J. Parallel Distrib. Comput. 68 (2008) 1504–1516

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Static resource allocation for heterogeneous computing environments with tasks having dependencies, priorities, deadlines, and multiple versions

Tracy D. Braun [a], Howard Jay Siegel [b,c,*], Anthony A. Maciejewski [b], Ye Hong [b]

[a] School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, USA
[b] Electrical and Computer Engineering Department, Colorado State University, Fort Collins, CO 80523, USA
[c] Computer Science Department, Colorado State University, Fort Collins, CO 80523, USA

ARTICLE INFO

*Article history:*
Received 5 June 2007
Received in revised form
14 March 2008
Accepted 3 June 2008
Available online 20 June 2008

*Keywords:*
Deadlines
Heterogeneous computing
Genetic algorithms
Mapping
Min–min
Priorities
Resource allocation
Resource management
Scheduling

ABSTRACT

Heterogeneous computing (HC) environments composed of interconnected machines with varied computational capabilities are well suited to meet the computational demands of large, diverse groups of tasks. One aspect of resource allocation in HC environments is matching tasks with machines and scheduling task execution on the assigned machines. We will refer to this matching and scheduling process as mapping. The problem of mapping these tasks onto the machines of a distributed HC environment has been shown, in general, to be NP-complete. Therefore, the development of heuristic techniques to find near-optimal solutions is required. In the HC environment investigated, tasks have deadlines, priorities, multiple versions, and may be composed of communicating subtasks. The best static (off-line) techniques from some previous studies are adapted and applied to this mapping problem: a genetic algorithm (GA), a GENITOR-style algorithm, and a two phase greedy technique based on the concept of Min–min heuristics. Simulation studies compare the performance of these heuristics in several overloaded scenarios, i.e., not all tasks can be executed by their deadlines. The performance measure used is the sum of weighted priorities of tasks that completed before their deadline, adjusted based on the version of the task used. It is shown that for the cases studied here, the GENITOR technique finds the best results, but the faster two phase greedy approach also performs very well.



## 1. Introduction

Mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different machines, interconnected with high-speed links, to perform collections of different tasks with diverse computational requirements (e.g., [18,34,44]). Such an environment coordinates the execution of tasks on machines within the system to exploit different capabilities among machines to attempt optimizing some performance metric [30].

Across the machines in an HC suite, the individual processors can vary by many factors, including age, CPU instruction sets, clock speeds, cache structures, cache sizes, memory sizes, and communication bandwidths. A cluster composed of different types (or models) of machines also constitutes an HC system. Alternatively, a cluster could be treated as a single machine in an HC suite. An HC system could also be part of a local area grid within an institution [16].

The act of assigning (matching) each task to a machine and ordering (scheduling) the execution of the tasks on each machine, and communication among machines is key to coordinating and exploiting an HC system to its fullest extent. This matching and scheduling is variously referred to as mapping, resource allocation, and resource management in the literature. We will refer to this process as mapping in this paper. The general mapping problem has been shown to be NP-complete (e.g., [10,24]). There are a variety of heuristic techniques that can be considered for different environments [1].

This study considers a collection of task characteristics in anticipated military environments (e.g., AICE [37,51], HiPer-D [54], and MSHN [21]). They make the HC mapping problem quite complex. Therefore, intelligent, efficient heuristics are sought to attempt to solve this recalcitrant problem.

In this study, tasks may be atomic or decomposable. Atomic tasks have no internal communications. Decomposable tasks consist of two or more communicating subtasks. Subtasks have data dependencies, but can be mapped to different machines. If there are communicating subtasks within a task, inter-machine data transfers may need to be performed when multiple machines are used.

* Corresponding author at: Electrical and Computer Engineering Department, Colorado State University, Fort Collins, CO 80523, USA.
*E-mail addresses:* tdbraun@computer.org (T.D. Braun), hj@colostate.edu (H.J. Siegel), aam@colostate.edu (A.A. Maciejewski), ye.hong@colostate.edu (Y. Hong).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.06.006

Tasks in this study have deadlines, priorities, and multiple versions. The deadlines are hard deadlines. The priority levels have associated weights to quantify their relative importance. The multiple versions of tasks provide alternatives for executing tasks. Those alternatives are of different preferences to the system users, and have different resource requirements.

The performance measure is the sum of weighted priorities of tasks that completed before their deadline, adjusted based on the version of the task executed. Simulations and performance bound calculations are used to evaluate and compare several heuristic techniques in several overloaded system scenarios, where not all tasks can complete by their deadlines.

Heuristics developed to perform mapping are often difficult to compare because of different underlying assumptions in the original study of each heuristic [1]. In [9], eleven heuristic techniques are investigated and directly compared for static mappings in a simpler environment. The much simpler environment in the [9] study has the goal of minimizing the total execution time of a set of independent tasks. These tasks do not have deadlines, priorities, versions, or inter-subtask communications. The eleven techniques cover a wide range of methods from the resource management and optimization literature. They are: Opportunistic Load Balancing [4,17,18], Minimum Execution Time [4,17], Minimum Completion Time [4], Min–min [4,17,24], Max-min [4,17,24], Duplex [4,17], Genetic Algorithm [22,36,53], Simulated Annealing [29,36,39], Genetic Simulated Annealing [43], Tabu Search [20,36], and A* [26,36,39,42].

Based on the results of that study, three static (off-line) techniques are selected, adapted, and applied to this more complex mapping problem: multiple variations of a two phase greedy method (based on the concept of Min–min), a standard genetic algorithm (GA), and the GENITOR approach [55] (a variation of the GA). Simulation studies are used to compare these heuristics in several different overloaded scenarios, i.e., not all tasks will be able to complete by their deadlines.

This research considers a type of environment, which may be found in certain military and commercial situations, where information needed to execute tasks arrives at predetermined times, e.g., when satellites are positioned appropriately to generate images of a given region or when scheduled weather updates are available [49,50]. This leads to several questions, such as: (a) if a given heterogeneous suite of machines can execute all of their tasks by their deadlines, and (b) if there are machine failures so that the system becomes overloaded, which tasks (and which version of each of these tasks) to execute based on their priorities. Because the arrival times of needed data are known *a priori*, the times at which the tasks can begin to execute (i.e., the tasks arrive) are known *a priori*. This research focuses on part (b), and designs heuristics to select tasks and their versions, based on priorities and deadlines, that will maximize the performance measure stated earlier. By conducting simulation studies with different subsets of the intended full machine suite, we can determine how much performance degradation occurs as a result of any particular machine failure. The question in (a) can be answered using the same heuristic techniques when all machines are considered to be operational.

The research in our paper makes the following contributions:

- A new HC paradigm is used. Tasks have deadlines, priorities, multiple versions, and may have communicating subtasks. Multiple overloaded scenarios are considered, i.e., not all tasks meet their deadline.
- Methods for performance bound calculations are proposed to evaluate the performances of different heuristic techniques.
- Several heuristics are developed, adapted, and applied to this version of the mapping problem.
- Customized chromosome structures and operations are developed for the GA and GENITOR.
- The results show that GENITOR finds the best solutions, compared to the GA and the two phase greedy approach; however the two phase greedy technique runs in less time.

The remainder of this paper is structured as follows. Section 2 describes the details of the HC environment. The mapping heuristics are defined in Section 3. In Section 4, the results from the simulation studies are examined. Section 5 examines some of the literature related to this work. Section 6 summarizes this research.

## 2. HC Environment

### 2.1. Static mapping heuristics

Two different types of mapping are static and dynamic mapping [1]. Static mapping is performed when tasks are mapped in an off-line planning phase, e.g., planning the mapping for tomorrow in a production environment (e.g., [9]). Static mapping techniques take a fixed set of applications, a fixed set of machines, and a fixed set of application and machine attributes as inputs and generate a single, fixed mapping (e.g., [1,9,13,14,32,34,41]). Dynamic mapping is performed when the tasks are mapped in an on-line, real-time fashion, e.g., when tasks arrive at unknown intervals and are mapped immediately (e.g., [33]).

There are several relative advantages and disadvantages of these two types of mappings. Static heuristics can typically use much more time to determine a mapping because it is being done off-line, e.g., for production environments; but static heuristics must then use estimated values for parameters such as when a machine will be available. In contrast, dynamic heuristics operate on-line, therefore they must make scheduling decisions in real-time, but have feedback for many system parameter values instead of estimates.

Static mapping is utilized for many different purposes [1]. It can be used to plan the execution of a set of tasks for a future time period (e.g., the production tasks to execute on the following day). Static mapping also is used in "what-if" predictive studies. For example, a system administrator might want to know the benefits of adding a new machine to the HC suite before purchasing it. By generating a static mapping (using estimated execution times for tasks and subtasks on the proposed new machines), and then deriving the estimated system performance for the set of applications, the impact of the new machine can be approximated. Static mapping also can be used to do a post-mortem analysis of dynamic mappers, to see how well they are performing. Dynamic mappers must be able to process applications as they arrive into the system, without knowledge of what applications will arrive next. When performing a post-mortem analysis, a static mapper can have the knowledge of all of the applications that have arrived over an interval of time (e.g., the previous day).

The development of heuristic techniques to find near-optimal mappings is an active area of research, e.g., [2,5,7–9,13,14,31,32,15,23,33,36,38,41,52,54,56]. This study focuses on static mapping heuristics.

### 2.2. Task characteristics

The static mapping heuristics in this study are evaluated using simulation experiments. It is assumed that an estimate of the execution time for each task on each machine is known *a priori* and contained within an *ETC* (estimated time to compute) matrix [3]. This is a common assumption when studying mapping heuristics (e.g., [5,19,26,31,41,45]); approaches for doing this estimation based on task profiling and analytical benchmarking of real systems are discussed in [28,34,44].

Each task $t_i$ has one of four possible weighted priorities, $p_i$. Two different priority scenarios are investigated: heavily-weighted priorities, $p_i \in \{1, 4, 16, 64\}$, and lightly-weighted priorities, $p_i \in \{1, 2, 4, 8\}$. Each value is equally likely to be assigned, where 64 or 8 represents the most important tasks.

This research assumes an oversubscribed system, i.e., not all tasks can finish by their deadline. To model this, let the arrival time for task $t_i$ be denoted $a_i$, and let the deadline for task $t_i$ be denoted $d_i$. To simulate different levels of oversubscription, two different arrival rates are used, based on a Poisson distribution [25]. For high arrival rates, the average arrival rate is 0.150 tasks per (arbitrary) unit of time and for moderate arrival rates the average is 0.075.

In this work, each task has three versions. At most one version of any task is executed, with version $v_k$ always preferred over version $v_{k+1}$, but also requiring more execution time. All versions of a task (or subtask) have the same dependencies, priority, and deadline. The execution time for each task (or subtask) and the user preference for that version (defined shortly) are the only parameters that varied among different versions.

The estimated execution time for task $t_i$, on machine $m_j$, using version $v_k$ is denoted $ETC(i, j, k)$. Thus, based on the previous assumption, $ETC(i, j, 0) > ETC(i, j, 1) > ETC(i, j, 2)$. The HC environment in the simulation study has $M = 8$ machines and $V = 3$ versions. To generate simulated execution times for version $v_0$ of each task in the *ETC* matrix, the coefficient-of-variation-based (CVB) method from [3] is used with task execution time (version 0) means of 1,000, and coefficients of variation for tasks and machines heterogeneities set to 0.6. Times for version $v_{k+1}$ of $t_i$ are randomly selected between 50% and 90% of $ETC(i, j, k)$. These parameters were based on previous research, experience in the field, and feedback from colleagues.

Task versions of lower preference are considered because, with their reduced execution times, they may be the only version possible to execute by their deadlines. Let $r_{ik}$ be the normalized user-defined preference for version $v_k$ of task $t_i$, and let $U(w, x)$ be a uniform random (floating point) number greater than $w$ and less than $x$. For the simulation studies: $r_{i0} = 1$ (most preferred), $r_{i1} = r_{i0} \times U(0, 1)$ (medially preferred), and $r_{i2} = r_{i1} \times U(0, 1)$ (least preferred). All subtasks of a task are required to use the same version.

Deadlines are assigned to each task as follows. First, for task $t_i$, the median execution time of the task, $med_i$, for all machines is found, using the $ETC(i, j, 0)$ values. Next, each task $t_i$ is randomly assigned a deadline factor, $\delta_i$, where $\delta_i \in \{1, 2, 3, 4\}$ (each value is equally likely to be assigned). Finally, the deadline for task $t_i$, $d_i$, is assigned as $d_i = (\delta_i \times med_i) + a_i$. All the subtasks within a decomposable task have just one deadline and one arrival time — those assigned to the decomposable task. A deadline achievement function, $D_i$, is also defined. Based on the mapping used, $D_i = 1$ if $t_i$ finished at or before $d_i$, otherwise $D_i = 0$.

The size and structure of the subtasks within a decomposable task are randomly generated, with between two and five subtasks per task, each with a maximum fanout of two. Subtask data transfer times also are generated using the CVB method [3], taking 5% to 12% of the average subtask execution time.

Atomic tasks and subtasks are called m-tasks (mappable tasks). The number of m-tasks to map in the simulation study is $T = 2000$, divided randomly into approximately 1000 atomic m-tasks and 1000 subtasks. Thus, there are approximately 1000 atomic tasks and 285 decomposable tasks. For evaluation purposes, there are $T_{eval} \approx 1285$ tasks. That is, $T$ is the number of atomic tasks and subtasks being mapped, but because all subtasks of a decomposable task must complete to get credit for that decomposable task, mappings are evaluated at the task level for $T_{eval}$ atomic and decomposable tasks.

### 2.3. Machine and network characteristics

The variation among the execution times for a given task across all the machines in the HC suite is the machine heterogeneity [1]. This study considers the case of high machine heterogeneity [3].

To help model and distinguish different types of HC systems, *ETC* matrices may be generated according to a specific type of consistency [3,9]. A consistent *ETC* matrix is one in which, if machine $m_j$ executes any task $t_i$ faster than machine $m_k$, then machine $m_j$ executes all tasks faster than machine $m_k$. Consistent matrices can be generated by sorting each row of the *ETC* matrix independently, with machine $m_0$ always being the fastest and machine $m_{M-1}$ 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 are generated (i.e., no consistency is enforced). Partially-consistent matrices are inconsistent matrices that include a consistent submatrix of a predefined size. Partially-consistent matrices represent the most realistic scenarios, and so are the only case examined here. The partially-consistent matrices used have consistent submatrices of size $0.25T \times 0.25M$.

Other assumptions about the machines in the HC environment studied include the following. The machines do not perform multi-tasking. Tasks cannot be preempted once they begin executing on a machine. Only the mapping heuristic can assign tasks to machines, no tasks from external sources are permitted.

For the HC suite's interconnection network, a high-speed LAN with a central crossbar switch is assumed. Each machine in the HC suite has one input data link and one output data link. Each of these links is connected to the central crossbar switch. Inter-subtask communications between different machines for subtasks within a decomposable task must be scheduled. After the destination subtask is mapped, the data transfer for the input data item is scheduled. A transfer starts at the earliest point in time from when the path from the source machine to the destination machine is free for a period at least equal to the needed transfer time. This (possibly) out-of-order scheduling [53] of the input item data transfers utilizes previously idle bandwidths of the communication links and thus could make some input data items available to some subtasks earlier than otherwise from the in-order scheduling. As a result, some subtasks could start their execution earlier, which would in turn decrease the overall task completion time. This is referred to as out-of-order scheduling of data transfers because the data transfers do not occur in the order in which destination tasks are mapped (i.e., the in-order schedule). Communications cannot be preempted or multiplexed. Although a high-speed LAN with a central crossbar switch is assumed as the underlying interconnection network, the heuristics that were implemented are easily modified for different interconnection networks.

### 2.4. Evaluation functions

To rate the quality of the mappings produced by the heuristics, a post-mapping evaluation function, $E$, is used. Assume that if any version of task $t_i$ completed, it is version $v_k$ ($k$ may differ for different tasks). Then, let $E$ be defined as

$$E = \sum_{i=0}^{T_{eval}-1} (D_i \times p_i \times r_{ik}). \quad (1)$$

Higher $E$ values represent better mappings.

A special deadline, called the evaluation point, $\epsilon$, is used as an absolute deadline beyond which tasks receive no credit for finishing execution. The evaluation point used in the simulation studies is the arrival time of the $(T_{eval} + 1)$th task, i.e., $\epsilon = a_{T_{eval}}$.

To determine $D_i$ in the evaluation function above, the task's completion time has to be determined. An m-task cannot begin executing until after its arrival time (and after any and all required input data have been received). Let the machine availability, $\text{mav}(i, j, k)$, be the earliest time (after m-task $t_i$'s arrival time) at which machine $m_j$ (1) is not reserved, (2) can receive all of subtask $t_i$'s input data, and (3) is available for a long enough interval to execute version $v_k$ of $t_i$. Then, the completion time of m-task $t_i$, version $v_k$, on machine $m_j$, denoted $ct(i, j, k)$, is $ct(i, j, k) = \text{mav}(i, j, k) + ETC(i, j, k)$. Thus, according to the definition of $\epsilon$, if $ct(i, j, k) \leq \min(d_i, \epsilon)$ then $D_i = 1$, else $D_i = 0$.

### 2.5. Upper bounds

Based on the definitions of $D_i$ and $r_{ik}$, the simple upper bound (SUB) of $E$, is $SUB = \sum_{i=0}^{T_{eval}-1} p_i$. This is the bound on $E$ when the system is not overloaded and all tasks can execute with their best version by their deadline.

A tighter upper bound, UB, for $E$ when system is oversubscribed can be calculated as follows. UB is based on the concept of value per unit time of task $t_i$ ($VT_i$),

$$VT_i = \max_{\substack{0 \leq j < M, \\ 0 \leq k < V}} \frac{p_i \times r_{ik}}{ETC(i, j, k)}. \tag{2}$$

$VT_i$ denotes the maximum value per unit time of the contribution to the evaluation function by executing task $i$.

For decomposable tasks, the value used for $ETC(i, j, k)$ in $VT_i$ is the sum of its subtasks execution times along the execution critical path. The execution critical path of a decomposable task is the largest sum of subtask execution times along any forward path of that decomposable task. The execution time used for each subtask is the minimum execution time (over all $M$ machines) for the specified version $v_k$ of the task. By minimizing the sum of subtask execution times, $VT_i$ will be maximized.

For UB, the total machine execution time considered available and allocable to tasks is $\mathcal{E} = M \times \epsilon$. Tasks are not assigned to machines, but they are allocated machine execution time from within the $\mathcal{E}$ units of time. UB sorts the tasks by $VT_i$, and then allocates machine execution time for tasks with the highest $VT_i$ first. By considering the most valuable tasks first, an upper bound is achieved. During the allocation process, the first task that requires more execution time than is still available in $\mathcal{E}$ is given a partial, prorated credit towards the UB. The pseudocode for computing UB is presented in Fig. 2.

UB is based on unrealistic assumptions. It implicitly assumes that: (1) all tasks have an arrival time $a_i = 0$, (2) there are no subtask communications, (3) each task can use the machine that provides its "best" execution time (based on $VT_i$), and (4) all tasks have a deadline $d_i = \mathcal{E}$.

UB is only valid if it cannot allocate all tasks within the allocable time $\mathcal{E}$. Otherwise, depending on arrival times and deadlines, a heuristic may be able to produce a mapping in which the unused portion of $\mathcal{E}$ is utilized to make the mapping a higher $E$ value. For example, if the time allocated to a given task is combined with the unused portion of $\mathcal{E}$, then it is possible that the given task can use a different version with longer execution time and lower value per unit time, but of higher value of contribution to $E$. This will undermine the rationale of UB based on $VT_i$ and may result in a value of $E$ higher than UB. For this simulation study environment, it is observed that if average arrival rate of tasks is greater than 0.60, then UB cannot allocate time for all tasks, and UB is a valid bound.

### 2.6. Robustness

In [2], the robustness of resource allocations is analyzed. In [40], it is proposed that three questions should be answered whenever robustness is discussed: (1) what behavior of the system makes it robust; (2) what uncertainties is the system robust against; and (3) quantitatively, exactly how robust is the system. In this study, the behavior needed for the system to be robust is for all tasks to execute with their best versions by their deadlines. The uncertainty is the subset of machines that will be operational. One way to quantify robustness in this case would be the minimum number of machines that can fail and still just be able to meet the performance requirement. However, here we extend the concept of robustness to include a measure of graceful degradation. In particular, we are concerned with situations where the performance requirement cannot be met and want to know how close we can get to the requirement. One way to quantify this for a given situation (i.e., collection of tasks and subset of machines) is to consider, for a given resource allocation, the evaluation function $E$ relative to the performance requirement $SUB$; e.g., $E/SUB$. Thus, this will provide one possible measure of graceful degradation that indicates how far a given allocation is from being robust.

## 3. Mapping heuristics

### 3.1. Greedy mapping heuristics

As a baseline for comparison, consider a greedy FIFO technique, referred to as the Minimum Current Fitness (MCF) technique. MCF considers m-tasks for mapping in ascending order of arrival time; it maps m-tasks to the machine that can complete the best (lowest) version possible by that task's deadline. If no machine/version combination can complete the m-task before its deadline, the task is not mapped. Version coherency for a decomposable task is strictly enforced. That is, the initial version considered for a subtask is the same version of any previously mapped subtasks. If the current subtasks requires a poorer version, then all previously mapped subtasks will be demoted to that version.

A two-phase fitness (TPF) greedy technique, based on the concept of Min–min [24], is proposed and applied to this mapping problem. Min–min style heuristics perform well in many situations (e.g., [9,24,33,56]). To describe TPF, let $f_i$, the task fitness for m-task $t_i$, be $f_i = D_i \times p_i \times r_{ik}$, where $t_i$ is executed using version $v_k$. The task fitness $f_i$ represents the contribution of each task $t_i$ to $E$.

Let $U$ be the set of all unmapped m-tasks. Let $UP \subset U$ consist of all unmapped subtasks whose predecessors have been mapped and all unmapped atomic tasks. The first phase of TPF finds the best machine (i.e., maximal $f_i$) for each m-task in $UP$, and then stores these m-task/machine pairs in the candidate tasks set $CT$. If two machines give the same best $f_i$ for task $t_i$, the machine that gives the minimum completion time is used.

Phase 2 of TPF selects the candidate task with the maximal $f_i$ over all $CT$, and maps this m-task to its corresponding machine. This task is then removed from $U$. Phases 1 and 2 are repeated until all m-tasks are mapped (or removed because they could not meet their deadline).

During the second phase of TPF, several m-tasks could have the same maximal $f_i$ value, requiring a method for breaking ties. Three different quantities are investigated as tie breaker functions in phase 2. The first tie breaker function is based on earlier arrival time, $a_i$. The second tie breaker function is based on the concept of urgency, i.e., how close the m-task is to missing its deadline [50]. To represent urgency at the subtask level, relative deadlines are used. A relative deadline divides the interval in which a decomposable task may execute into subintervals, which, for purposes of breaking ties, is based on the number of subtasks in the communication

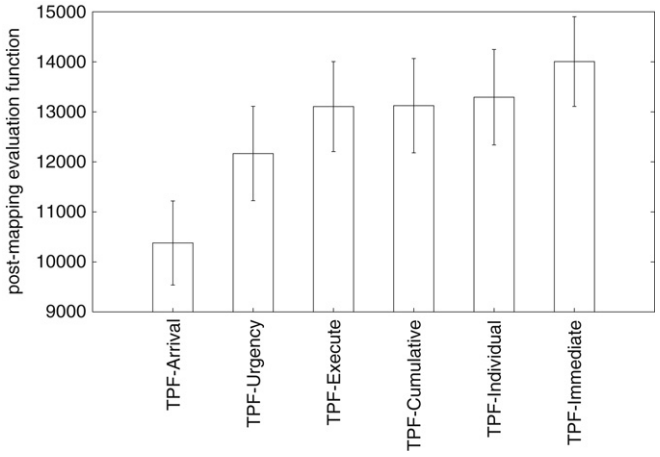


**Fig. 1.** Comparison of TPF techniques for 2000 m-tasks with high arrival rate and heavily-weighted priorities, averaged over 50 *ETC* matrices.

critical path of that task. The communication critical path is the path with the largest number of nodes (and therefore, the largest number of communications) in the decomposable task. The third tie breaker function is based on the execution time of each m-task. Tasks with shorter execution times will have a higher fitness per unit time, and will be mapped first.

A comparison of how TPF performs using the three different tie breaker functions is shown in Fig. 1. The first three bars in the figure show how TPF performs using arrival time (labeled as TPF-Arrival), urgency (labeled as TPF-Urgency), and execution time (labeled as TPF-Execute), as tie breaker functions, respectively. The results are averaged over 50 different *ETC* matrices. The range bars show the 95% confidence interval for each average [25]. TPF-Execute performed the best on average, and so is used to select the candidate task (in case of ties) in phase 2 in all remaining TPF experiments. However, its confidence interval overlaps with TPF-Urgent.

In the initial TPF implementations used to evaluate the tie breaker techniques, once a subtask is mapped, it remains in the mapping even if not all of the other subtasks in the same task can complete by their deadlines. Three versions of TPF that remove such subtasks from the mapping also are studied. These three different versions of TPF all detect partially-mapped decomposable tasks (i.e., decomposable tasks with some subtasks mapped and some subtasks still in $U$). If a partially-mapped decomposable task cannot complete all of its subtasks by its deadline, these three versions of TPF attempt to recover the resources assigned to the mapped portion of the decomposable task.

The three different versions of TPF operate as follows. TPF-Immediate unmaps subtasks right away, during phase 1, when it detects that the corresponding decomposable task cannot meet its deadline. These tasks are removed from further consideration for mapping.

TPF-Individual has an intermediate stage between phase 1 and phase 2. After phase 1, TPF-Individual temporarily unmaps the subtasks of each partially-mapped decomposable task that will not meet its deadline, one decomposable task at a time (in random order). It then attempts to map other m-tasks that missed their deadline during phase 1 (in random order). If a mapping is found, it is added to $CT$. Otherwise, the temporarily unmapped subtasks of the decomposable task are restored. (The subtasks are restored into the mapping so that the partially-mapped decomposable task is available in the future and can compete for resources when subtasks of other partially-mapped decomposable tasks are unmapped). After all partially-mapped tasks are checked, phase 2 begins.

TPF-Cumulative also has an intermediate stage between phase 1 and phase 2. After phase 1, for each partially-mapped decomposable task that will not meet its deadline, TPF-Cumulative computes $g_i = (p_i \times r_{ik})/ct(i,j,k)$ using each task's fastest (least-preferred) version to estimate a value per unit time. TPF-Cumulative also computes $h_i = (p_i \times r_{ik})/(ct(i,j,k) - d_i + 1)$ for the m-tasks that missed their deadlines in phase 1 using their fastest (least-preferred) version, where $h_i$ increases with value, and decreases with difficulty to meet the deadline. Then, TPF-Cumulative unmaps the partially-mapped tasks one at a time, in order from worst (smallest $g_i$) to best, and does not replace them. Therefore, cumulative benefits from unmapping several tasks can be found. Then, it tries to map the other m-tasks in order of best (biggest $h_i$) to worst. If a mapping is found, it is added to $CT$. After all partially-mapped tasks are unmapped, phase 2 begins.

The performance of these three versions of TPF is shown Fig. 1. Recall that all three of these versions of TPF use TPF-Execute (to break ties during phase 2). TPF-Immediate performed the best on average, and so is used to augment TPF-Execute for all remaining experiments. However, the confidence intervals of all three methods overlap.

### 3.2. Generational genetic algorithm

The steps of a general genetic algorithm (GA) are in Fig. 3. One iteration of the loop in Fig. 3 is considered one generation, i.e., life-cycle. The approach described in this subsection will be referred to as a generational GA, as it may replace the entire population at each generation [22]. This is in contrast to a steady-state GA [47], where one member of the population is replaced at a time, e.g., GENITOR, presented in the next subsection.

In GAs, a chromosome is used to represent a solution to the given problem (in this case, a mapping). The chromosome structure implemented for this study is composed of two data structures, the mapping table and the version vector. Fig. 4 shows an example of this chromosome structure. The mapping table stores matching, scheduling, and subtask dependency information. In the version vector, if $V[i] = k$, then version $v_k$ is used for task $t_i$. The GA uses a population size, $P$, of 100 chromosomes.

The mapping table is a dynamic array structure; each row could change in length as necessary. The number of rows is $M$. Each column, or index position, represents the order in which the m-tasks on each machine are to be considered for scheduling (but not necessarily the order in which they execute). Empty positions are used to maintain subtask dependencies.

A task in row $m_j$ of the mapping table is assigned to machine $m_j$. To obey data dependency constraints, a subtask's ancestors always have a lower index in the same or different row, and a subtask's descendents have a higher index.

Scheduling information is implicitly represented by the columns of the mapping table. The m-tasks are considered for actual scheduling on the machines (and network, if necessary) beginning from index 0, machine $m_0$, and then going top to bottom, left to right. That is, for column $i$, m-tasks are considered in order from $m_0$ to $m_{M-1}$, then m-tasks in column $i+1$ are considered, and so on. The m-task positions in the actual schedule (computed whenever the chromosome is evaluated) could differ from their positions in the mapping table (e.g., because of different arrival times).

Because the chromosome itself does not represent the actual, final mapping, a decoder is implicitly required. A decoder uses chromosomal information and generates the final, actual mapping (i.e., it decodes the chromosome). The decoder begins at machine $m_0$, index position zero. If there is an m-task, $t_i$, in this mapping table position, machine time on $m_0$ is reserved for this m-task, using the task version stored in $V[i]$. The decoder then considers mapping table positions in increasing order of machine numbers,

```
0  find amount of time to allocate E = M × ε;
1  for each task t_i
2      find j and k used in calculating VT_i and save them in m_i and v_i (machine and version indices);
3  end for
4  sort tasks based on VT_i;
5  sum = 0; D_i = 0 for 0 ≤ i < T_eval; bound_valid_flag = false;
6  for each t_i (in decreasing order of VT_i)
7      etc = ETC(i, m_i, v_i);          /* m_i and v_i from line 2*/
8      if ((sum + etc) ≤ E) {
9          sum = sum + etc;             /* task is allocated time from E */
10         D_i = 1;                     /* task will contribute to E */
11     }
12     else {                           /* task may only partially contribute to E */
13         bound_valid_flag = true;     /* UB is invalid unless E cannot allocate all tasks. */
14         f = i;                       /* save task t_i */
15         break;                       /* goto line 18 */
16     }
17 end for
18 if (bound_valid_flag == false) return SUB = Σ_{i=0}^{T_eval-1} p_i;
19 E = Σ_{i=0}^{T_eval-1} (D_i × p_i × r_{iv_i});   /* Equation (1) */
20 E = E + (p_f × r_{fv_f} × (E − sum)/ETC(f, m_f, v_f));   /*partial credit for task t_f */
21 return E;
```

**Fig. 2.** Pseudocode for computing the upper bound (UB).

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

**Fig. 3.** General procedure for a genetic algorithm, based on [46].

and then increasing index positions (i.e., top down, left to right). Mathematically, the m-task in the entry for machine *j*, index *i* is considered for scheduling before the entry for machine $j+1$, index *i*; and any index *i* entry is considered before any index $i+1$ entry. At each position, the decoder reserves the earliest possible time for each m-task that can be completed by its deadline. This earliest possible time is a function of machine availability, task arrival time, and the time when all input data can be received. The reservation table storing these actual task/machine reservation times is stored separately from the mapping table. This reservation table represents the information that would be used by an actual HC system to map the m-tasks. This reservation table is also used to evaluate the fitness of the mapping in the chromosome (by determining if each task satisfied its deadline).

The decoder also considers subtask communication times and link availabilities when trying to reserve machine time for each subtask. This communication link reservation table is stored separately from the mapping table. The earliest time at which

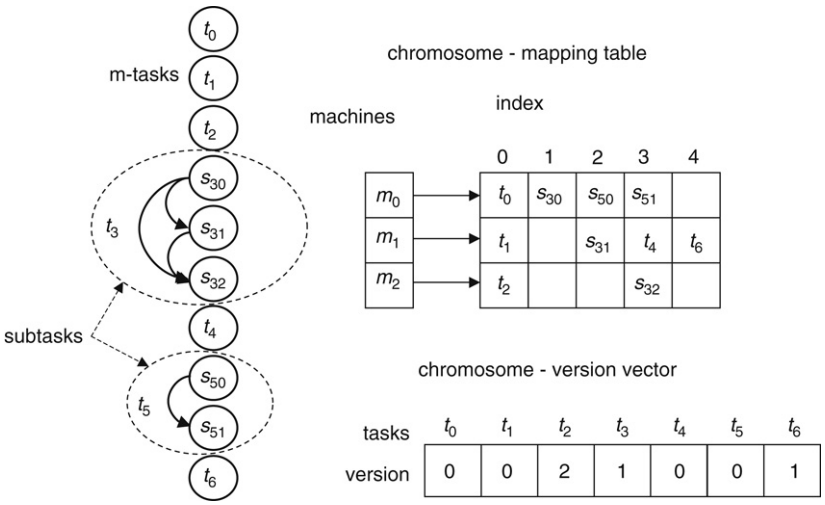


**Fig. 4.** Example chromosome structure for a seven-task, three-machine mapping.

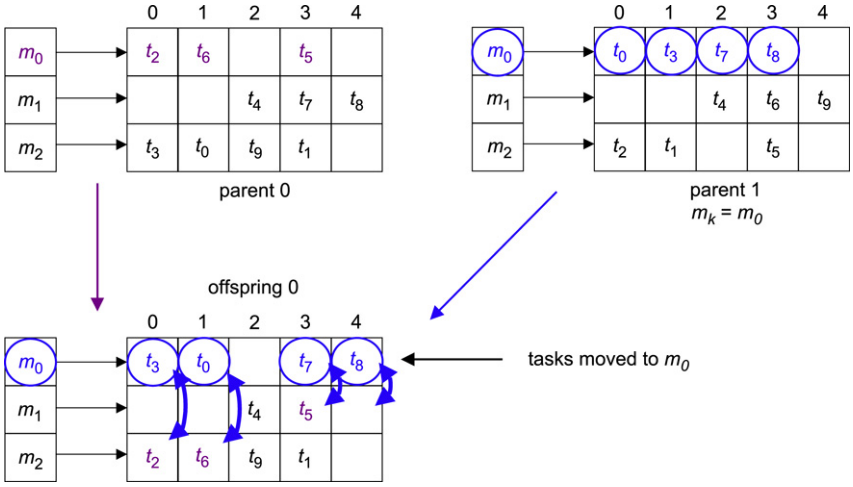


**Fig. 5.** Example of matching crossover. Offspring 0 is initially a copy of parent 0. Then, matching information from one machine (here, machine $m_0$) in parent 1 is imposed on offspring 0. To implement this, whenever possible, contents within the same column are exchanged to maintain subtask properties when subtasks are involved. For example, $t_3$ is to be moved to $m_0$, so $t_2$ is exchanged with $t_3$. If multiple conflicts occur, a new, initially empty column is inserted.

a subtask can possibly be executed is the maximum of all that subtask's input communication transfer completion times and that subtask's arrival time. (This implies that when a subtask $t_i$ is executed, all of $t_i$'s predecessors will have already executed.) Thus, the earliest time at which $t_i$ could start can be computed, and $t_i$ will be scheduled to execute at an appropriate time after that, based on the availability of its assigned machine (if it can meet its deadline). In this way, a valid mapping is always maintained and no dependency constraints are violated.

*Initial population generation*. The initial population is generated using one chromosome that is the TPF mapping and 99 random solutions (valid mappings that obey precedence constraints). For each *ETC* matrix, the GA executes multiple times, where each time a new set of 99 random solutions are generated. In particular, the GA is executed a minimum of four times, and conditionally up to eight times, stopping in less than eight times if the same overall best solution is found again. The overall best mapping found is used as the final solution.

*Evaluation*. The fitness of a chromosome is the quality of the solution, i.e., mapping, it contains. A higher quality solution has a higher fitness value. The fitness function used is $E$.

*Selection*. After evaluation, selection occurs. Stochastic universal sampling (SUS) [6] is used for the selection phase of the GA. Chromosomes in the population are sorted (ranked) based on their fitness value. If $c_i$ is the $i$th chromosome in the population, then $c_0$ is the most fit chromosome, and $c_{P-1}$ is the least fit (ties were broken arbitrarily).

Each chromosome is then allocated a sector of a roulette wheel. The size of the sector is based on the fitness, i.e., more fit chromosomes will have larger sectors. Let $A_i$ be the angle of the sector for chromosome $c_i$. Then $A_0$ (the sector for the most fit chromosome) has the largest angle, and $A_{P-1}$ (the sector for the least fit chromosome) has the smallest angle. Let $R$, the ratio of two adjacent sector angles, be a constant (greater than 1). Then $R = \frac{A_i}{A_{i+1}}$ with $0 \leq i < P-1$. Given $R$, the angles for each sector can be explicitly stated in terms of the smallest sector $A_i = R^{(P-1)-i}A_{P-1}$ where $0 \leq i < P-1$. The sum of all angles for the roulette wheel, normalized to one, can then be computed as

$$\sum_{i=0}^{P-1} A_i = R^{(P-1)}A_{P-1} \sum_{i=0}^{P-1} \frac{1}{R^i} = 1, \tag{3}$$

with $A_i = \frac{1}{R^i}A_0$. The value used for $R$ is $R = 1 + 1/P$. For $P = 100$, this approximately gives the ratio of 1.5 between the best sector and median sector in the roulette wheel, as is used in [53,55].

SUS then generates $P$ pointers around the circumference of the roulette wheel, each $1/P$ apart. (Recall that the angles around the roulette wheel are normalized to one, and $P \times (1/P) = 1$.) The chromosomes represented by the sectors on which pointers land are then included in the next generation. The first pointer is generated from a random (floating point) number between 0 and $1/P$, inclusive, $x \in U[0, 1/P]$. This simulates a spin of the roulette wheel, where $x$ is a random distance somewhere within the circumference of the first sector. Starting from $x$, the additional $P-1$ pointers are then placed, each $1/P$ apart.

Elitism, which guarantees that the the best solution found by the GA always remains in the population, is also employed [12]. Therefore, the quality of the solutions found is monotonically non-decreasing.

*Crossover*. Crossover combines elements of two parent chromosomes to generate a new offspring chromosome. To reduce the execution time of the GA, but not limit the range of possible offspring, a modification called gyre crossover is implemented. It is a form of single offspring crossover [35]. A brood of size $\mathcal{B}$ is a group of chromosomes selected from the entire population at random that is used for breeding. Each chromosome can be selected only once. Next, a random segment in each chromosome is selected. Then, in a gyral fashion, each member of the brood imposes the selected portion of itself on its the next chromosome in the brood. The result is $\mathcal{B}$, new offspring that replace the parents in the population.

Let $P_c$ be the probability of crossover for each chromosome. The initial step of each crossover procedure is to process the entire population, and for each chromosome generate a random (floating point) number $x \in U[0, 1]$. If $x \leq P_c$, that chromosome is part of a brood. If $x > P_c$, that chromosome is not in this particular crossover procedure. Each crossover procedure, i.e., matching, scheduling, and versions, selects chromosomes independently. Based on the results from [53] and initial testing, $P_c = 0.5$ is used. Brood sizes are $\mathcal{B} = 10$.

The first crossover operation, matching crossover, is illustrated in Fig. 5. The figure only shows the process for two parents and one offspring, but the basic process is the same for the entire brood. The procedure selects one machine queue, $m_k$, in one chromosome of the brood at random (e.g., $m_0$ in parent 1 in Fig. 5). Then, for the m-tasks in $m_k$, the same m-tasks are found in the partner chromosome (e.g., parent 0 in Fig. 5). Offspring 0 is initially a copy of parent 0. The m-tasks in offspring 0 are then moved so that they have the same matching as parent 1 (see Fig. 5). To implement this, whenever possible, contents within the same column are exchanged to maintain subtask properties when subtasks are

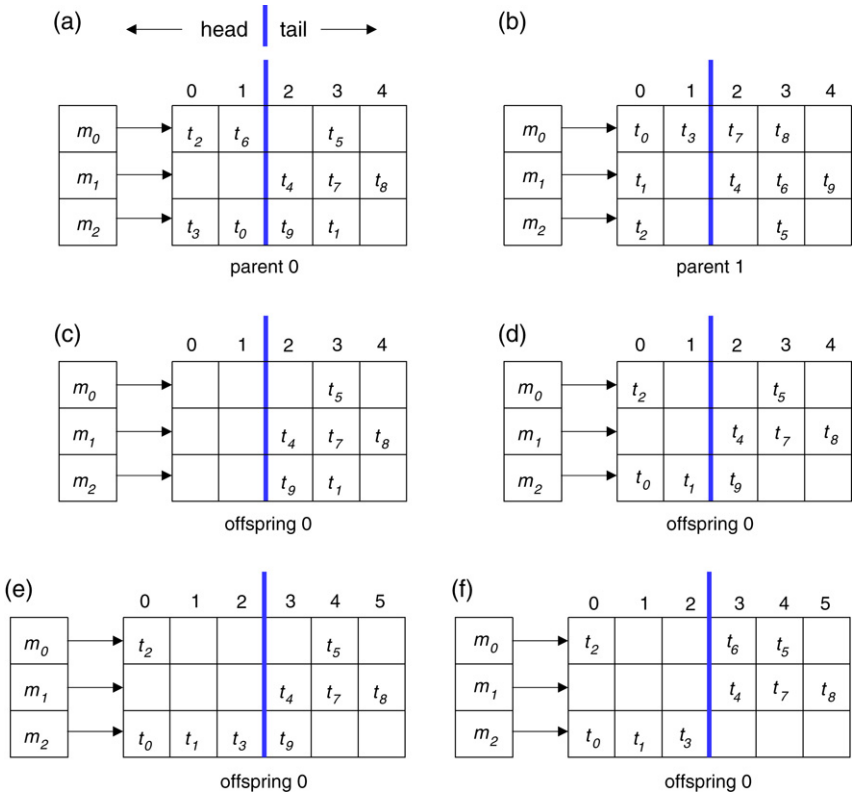


**Fig. 6.** Example of scheduling crossover. Initially the offspring is a copy of parent 0 with blank head (see (c)). Then, scheduling information from the head of parent 1 is imposed on the head of offspring 0. First, the tasks in 0th column of parent 1 are transferred to the offspring using parent 0's matching information (see (d)). Next, transfer the tasks in the 1st column of parent 1 to the offspring using parent 0's matching information (see (e)). Finally, insert the tasks from the head of parent 0 that are not yet in the offspring using parent 0's matching information (see (f)). The final offspring retains parent 0's matching information (tasks stay in the same row), but has scheduling information from the head of parent 1 (with tasks possibly moved to new columns). If multiple conflicts occur, a new, empty column is inserted. For example, column 2 is added to accommodate $t_3$ (see (e)). Extra tasks are replaced as appropriate to obey any data dependencies (e.g., $t_6$ in (f)).

involved. For example, $t_3$ is to be moved to $m_0$, so $t_2$ is exchanged with $t_3$. If multiple conflicts occur, a new, initially empty column is inserted. This matching crossover is similar to the order-based TSP crossover scheme in [48].

The next crossover procedure, scheduling crossover, is illustrated in Fig. 6. A parent chromosome and the next parent chromosome within the brood create an offspring together. A random cut for both parents is selected, dividing both chromosomes into a head and tail. Offspring 0 is initialized with a copy of parent 0. Then, the tasks in the head of offspring 0 are deleted. Tasks are extracted from the head of parent 1, and while maintaining their scheduling, are placed into the head of offspring 0 using the matching from parent 0. Extra tasks are replaced as appropriate (e.g., $t_6$ in Fig. 6). Thus, the matching information of parent 0 is preserved, but the scheduling information from the head of parent 1 is imposed. This procedure works because a valid subtask ordering is maintained at all times. These crossover procedures are similar to schemes in [11,48].

For version crossover, a gyre crossover of version vectors is used. In each parent, two crossover points are selected at random, and the middle segments are imposed on the next parent in a gyral fashion.

*Mutation.* After all three crossover procedures are completed, the mutation procedures are performed. Let $P_m$ be the probability of mutation for each chromosome. Based on the results from [53] and initial testing, $P_m = 0.5$ is used. Then, similar to crossover, the initial step of each mutation procedure is to process the entire population (which has been changed by the three types of crossover), and for each chromosome generate a random (floating point) number $x \in U[0, 1]$. If $x \leq P_m$, that chromosome is mutated. Each mutation procedure, i.e., matching, scheduling, and versions, selects chromosomes independently.

The method used to perform a matching mutation first selects a target m-task at random. Next, a new machine to which the target m-task is matched is selected at random. Finally, the contents of these two mapping table positions (within the same column) are exchanged.

The method used to perform a scheduling mutation only affects the ordering of tasks on a single machine. The first step selects a target m-task. Next, the valid range for the target m-task is determined. The valid range is the set of index positions to which the target m-task can be moved and not violate any dependency constraints [53]. Next, a new position from within the valid range is selected at random, and the contents of these two positions are exchanged. The procedures defined for matching mutations and scheduling mutations are similar to the order-based mutation reported in [35,48].

For a version mutation, a random task is selected. Then a new, different value for that task's version is randomly generated.

*Stopping Criteria.* Four stopping criteria are used for the generational GA. The GA is stopped as soon as one of these conditions is met: (1) 1,000 generations have been computed, (2) the elite chromosome has not changed for 150 generations [53], (3) all chromosomes have converged to the same mapping, or (4) the elite chromosome represents the UB solution.

### 3.3. GENITOR-based mapping heuristic

GENITOR is a steady-state genetic search algorithm that has been shown to work well for several problem domains, including job shop scheduling and parameter optimization (e.g., [52,55]). A

typical GENITOR-style GA would be implemented as follows. First, an initial population is generated and evaluated, as it is with the generational GA. Next, the entire population is sorted (ranked) by each chromosome's fitness value, and stored in this sorted order. Then, a special function is used to select two chromosomes to act as parents. The two parents perform a crossover, and a (single) new offspring is generated. This offspring is then evaluated, and must immediately compete for inclusion in the population. If the offspring has a higher fitness than the poorest member of the population, the offspring is inserted in sorted order in the population, and the poorest chromosome is removed. Otherwise, the offspring is discarded. This would continue until a stopping condition is reached.

The special function for selecting parents is a bias function, used to provide a specific selective pressure [55]. For example, a bias of 1.5 implies that the top ranked chromosome in the population is 1.5 times more likely to be selected for a crossover than the median chromosome. Elitism is implicitly implemented by always removing the poorest chromosome.

In this study, instead of individual pairs performing crossover, broods of size $\mathcal{B} = 10$ are used, to take advantage of gyre crossover and remain consistent with the generational GA. A linear bias of 1.5 is used to select ten chromosomes for crossover. The same three crossovers from the generational GA are used.

After each type of crossover, the offspring are considered for that same type of mutation. For example, after a brood undergoes a matching crossover, each new offspring is considered for matching mutation. A probability of mutation of $P_m = 0.5$ is used. After (possibly) being mutated, the new offspring are evaluated and considered for insertion into the population. The fitness of each chromosome is evaluated using $E$.

The stopping conditions for the GENITOR-style GA are: (1) 100,000 total offspring have been generated, (2) the upper 50% of the population remains unchanged for 15,000 consecutive offspring, (3) the UB is found, or (4) all chromosomes converge to the same solution. Condition (1) is based on a stopping condition from [55] and condition (2) approximates the no change in elite condition from the generational GA. Similar to the generational GA, between four and eight runs for each *ETC* matrix are performed. Although GENITOR is a steady-state GA, the GENITOR-style GA implemented in this study is not a true steady-state GA. The use of broods introduces a generational component.

## 4. Results from simulation studies

Three different HC mapping test case scenarios are examined: (1) highly-weighted priorities and high arrival rate, (2) lightly-weighted priorities and moderate arrival rate, and (3) lightly-weighted priorities and high arrival rate. Each result reported is the average of 50 different trials (i.e., *ETC* matrices), with $T = 2000$ m-tasks, $M = 8$ machines, and $V = 3$ versions. The 95% confidence interval for each heuristic is shown at the top of each bar [25], and GENITOR is abbreviated as GEN.

The average execution time on a single trial for MCF is 2 min, whereas TPF took, on average, 23 min. The GA and GENITOR heuristics are designed with similar stopping conditions; so both averaged 12 h per trial.

The left side of Fig. 7(a) shows the results for the heavily-weighted priorities, high arrival rate scenario. MCF performs the poorest, achieving only 20% of the performance of TPF. TPF performs well and achieves 62% of UB. GA is able to improve the initial TPF seeded mappings by 2%, achieving 64% of UB. GENITOR does the best of all the mapping heuristics, finding mappings that are 5% higher than TPF and achieving 67% of UB.

The right side of Fig. 7(a) shows the average performance that MCF, TPF, GA, and GENITOR achieve for the lightly-weighted priorities, high arrival rate scenario. The heuristics have the same relative performance as in the heavily-weighted priorities, high arrival rate scenario.

The right side of Fig. 7(b) shows the results for the heavily-weighted priorities, moderate arrival rate scenario. The lower-arrival rate presents an easier mapping problem than the other scenarios. Thus, each heuristic performs better at the slower arrival rate. For example, on the right side of Fig. 7(b), TPF is 76% of UB, and GENITOR is 82% of UB. Compare this with 62% and 67%, respectively, for the heavily-weighted priorities, high arrival rate case.

Fig. 8 shows a breakdown of the average number of tasks at each priority level that meet their deadline, as mapped by each heuristic. Tasks with higher priority weightings have more relative importance, and more of them should get mapped. This is substantiated by Fig. 8.

The stopping conditions encountered by each of the GA and GENITOR do not vary much among the different scenarios. For the GENITOR experiments, the most common stopping condition is reaching 100,000 total offspring, occurring for 99% of all runs (the other 1% is no change in the upper 50% of the population). For GA, the most common stopping condition is 150 generations with no change in elite, occurring for 85% of all runs. The other 15% of these GA runs encounter the stopping condition of 1000 total generations. GA and GENITOR stop after an average of 6.9 and 7.2 runs for each *ETC* matrix, respectively.

In these experiments, a mapping that can achieve the SUB or UB is impossible to create. Thus, another set of experiments is introduced, based on a non-overloaded, contrived set of data, where the optimal solution is known, and the heuristics' best solutions can be compared to optimal solutions.

For this contrived class of mapping problems, the optimal mapping has the following properties. Each task is able to execute its most preferred version. Each task is able to meet its deadline. All subtasks in a task have the same best machine (so there are no inter-machine communications). This achievable optimal solution represents a perfect packing of tasks to machines in the HC environment. The arrival rate of tasks is much slower, to allow for the perfect packing solution. This set of experiments uses the average of 25 *ETC* matrices with heavily-weighted priorities, 512 m-tasks, and 8 machines.

For these experiments, TPF performs very well. On average, TPF achieves 94.5% of the optimal solution. GA comes slightly closer (94.6%), and GENITOR the closest, 95.0% of the optimal. While not a mathematical proof, this would indicate that the heuristics may perform similarly in the overloaded experiments because they come close to the (unknown) optimal solution.

## 5. Related work

The work in [31] investigates dynamic resource allocation for tasks with priorities and multiple deadlines in an HC environment and compares the performance of mapping heuristics to two static mapping schemes, simulated annealing and a genetic algorithm. The problem differs from ours because the tasks in [31] (a) are independent (i.e., there is no inter-task communication), (b) have soft deadlines instead of one hard deadline, and (c) do not have multiple versions.

In [38], the authors propose a dynamic real-time job scheduling algorithm with hard deadlines in a heterogeneous cluster environment. The work uses a reliability-cost-driven approach to minimize the reliability cost, which is defined based on machines and links failure rate. No task priorities or versions are included in the task model.

In [32], a semi-static approach for mapping dynamic iterative tasks in HC systems is presented. The mapping technique is semi-static because it uses the off-line-derived (static) mappings in

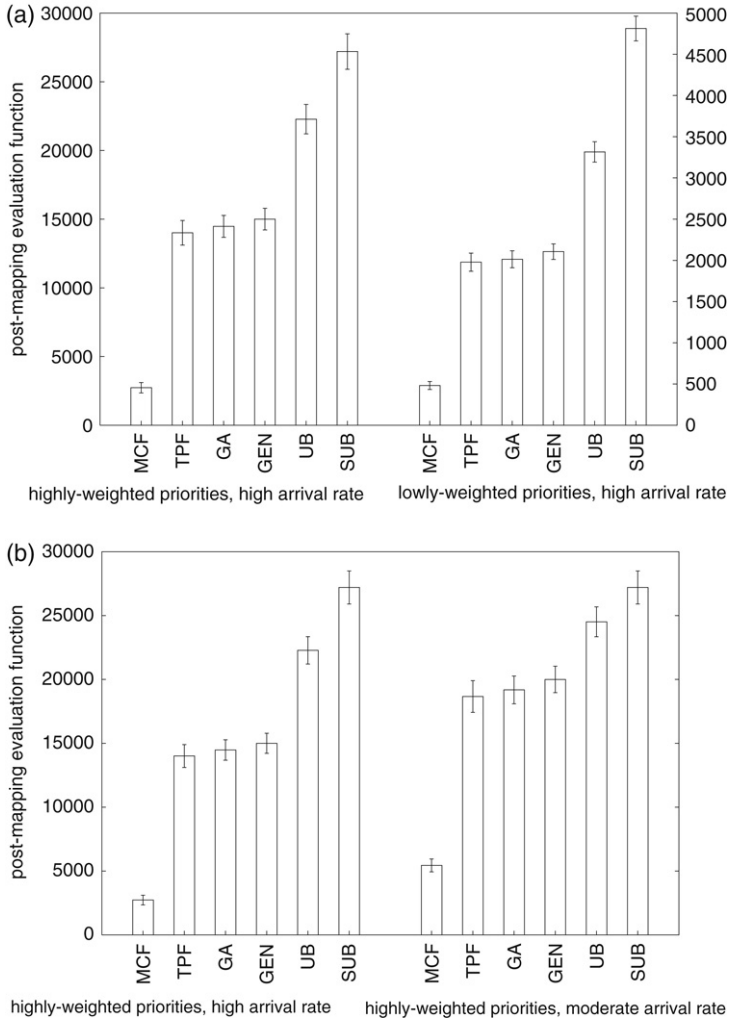


**Fig. 7.** (a) Comparison of heuristics for heavily-weighted priorities and lightly-weighted priorities, both with high arrival rate. Note that the *y*-axis scales are different for the two scenarios. (b) Comparison of heuristics for high and moderate arrival rates, both with heavily-weighted priorities.

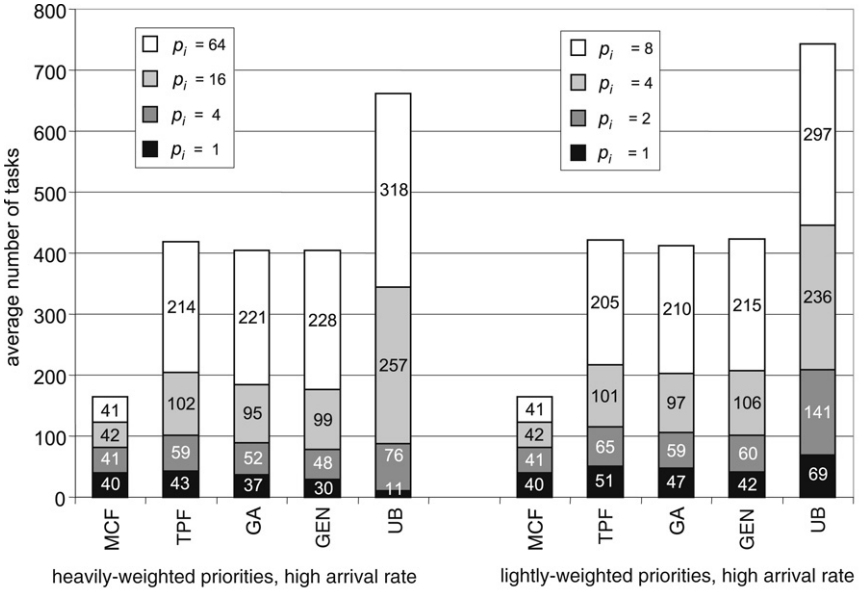


**Fig. 8.** Number of tasks at each priority level that meet their deadline for heavily-weighted priorities (left) and lightly-weighted priorities (right). For the heavily-weighted priorities scenario, highest priority tasks are more important relative to the lightly-weighted priorities scenario, and more of them meet their deadline.

a dynamic environment. The technique incorporates a statically derived mapping table as a reference and observes actual dynamic parameters to choose the most suitable mapping during the online phase. The task model used in this work does not have priorities, deadlines, or versions.

As discussed in Section 2.1, the static mapping heuristics used in this paper can typically use much more time and can have more information, e.g., task arrival times to determine a mapping, which differs with the dynamic mapping heuristics mentioned above.

In [14], Doğan and Özgüner propose a QoS-based meta-task scheduling algorithm that addresses the problem of scheduling a set of independent tasks with multiple QoS requirements, thus maximizing the system performance subject to users' constraints. While a related and very interesting work, it differs in four significant ways from ours: (a) our performance metric is different from the one they use; (b) the tasks in [14] do not have dependencies; (c) their tasks' characteristics do not include a feature analogous to our multiple versions (they have different security levels, but they are treated differently); and (d) they do not provide a performance bound for comparison with experimental results.

The work presented in [41] explores the problem of minimizing the average percentage of energy consumptions of machines to execute a single application in an *ad hoc* grid environment of mobile devices. The application considered is composed of communicating subtasks. Thus the goal differs from the work here while our goal is to maximize a post-mapping evaluation function for independent tasks (some with communication subtasks), where tasks have priorities, versions, and deadlines in a general HC environment.

In [5], Attiya and Hamam address the problem of maximizing the system reliability for static task allocation in HC environments. The reliability model proposed is similar to that in [38] and is based on probabilities that processors and paths/links for communication between processors are operational. The work uses a simulated annealing approach as the heuristic algorithm to find a near-optimal solution. The task models used in our work are different from those found in [5].

In [13], a GA-based technique is applied to the static matching and scheduling of tasks represented by a directed acyclic graph (DAG). The tasks concerned have communications, however, they do not have priorities, versions, or deadlines.

The model presented in [27] is a dynamic system of computational nodes (resources) that schedules tasks with multiple deadlines (soft and hard). The performance metric used in [27] is the makespan. This makespan incorporates the communication time to move a task from the portal of the computational grid to the computational node where the task is scheduled to execute, and (after the completion of the task) the communication time to move the results back to the portal. A comparative analysis is made among three modes of cooperation for computational nodes. It was reported that due to the incorporation of communication time in the makespan, the semi-cooperative (semi-distributed) approach provided superior performance compared to (a) the non-cooperative (current state-of-the art computational grid resource allocation mechanisms) approach and (b) the cooperative (centralized) approach. The work in [27] differs from the work in this study because (a) the system models differ — we consider tasks with priorities and multiple versions ([27] does not), (b) the performance measures differ — we use the sum of weighted priorities of tasks that completed before their deadline, adjusted based on the version of the task used (versus makespan in [27]), and (c) the environments differ — we consider an oversubscribed system where some tasks cannot complete by their deadlines (as opposed to the situation in [27] where all tasks can complete by their deadlines).

## 6. Summary

This paper presents a new paradigm for an HC environment, where tasks have deadlines, priorities, multiple versions, and communicating subtasks. Two upper bounds, multiple variations of TPF, and two kinds of genetic algorithms are implemented and compared.

It is shown that the TPF approach performs very well, achieving 59% to 76% of the upper bound. The generational GA approach, seeded with TPF, improves these mappings by only 2% to 3%, achieving 61% to 78% of the upper bound. The GENITOR approach, also seeded with TPF, finds the best mappings with an improvement of 6% to 8% over TPF, achieving 67% to 82% of the upper bound.

The heuristics are then examined against achievable optimal solutions. TPF, GENITOR, and GA are all within 5.5% of known optimal solutions in these special cases.

This study presents one implementation and application of GAs for the HC mapping problem (many others are possible). In situations where a high-quality assignment of machines to tasks is critical, their use is justified. However, the faster TPF heuristic also provides very good results.

## Acknowledgments

The authors thank Shoukat Ali, Luis Diego Briceño, David Leon Janovy, Jong-Kook Kim, Loretta Vandenberg, Samee U. Khan, and Darrell Whitley for their comments. A preliminary version of portions of this paper was presented at the 16th International Parallel and Distributed Processing Symposium.

This research was supported in part by the NSF under grant CNS-0615170, by the DARPA/ITO Quorum Program under GSA subcontract number GS09K99BH0250, by a Purdue University Dean of Engineering Donnan Scholarship, and by the Colorado State University George T. Abell Endowment. Some of the equipment used was donated by Intel, Microsoft, and Noemix.

## References

[1] S. Ali, T.D. Braun, H.J. Siegel, A.A. Maciejewski, N. Beck, L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, Characterizing resource allocation heuristics for heterogeneous computing systems, in: A.R. Hurson (Ed.), Advances in Computers Volume 63: Parallel, Distributed, and Pervasive Computing, Elsevier, Amsterdam, The Netherlands, 2005, pp. 91–128.

[2] S. Ali, A.A. Maciejewski, H.J. Siegel, J.-K. Kim, Measuring the robustness of a resource allocation, IEEE Transactions on Parallel and Distributed Systems 15 (7) (2004) 630–641.

[3] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang Journal of Science and Engineering 3 (3) (2000) 195–207. (Special Tamkang University 50th Anniversary Issue) invited.

[4] R. Armstrong, D. Hensgen, 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), Mar. 1998, pp. 79–87.

[5] G. Attiya, Y. Hamam, Task allocation for maximizing reliability of distributed systems: A simulated annealing approach, Journal of Parallel and Distributed Computing 66 (10) (2006) 1259–1266.

[6] J.E. Baker, Reducing bias and inefficiency in the selection algorithm, in: J.J. Grefenstette, ed., 2nd International Conference on Genetic Algorithms, July 1987, pp. 14–21.

[7] I. Banicescu, V. Velusamy, Performance of scheduling scientific applications with adaptive weighted factoring, 10th IEEE Heterogeneous Computing Workshop (HCW 2001), in: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), Apr. 2001.

[8] H. Barada, S.M. Sait, N. Baig, Task matching and scheduling in heterogeneous systems using simulated evolution, 10th IEEE Heterogeneous Computing Workshop (HCW 2001), in: Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), Apr. 2001.

[9] 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, R.F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[10] E.G. Coffman Jr. (Ed.), Computer and Job-Shop Scheduling Theory, John Wiley & Sons, New York, NY, 1976.

[11] L. Davis, Applying adaptive algorithms to epistatic domains, in: 9th International Joint Conference on Artificial Intelligence, Aug. 1985, pp. 162–164.

[12] K.A. DeJong, An Analysis of the Behavior of a Class of Genetic Adaptive Systems, doctoral dissertation, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1975.
[13] M.K. Dhodhi, I. Ahmad, I. Ahmad, A. Yatama, An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems, Journal of Parallel and Distributed Computing 62 (9) (2002) 1338–1361.
[14] A. Doğan, F. Özgüner, Scheduling of a meta-task with QoS requirements in heterogeneous computing systems, Journal of Parallel and Distributed Computing 66 (2) (2006) 181–196.
[15] M.M. Eshaghian (Ed.), Heterogeneous Computing, Artech House, Norwood, MA, 1996.
[16] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Second Edition, Morgan Kaufman, New York, NY, 2004.
[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, H.J. Siegel, Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet, in: 7th IEEE Heterogeneous Computing Workshop (HCW '98), Mar. 1998, pp. 184–199.
[18] R.F. Freund, H.J. Siegel, Heterogeneous processing, IEEE Computer 26 (6) (1993) Guest Editor's Introduction, pp. 13–17.
[19] A. Ghafoor, J. Yang, Distributed heterogeneous supercomputing management system, IEEE Computer 26 (6) (1993) 78–86.
[20] F. Glover, M. Laguna, Tabu Search, Kluwer Academic Publishers, 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, A. Alhusaini, An overview of MSHN: A management system for heterogeneous networks, in: 8th IEEE Workshop on Heterogeneous Computing Systems (HCW '99), Apr. 1999, pp. 184–198.
[22] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.
[23] E.-N. Huh, L.R. Welch, B.A. Shirazi, C.D. Cavanaugh, Heterogeneous resource management for dynamic real-time systems, in: 9th IEEE Heterogeneous Computing Workshop (HCW 2000), May 2000, pp. 287–294.
[24] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, Journal of the ACM 24 (2) (1977) 280–289.
[25] R. Jain, The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, New York, NY, 1991.
[26] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency 6 (3) (1998) 42–51.
[27] Samee Ullah Khan, Ishfaq Ahmad, Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation, in: 20th IEEE International Parallel and Distributed Processing Symposium, March 2006.
[28] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, C.L. Wang, Heterogeneous computing: Challenges and opportunities, IEEE Computer 26 (6) (1993) 18–27.
[29] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[30] J.-K. Kim, D.A. Hensgen, T. Kidd, H.J. Siegel, D.St. John, C. Irvine, T. Levin, N.W. Porter, V.K. Prasanna, R.F. Freund, A flexible multi- dimensional QoS performance measure framework for distributed heterogeneous systems, in: Cluster Computing in Science and Engineering, Cluster Computing 9 (3) (2006) 281–296 (special issue).
[31] J.-K. Kim, S. Shivle, H.J. Siegel, A.A. Maciejewski, T.D. Braun, M. Schneider, S. Tideman, R. Chitta, R.B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, S.S. Yellampalli, Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment, Journal of Parallel and Distributed Computing 67 (2) (2007) 154–169.
[32] Y.-K. Kwok, A.A. Maciejewski, H.J. Siegel, I. Ahmad, A. Ghafoor, A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 66 (1) (2006) 77–98.
[33] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 59 (2) (1999) 107–121.
[34] M. Maheswaran, T.D. Braun, H.J. Siegel, Heterogeneous distributed computing, in: J.G. Webster (Ed.), in: Encyclopedia of Electrical and Electronics Engineering, vol. 8, John Wiley & Sons, New York, NY, 1999, pp. 679–690.
[35] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, New York, NY, 1996.
[36] Z. Michalewicz, D.B. Fogel, How to Solve It: Modern Heuristics, Second Edition, Springer-Verlag, New York, NY, 2004.
[37] A. Pavan, V. Gopal, S. Song, N. Birch, R. Harinath, D. Castanon, Admission control and resource allocation in a strictly priority based network, in: International Conference on Real-Time Computing Systems and Applications, Dec. 2000, pp. 231–238.
[38] X. Qin, H. Jiang, A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters, Journal of Parallel and Distributed Computing 65 (8) (2005) 885–900.
[39] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 2003.
[40] V. Shestak, H.J. Siegel, Anthony A. Maciejewski, S. Ali, Robust resource allocations in parallel computing systems: Model and heuristics, in: 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2005), Dec. 2005. Invited.
[41] S. Shilve, H.J. Siegel, A.A. Maciejewski, P. Sugavanam, T. Banka, R. Castain, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, J. Velazco, Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment, Journal of Parallel and Distributed Computing 66 (4) (2006) 600–611.
[42] C.-C. Shen, W.-H. Tsai, A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion, IEEE Transactions on Computers C-34 (3) (1985) 197–203.
[43] P. Shroff, D. Watson, N. Flann, R. Freund, Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments, in: 5th IEEE Heterogeneous Computing Workshop (HCW '96), Apr. 1996, pp. 98–104.
[44] H.J. Siegel, H.G. Dietz, J.K. Antonio, Software support for heterogeneous computing, in: A.B. Tucker Jr. (Ed.), The Computer Science and Engineering Handbook, CRC Press, Boca Raton, FL, 1997, pp. 1886–1909.
[45] H. Singh, A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: 5th IEEE Heterogeneous Computing Workshop (HCW '96), Apr. 1996, pp. 86–97.
[46] M. Srinivas, L.M. Patnaik, Genetic algorithms: A survey, IEEE Computer 27 (6) (1994) 17–26.
[47] G. Syswerda, Uniform crossover in genetic algorithms, in: J. Schaffer (Ed.), 3rd International Conference on Genetic Algorithms, Morgan-Kaufmann, San Mateo, CA, 1989, pp. 2–9.
[48] G. Syswerda, Schedule optimization using genetic algorithms, in: L. Davis (Ed.), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991, pp. 332–349.
[49] M.D. Theys, N. Beck, H.J. Siegel, M. Jurczyk, An analysis of procedures and objective functions for heuristics to perform data staging in distributed systems, Journal of Interconnection Networks 7 (2) (2006) 257–293.
[50] M.D. Theys, M. Tan, N.B. Beck, H.J. Siegel, M. Jurczyk, A mathematical model and scheduling heuristics for satisfying prioritized data requests in an oversubscribed communication network, IEEE Transactions on Parallel and Distributed Systems 11 (9) (2000) 969–988.
[51] I.-J. Wang, S.D. Jones, Agile information control environment program, Johns Hopkins APL Technical Digest 20 (3) (1999) 271–275.
[52] J.-P. Watson, S. Rana, L.D. Whitley, A.E. Howe, The impact of approximate evaluation on the performance of search algorithms for warehouse scheduling, Journal of Scheduling 2 (2) (1999) 79–98.
[53] L. Wang, H.J. Siegel, V.P. Roychowdhury, A.A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, in: Parallel Evolutionary Computing, Journal of Parallel and Distributed Computing 47 (1) (1997) 8–22 (special issue).
[54] L.R. Welch, P.V. Werme, L.A. Fontenot, M.W. Masters, B.A. Shirazi, B. Ravindran, D.W. Mills, Adaptive QoS and resource panagement using *a posteriori* workload characterizations, in: 5th IEEE Real-Time Technology and Applications Symp., 1999, pp. 266–275.
[55] D. Whitley, The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best, in: D. Schaffer (Ed.), 3rd International Conference on Genetic Algorithms, Morgan Kaufmann, San Francisco, CA, 1989, pp. 116–121.
[56] M.-Y. Wu, W. Shu, H. Zhang, Segmented Min–Min: A static mapping algorithm for meta-tasks on heterogeneous computing systems, in: 9th IEEE Heterogeneous Computing Workshop (HCW 2000), May 2000, pp. 375–385.

**Tracy Braun** (CISSP, CCNA) received a Ph.D. in Electrical and Computer Engineering from Purdue University in 2001. Dr. Braun also received a Master's Degree in Electrical and Computer Engineering from Purdue University in 1997. Dr. Braun graduated with Honors and High Distinction from the University of Iowa in 1995, with a Bachelor of Science degree in Electrical and Computer Engineering. Dr. Braun's current research interests include computer security, information assurance, and reverse engineering.

**Howard Jay Siegel** was appointed the Abell Endowed Chair Distinguished Professor of Electrical and Computer Engineering at Colorado State University (CSU) in 2001, where he is also a Professor of Computer Science. He is the Director of the CSU Information Science and Technology Center (ISTeC), a university-wide organization for promoting, facilitating, and enhancing CSUs research, education, and outreach activities pertaining to the design and innovative application of computer, communication, and information systems. From 1976 to 2001, he was a professor at Purdue University. Prof. Siegel is a Fellow of the IEEE and a Fellow of the ACM. He received two B.S. degrees from the Massachusetts Institute of Technology (MIT), and the M.A., M.S.E., and Ph.D. degrees from Princeton University. He has co-authored over 340 technical papers. His research interests include heterogeneous parallel and distributed computing, parallel algorithms, and parallel machine interconnection networks. He was a Co-editor-in-Chief of the Journal of Parallel and Distributed Computing, and was on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers. He has been an international keynote speaker and tutorial lecturer, and has consulted for industry and government. For more information, please see www.engr.colostate.edu/~hj.

**Anthony A. Maciejewski** received the BSEE, MS, and PhD degrees from Ohio State University in 1982, 1984, and 1987. From 1988 to 2001, he was a professor of Electrical and Computer Engineering at Purdue University, West Lafayette. He is currently the Department Head of Electrical and Computer Engineering at Colorado State University. He is a Fellow of the IEEE. A complete vita is available at: www.engr.colostate.edu/~aam.

**Ye Hong** is pursuing his Ph.D. degree in Electrical and Computer Engineering at Colorado State University. He received his Master degree in Computer Science from Tsinghua University in 2006, and his Bachelor degree from Tsinghua University in 2002. His current research interests include parallel and distributed computing, heterogeneous computing, robust computer systems, and performance evaluation.