Science in electrical engineering from Beijing University in 1996. He received an 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 a member of the Research and Evaluation Team at OpenTV in Mountain View, California. OpenTV produces middleware for set-top boxes in support of interactive television. She received her Ph.D. in the area of distributed operating systems from the University of Kentucky. Prior to moving to private industry, she was an associate professor in the systems area at Naval Postgraduate School. She worked with students and colleagues to design and develop tools and systems for resource management, network rerouting algorithms and systems that preserve quality of service guarantees and visualization tools for performance debugging of parallel and distributed systems. 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 SRAM 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.