DISSERTATION

RESOURCE ALLOCATION FOR HETEROGENEOUS COMPUTING SYSTEMS: PERFORMANCE CRITERIA, ROBUSTNESS MEASURES, OPTIMIZATION HEURISTICS, AND PROPERTIES

Submitted by

Luis Diego Briceño Guerrero

The Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2010

Copyright by Luis Diego Briceño Guerrero 2010

All Rights Reserved

COLORADO STATE UNIVERSITY

June 27, 2010

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY LUIS DIEGO BRICEÑO GUERRERO ENTITLED RESOURCE ALLOCATION FOR HETEROGENEOUS COMPUTING SYSTEMS: PERFORMANCE CRITERIA, ROBUSTNESS MEASURES, OPTIMIZATION HEURISTICS, AND PROPERTIES BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Anton Willem Böhm

Anura Jayasumana

James T. Smith II

Advisor: Howard Jay Siegel

Co-Advisor: Anthony A. Maciejewski

Department Head: Anthony A. Maciejewski

ABSTRACT OF DISSERTATION

RESOURCE ALLOCATION FOR HETEROGENEOUS COMPUTING SYSTEMS: PERFORMANCE CRITERIA, ROBUSTNESS MEASURES, OPTIMIZATION HEURISTICS, AND PROPERTIES

Heterogeneous computing (HC) is the coordinated use of different types of machines, networks, and interfaces to maximize the combined performance and/or cost effectiveness of the system. The application environments studied in this research are: a weather data-processing system, a massive multi-player on-line gaming system, and a distributed satellite image processing system. Each one of these application environments was simulated on different computation platforms. Contributions for each environment: (1) mathematical model of environment, (2) defined a performance criterion, (3) defined robustness metric, (4) designed resource allocation heuristics based on performance and robustness measures, and (5) conducted simulation studies for evaluating and comparing heuristic techniques.

We consider an iterative approach that decreases the finishing time of machines by repeatedly executing a resource allocation heuristic to minimize the makespan of the considered machines and tasks. For each successive iteration, the makespan machine of the previous iteration and the tasks assigned to it are removed from the set of considered machines and tasks. The contribution include identifying which characteristics heuristics need to generate improvement with the iterative approach, showing that the effectiveness of the iterative approach is heuristic dependent, and deriving a theorem to identify which heuristics cannot attain improvements.

Luis Diego Briceño Guerrero
The Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, CO 80523
Summer 2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

To my parents and sister for their encouragement, and sacrifices made during this period of study.

To my friends and family for all your support.

To Jaime for your patience and understanding.

ACKNOWLEDGEMENTS

I would like to thank my advisors Prof. Siegel and Prof. Maciejewski for their guidance and help through this process. Special thanks to Prof. Smith, Prof. Jayasumana, and Prof. Bohm. Thanks to Vladimir Shestak, Samee Khan, Ricky Kwok, Jerry Potter, Paul Maxwell, Abdulla Al-Qawasmeh, and Dalton Young for their valuable comments on various portions of this work. This research was supported by the NSF under Grant CNS-0615170, CNS-0905399, and by the Colorado State University George T. Abell Endowment.

CHAPTER I

INTRODUCTION

Heterogeneous computing (HC) is the coordinated use of different types of machines, networks, and interfaces to maximize the combined performance,cost effectiveness, and/or the robustness of the system. This thesis considers using the FePIA procedure derived in [5] to mathematically model robustness metrics for several different HC environments. Resource allocation heuristics are designed for each of these environments to improve the system performance. This thesis also studies the performance of several greedy heuristics using an iterative approach.

The second chapter of this thesis considers an HC environment like the one described in [17]. In some environments, it is useful to minimize the finishing times of each machine in the system not just the machine that is the last to finish, i.e., the makespan machine. For example, if the execution times of the tasks on the makespan machine are overestimated then the minimization of the completion of other machines becomes important. In Chapter 2, we consider an iterative approach that decreases the finishing time of machines by repeatedly executing a resource allocation heuristic to minimize the makespan of the considered machines and tasks. For each successive iteration, the makespan machine of the previous iteration and the tasks assigned to it are removed from the set of considered machines and tasks. The goal is to identify which characteristics heuristics need to generate improvement with the iterative approach. We show that the effectiveness of the iterative approach is heuristic dependent, and derive a theorem to identify which heuristics cannot attain improvements.

We observe that no heuristic can guarantee an improvement; however, we can find heuristics that can change the resource allocation from one iteration to the next.

The third chapter of this thesis considers the data processing portion of a space-based weather monitoring system. It is a heterogeneous weather data processing system that is susceptible to uncertainties in the arrival time of data sets. The resource allocation must be robust with respect to these uncertainties. The tasks to be executed by the data processing system are classified into two broad categories: critical (e.g., telemetry, tracking and control), and revenue producing (e.g., data processing and data research). In this environment, the resource allocation of the critical tasks must be done before the resource allocation of the non-critical (revenue) tasks. A two-phase allocation scheme is presented in this research. The goal of the first part is to find a resource allocation that minimizes makespan of the critical tasks. For the second part, the robustness of the mapping is the difference between the expected arrival time and the time at which the revenue earned is equal to the operating cost. Thus, the heuristics for the second part find a mapping that minimizes the time for the revenue (gained by completing non-critical tasks) to be equal to the cost. We design and evaluate different resource allocation heuristics and compare their performance to a mathematical bound.

The fourth chapter considers a massive multiplayer online gaming (MMOG) environment. In this environment, each user controls an avatar (an image that represents and is manipulated by a user) in a virtual world and interacts with other users. An important aspect of MMOG is maintaining a fair environment among users (i.e., not give an unfair advantage to users with faster connections or more powerful computers). The experience (either positive or negative) the user has with the MMOG environment is dependent on how quickly the game world responds to the user's actions and the fairness of the environment. This study focuses on scaling the system based on demand. If the performance falls below acceptable standards, the main server can off-load calculations to secondary servers. A secondary server is a user's computer that is converted into a server. We consider two optimization criteria: first to minimize the maximum time it takes to get a response from the main server, and the second is to maximize the number of players that can join an on-going game, while maintaining a fair environment.

The fifth chapter considers a heterogeneous cluster of multicore-based machines used to analyze satellite images. The workload involves large data sets, and is typically subject to deadline constraints. Multiple applications, each represented by a directed acyclic graph

(DAG), are allocated to this multicore system. Each node in the DAG represents a task that needs to be executed and task execution times vary substantially across machines. The goal of this research is to assign applications to multicore-based parallel system in such a way that they all complete before a common deadline, and their completion times are robust against uncertainties in task execution times. We define a metric to quantify this robustness. In this study, we design, compare, and evaluate resource allocation heuristics that attempt to maximize this robustness metric.

CHAPTER II

ITERATIVE MINIMIZATION OF NON-MAKESPAN MACHINES

2.1 Introduction

The use of heuristics for resource allocation in a heterogeneous parallel and distributed computing environment is an important area of research and has been widely studied (e.g., [11, 17, 75, 86]). Static and dynamic mapping of tasks to machines are both used to do resource allocation [2]. Static heuristics can be used in production environments, where the tasks to be executed are known in advance; dynamic heuristics are used to allocate resources on-line, without prior knowledge of when or which tasks arrive. In this study, we will consider static task mapping. One metric for evaluating the performance of heuristics is the time to complete a set of tasks on a heterogeneous suite of machines, i.e., makespan. The makespan machine is defined as the machine with the largest completion time.

An issue one needs to consider when using makespan as an optimization criterion is what does it mean to optimize the performance of "non-makespan machines" (the non-makespan machines are all the machines in the heterogeneous computing (HC) system except the makespan machine). The procedure we consider here is to repeatedly minimize the makespan among the non-makespan machines. To achieve this goal we use the following iterative approach. This approach decreases the finishing time of non-makespan machines by repeatedly running a mapping heuristic to minimize the makespan of the remaining considered machines and tasks. For each successive iteration, the makespan machine of the previous iteration and the tasks assigned to it are removed from the set of considered machines and tasks. The scenario in [68] is an example of a situation where the minimization of the completion time of non-makespan machines can be important. Because allocation

A preliminary version of portions of this research appeared in [19].

decisions are based on predicted values of tasks execution times, the predicted makespan machine may finish earlier than expected and another machine may have a larger completion time and thus become the actual (versus predicted) makespan machine. The iterative approach studied here is just one of many methods that can be used for considering the completion time of non-makespan machines, e.g., sum of each machine's completion time squared.

This paper has three main contributions. The first contribution is the introduction of an iterative technique that may be able to reduce the completion times of non-makespan machines when used with certain heuristics. The second contribution is the definition and mathematical characterization of "non-improving" heuristics. The third contribution is to determine the characteristics of a heuristic, that will cause the mapping to change across iterations. The heuristics considered for this study were Minimum Execution Time, Minimum Completion Time, Min-Min Genetic Algorithm, Switching Algorithm, Sufferage, and K-Percent Best.

Makespan is often the performance feature to be optimized in the study of resource allocation in a heterogeneous computing system. Many studies explore different methods of reducing the makespan of the given set of tasks. The literature was examined to select a set of heuristics appropriate for the HC environment considered in this study. The MET, MCT [1,17,39,64] , and Min-Min [13,16,31,39,42,44,47,52,64,86] heuristics implemented here are adapted from [42]. The K-percent Best [1,39,65] and Switching Algorithm [39,52,64,65] were adapted from [65] and the Sufferage Algorithm [13,14,23,31,38,44,47,52,65,69,72,84] was adapted from [65]. The variation of the GA implemented here is an adaptation of the GA in [71].

In this study, we use a heuristic and iteratively apply it to a set of tasks and machines that becomes smaller after each iteration. This approach has not been studied in the past. Note that the iterative approach is different from simply running a heuristic multiple times on the same set of machines and tasks (e.g., multiple runs of a GA). In this study, the iterative approach is presented as a possible complement to any heuristic for resource allocation in an HC system.

The remainder of the paper is organized as follows. Section 2.2 describes the problem statement in detail. In Section 2.3, we describe three heuristics that when used in conjunction with the iterative approach their iterative mappings are the same as their original mapping. Some examples of heuristics where the original and iterative mappings may be different are described in Section 2.4. The observations from Sections 2.3 and 2.4 are analyzed in Section 2.5, and Section 2.6 concludes the work.

2.2 Problem Statement

Let $\underline{T}$ be the set of tasks that must be executed on a set of machines $\underline{M}$. The estimated time to compute (ETC) each task on each machine is assumed to be known in advance and contained in an ETC matrix [17]. The ETC values can be based on user supplied information, experimental data, or task profiling and analytical benchmarking [2,36,37,50, 62,89]. Determination of ETC values is a separate research problem; the assumption of such ETC information is a common practice in resource allocation research (e.g., [9,30,37,46,50, 58,78,88]). The initial ready time for a machine is the time at which the machine will become available to begin processing its first task from T. Tasks are assumed to be independent, i.e., no inter-task communication is required. We make the common simplifying assumption that each machine can only execute one task at a time, i.e., multitasking is not allowed (e.g., [32,56]).

For each heuristic, the mapping it produces when all tasks and machines are available is called the original mapping. After each iteration (of the iterative approach), the makespan machine and the tasks assigned to it are removed from consideration, and the ready times for all remaining machines are reset to their initial ready times. The tasks that are available for mapping (were not mapped to the makespan machine in the original mapping) are mapped again, using the same heuristic to minimize makespan among the remaining machines; this mapping is called the iterative mapping. In the context of this paper, mappable tasks are those tasks that a heuristic can select from to assign at a given point during the allocation. The available machines are the machines that can be assigned a mappable task. It is important to note that both mappable tasks and available machines are dependent on the heuristic used in the iterative approach. This iterative process is repeated until only one machine remains. The goal of this study is to identify which characteristics heuristics need to generate improvement with the iterative approach. Improvement is when the makespan among the remaining machines compared to the makespan among the non-makespan machines generated in the original mapping is reduced. We show that the effectiveness of the iterative approach is heuristic dependent and study the behavior of the iterative approach for each of the chosen heuristics.

Whether the iterative approach will change a mapping often depends on how ties are broken within a heuristic. A tie in a resource allocation heuristic is when a heuristic must choose from two equally good solutions, i.e., the heuristic determines both task assignments are the best possible task assignments. Two types of methods to break ties will be considered for this study. The first method is to break ties deterministically, e.g., the task and machine with the lowest identification number are chosen. The second method is to break ties randomly, e.g., if multiple machines are tied each will have a an equal probability of being chosen.

The problem we wish to address has two main components: (1) for a given heuristic considered here, will the iterative approach result in a change to the original mapping, (2) what are the characteristics of a heuristic whose original mapping may be improved by this iterative technique.

Makespan is often the performance feature to be optimized in the study of resource allocation in a heterogeneous computing system. Many studies explore different methods of reducing the makespan of the given set of tasks. The literature was examined to select a set of heuristics appropriate for the HC environment considered in this study. The MET, MCT [1, 17, 39] , and Min-Min [16, 31, 39, 42, 44, 47, 52, 86] heuristics implemented here are adapted from [42]. The K-percent Best [1, 39, 65] and Switching Algorithm [39, 52, 65] were adapted from [65] and the Sufferage Algorithm [14, 23, 31, 38, 44, 47, 52, 65, 69, 72, 84] was adapted from [65]. The variation of the GA implemented here is an adaptation of the GA in [71].

In this study, we use a heuristic and iteratively apply it to a set of tasks and machines

1 A task list is generated that includes all unmapped tasks in a given arbitrary order.

2 The first task in the list is assigned to its minimum execution time machine.

3 The task selected in step 2 is removed from the task list.

4 Steps 2-3 are repeated until all tasks have been mapped.

Figure 1: Procedure for using MET to generate a resource allocation.

that becomes smaller after each iteration. This approach has not been studied in the past. Note that the iterative approach is different from simply running a heuristic multiple times on the same set of machines and tasks (e.g., multiple runs of a GA). In this study, the iterative approach is presented as a possible complement to any heuristic for resource allocation in an HC system.

In the following section, we describe three common heuristic techniques where, if ties are broken deterministically, the iterative approach will not change the mapping. We will also formalize the properties that allow this to occur. In Section 2.4, we present four heuristics from the literature where the iterative approach may change the mapping for better or for worse.

2.3 Heuristics that Will Not Improve With the Iterative Approach

2.3.1 Minimum Execution Time (MET) with Deterministic Tie Breaking

The details of the Minimum Execution Time (MET) heuristic [1, 17, 39] are shown in Figure 24. The MET heuristic will not change its mapping from iteration to iteration. The generalized proof of why this occurs will be presented in Section 2.3.5.

2.3.2 Minimum Completion Time with Deterministic Tie Breaking (MCT)

The procedure to implement the Minimum Completion Time (MCT) heuristic [1,17,39] is shown in Figure 25. With the iterative approach, the individual completion time for each machine does not improve over iterative mappings if ties are broken deterministically (will be demonstrated in Section 2.3.5).

1 A task list is generated that includes all unmapped tasks in a given arbitrary order.

2 The first task in the list is assigned to its minimum completion time machine (machine ready time plus estimated computation time of the task on that machine).

3 The task selected in step 2 is removed from the task list.

4 The ready time of the machine on which the task is assigned is updated.

5 Steps 2-4 are repeated until all the tasks have been mapped.

Figure 2: Procedure for using MCT to generate a resource allocation.

1 A task list is generated that includes all the tasks as unmapped tasks.

2 For each task in the task list, the machine that gives the task its minimum completion time (first Min) is determined (ignoring other unmapped tasks).

3 Among all task-machine pairs found in 2, the pair that has the minimum completion time (second Min) is determined.

4 The task selected in 3 is removed from the task list
and is assigned to the paired machine.

5 The ready time of the machine on which the task is mapped is updated.

6 Steps 2-5 are repeated until all tasks have been mapped.

Figure 3: Procedure for using Min-Min to generate a resource allocation

2.3.3 Min-Min with Deterministic Tie Breaking

The Min-Min heuristic [31, 39, 42, 44, 47, 52, 86] is a two phase greedy heuristic. The procedure for this heuristic is given in Figure 27. The performance of the Min-Min heuristic using the iterative approach will depend on the method used to break ties. If the ties are broken deterministically, the individual completion times for each machine do not improve (will be demonstrated in Section 2.3.5).

2.3.4 Generalized Completion Time Function and Iteration Invariant Heuristics

The MET, MCT, and Min-Min heuristics are minimizing very similar performance features. We will relate the different performance functions by a generalized one based on machine ready time and task execution time. Let $RT_{k,n}(m)$ be the ready time of machine m at the n^{th} mapping event (assignment of a task to a machine) of the k^{th} iteration, and $ETC(t,m)$ be the estimated time to compute task t on machine m. A generalized completion time (GCT) function of task t on machine m (where λ and η are arbitrary values) is:

$$GCT(t,m,n,k) = \lambda \cdot ETC(t,m) + \eta \cdot RT_{k,n}(m). \tag{1}$$

We can then define the completion time, CT, of a new task t on machine m with Equation 1 and $\lambda = \eta = 1$. For both MCT and Min-Min the values of λ, η are equal to 1, the difference between the set of machines considered for the MCT and Min-Min heuristics. For the MET, the value of λ is equal to 1, and η is equal to 0. For GCT functions (Equation 1), the ETC and *initial* ready times ($RT_{k,1}(m)$) do not vary across iterations (i.e., $RT_{1,1}(m) = RT_{2,1}(m) = ... = RT_{M,1}(m)$).

We define an Iteration Invariant Heuristic (IIH) as a heuristic whose mapping will not change across all iterations. Let $T_{k,n}$ be the set of mappable tasks and $M_{k,n}$ be the set of available machines at the n^{th} mapping event in the k^{th} iteration. We will show in the next section that a specific type of IIH (which encompasses MCT, MET, and Min-Min) can be defined that at every mapping event n a task (t_{min}) is assigned to machine m_{min} (with an argmin* approach), where

$$t_{min}, m_{min} \quad = \underset{t \in T_{k,n}, m \in M_{k,n}}{\text{argmin}} \; GCT(t,m,n,k) \; . \tag{2}$$

It follows that at each mapping event these heuristics assign one task to one machine. [1] Additionally, this type of IIH (GCT IIH) breaks ties deterministically, i.e., they will always pick the same pair. In particular, we select the t_{min} with the lowest task identification number, and if necessary, the lowest numbered m_{min}.

2.3.5 Properties of Iteration Invariant Heuristics

In GCT IIHs, the assignment of tasks to machines does not change across iterations, and thus these heuristics are not well suited for use with the iterative approach, that is, no improvement is made over the original mapping. To design heuristics that can cause the

[1] *argmin: returns the arguments at which a function attains its minimum value

	machines		
task	m_1	m_2	m_3
t_1	3	2	4
t_2	4	5	6
t_3	8	3	1

(a)

iteration k=1

		completion time			
mapping event (n)	task	m_1	m_2	m_3	assignment
1	t_1	3	2	4	$t_3 \rightarrow m_3$
	t_2	4	5	6	
	t_3	8	3	①	
2	t_1	3	②	5	$t_1 \rightarrow m_2$
	t_2	4	5	7	
3	t_2	④	7	7	$t_2 \rightarrow m_1$

(b)

iteration k=2

		completion time			
mapping event (n)	task	m_1	m_2	m_3	assignment
1	t_1	makespan machine of iteration k=1	2	4	$t_3 \rightarrow m_3$
	t_2		assigned in iteration k=1		
	t_3		3	①	
2	t_1		②	5	$t_1 \rightarrow m_2$

(c)

iteration k=3

		completion time			
mapping event (n)	task	m_1	m_2	m_3	assignment
1	t_1	makespan machine of iteration k=1	makespan machine of iteration k=2	assigned in iteration k=2	$t_3 \rightarrow m_3$
	t_2			assigned in iteration k=1	
	t_3			①	

(d)

Figure 4: Example of a resource allocation using the Min-Min heuristic. In (a), the ETC matrix used for this example is found. The allocation of the first iteration is shown in (b), the second iteration in (c), and the final iteration in (d). This example illustrates that iterations 2 and 3 do not change the original mapping.

mapping to change with the iterative approach, it is important to understand the properties of GCT IIHs.

Consider the ETC and the example resource allocation done by the Min-Min heuristic with three tasks (t_1, t_2, t_3) and three machines (m_1, m_2, m_3) shown in Figure 4. We can observe that for the Min-Min heuristic the relative order in which tasks are assigned to machines does not change, i.e., if a task $\Gamma_{k,i}$ (i^{th} task assigned in iteration k) was assigned before a task $\Gamma_{k,j}$ $(i < j)$ then $\Gamma_{k,i}$ is always assigned before $\Gamma_{k,j}$ in every iteration, assuming both tasks are available for mapping. This property will be proved later in this sub-section.

We cannot use n to compare the assignment of a task across the k^{th} and $(k+1)^{th}$ iterations, because the n in the $(k+1)^{th}$ iteration could represent the mapping event of different task. Assume task t is available for mapping at iterations k and $k+1$, i.e., it

n	mapping event of the k^{th} iteration
$RT_{k,n}(m)$	ready time for machine m at the n^{th} mapping event of the k^{th} iteration
$ETC(t,m)$	estimated time to compute task t on machine m
$GCT(t,m,n,k)$	$GCT(t,m,n,k) = \lambda \cdot ETC(t,m) + \eta \cdot RT_{k,n}(m)$
$T_{k,n}$	set of mappable tasks at the n^{th} mapping event of the k^{th} iteration
$M_{k,n}$	set of available machines at the n^{th} mapping event of the k^{th} iteration
T_{μ_k}	tasks assigned to the makespan machine in the k^{th} iteration
$\Gamma_{k,n}$	n^{th} task mapped at the k^{th} iteration
$\beta_{k,n}$	machine to which $\Gamma_{k,n}$ is mapped
μ_k	makespan machine of iteration k
$\omega(k,\Gamma_{k,n})$	represents the mapping event where task $\Gamma_{k,n}$ is assigned in the k^{th} iteration
$\hat{n}$	$\hat{n} = \omega(k+1,\Gamma_{k,n})$
$\Gamma_{k+1,\hat{n}}$	$\hat{n}^{th}$ task mapped at the $(k+1)^{th}$ iteration, by definition it is equal to $\Gamma_{k,n}$
$\beta_{k+1,\hat{n}}$	machine to which $\Gamma_{k+1,\hat{n}}$ is mapped (proved in Thm. 1 to be equal to $\beta_{k,n}$)

Figure 5: Glossary of Notation.

is not on the makespan machine for iterations $z \leq k$. To compare across iterations, we use the function $\underline{\omega(k,t)}$ to represent the mapping event where task t is assigned in the k^{th} iteration, i.e., by definition $\underline{n = \omega(k,\Gamma_{k,n})}$. For convenience, we will define $\underline{\hat{n}} = \omega(k+1,\Gamma_{k,n})$ to compare the mapping of $\Gamma_{k,n}$ in the k^{th} and the $(k+1)^{th}$ iteration (at the n^{th} and $\hat{n}^{th}$ mapping events respectively). Thus, $\Gamma_{k,n}$ is equal to $\Gamma_{k+1,\hat{n}}$ by definition.

We will show that if $\Gamma_{k,n}$ is mapped to machine $\underline{\beta_{k,n}}$ in the k^{th} iteration, i.e.,

$$\Gamma_{k,n}, \beta_{k,n} = \operatorname*{argmin}_{t \in T_{k,n}, m \in M_{k,n}} GCT(t,m,n,k), \tag{3}$$

and $\Gamma_{k+1,\hat{n}}$ (recall that $\Gamma_{k,n} = \Gamma_{k+1,\hat{n}}$) is mapped to machine $\beta_{k+1,\hat{n}}$ in the $(k+1)^{th}$ iteration, i.e.,

$$\Gamma_{k,n}, \beta_{k+1,\hat{n}} = \operatorname*{argmin}_{t \in T_{k+1,\hat{n}}, m \in M_{k+1,\hat{n}}} GCT(t,m,\hat{n},k+1), \tag{4}$$

then $\beta_{k+1,\hat{n}} = \beta_{k,n}$. The relationship of $T_{k,n}$ to $T_{k+1,\hat{n}}$, and $M_{k,n}$ to $M_{k+1,\hat{n}}$ is an important aspect of GCT IIHs. Let us define the makespan machine of the k^{th} iteration as $\underline{\mu_k}$, and the set of tasks assigned to the makespan machine in the k^{th} iteration as $\underline{T_{\mu_k}}$. Between the sets of tasks $T_{k,n}$ and $T_{k+1,\hat{n}}$ two conditions are needed for the mapping to be iteration invariant:

$$
\begin{aligned}
T_{k+1,\hat{n}} &\subseteq T_{k,n}, \text{ and} && (5)\\
T_{k,n} - T_{k+1,\hat{n}} &\subseteq \{T_{\mu_k}, \varnothing\}. && (6)
\end{aligned}
$$

Two additional conditions for the relationship between $M_{k,n}$ and $M_{k+1,n_{k+1}}$ are:

$$
\begin{aligned}
M_{k+1,\hat{n}} &\subset M_{k,n}, \text{ and} && (7)\\
M_{k,n} - M_{k+1,\hat{n}} &= \{\mu_k\}. && (8)
\end{aligned}
$$

A GCT IIH is a heuristic that has the properties shown in Equations 1, 5, 6, 7, 8, and breaks ties deterministically. The proof that a mapping generated by GCT IIHs will not change across iterations is shown in Theorem 1 using Lemma 1. Lemma 1 states that if ready times are identical at the n^{th} mapping event of the k^{th} and the $\hat{n}^{th}$ mapping event of the $(k+1)^{th}$ iteration, then the task/machine assignment will be the same. Theorem 1 uses the result from Lemma 1 to show that the mapping will not change across iterations. A glossary of relevant notation is shown in Figure 5.

Lemma 1. *Consider tasks not assigned to μ_k for a GCT IIH. If the ready times are identical at the n^{th} mapping event of the k^{th} iteration and the $\hat{n}^{th}$ mapping event of the $(k+1)^{th}$ iteration then the same task/machine assignment is chosen in both iterations.*

Proof. From Equation 2, we know that the GCT function of the task/machine pairs in the k^{th} iteration will have the following property:

$$GCT(\Gamma_{k,n}, \beta_n, \hat{n}, k) \leq GCT(t, m, n, k) \ \forall t \in T_{k,n}, \forall m \in M_{k,n}, \tag{9}$$

and the $(k+1)^{th}$ iteration has the following property:

$$GCT(\Gamma_{k+1,\hat{n}}, \beta_{k+1,\hat{n}}, \hat{n}, k+1) \leq GCT(t, m, \hat{n}, k+1) \ \forall t \in T_{k+1,\hat{n}}, \forall m \in M_{k+1,\hat{n}}. \tag{10}$$

Given the assumption that $RT_{k+1,\hat{n}}(m) = RT_{k,n}(m)$, and the ETC values are fixed it follows that:

$$GCT(t, m, \hat{n}, k+1) = GCT(t, m, n, k) \forall t \in T_{k+1,\hat{n}}, \forall m \in M_{k+1,\hat{n}}. \tag{11}$$

Because $\{\Gamma_{k+1,\hat{n}}, \beta_{k+1,\hat{n}}\}$ and $\{\Gamma_{k,n}, \beta_{k,n}\}$ are the task-machine pairs that minimize the GCT function in the $(k+1)^{th}$ and k^{th} iterations, and ties are broken deterministically then $\beta_{k+1,\hat{n}} = \beta_{k,n}$, i.e., respectively

$$\underset{t \in T_{k+1,\hat{n}}, m \in M_{k+1,\hat{n}}}{\operatorname{argmin}} GCT(t, m, \hat{n}, k+1) = \underset{t \in T_{k,n}, m \in M_{k,n}}{\operatorname{argmin}} GCT(t, m, n, k), \tag{12}$$

$$\Gamma_{k+1,\hat{n}}, \beta_{k+1,\hat{n}} = \Gamma_{k,n}, \beta_{k,n}. \tag{13}$$

□

Theorem 1. *The mapping generated by a GCT IIH will not change across iterations.*

Proof. **Inductive hypothesis**: Consider the n^{th} task mapped by a GCT IIH in the k^{th} iteration ($\Gamma_{k,n}$). Let $\underline{P(n)}$ be the statement that $\Gamma_{k,n}$ will have the same assignment in both the k^{th} iteration and the $(k+1)^{th}$ iteration. For the basis and inductive steps, there are two cases to consider: the case when the task is mapped to the makespan machine in the k^{th} iteration ($\underline{\mu_k}$) and the case when the task is not mapped to machine μ_k.

To prove $P(n)$ is true $\forall n \geq 1$ we need to prove:

1) $P(1)$ is true

2) $(\forall n)$ [$P(n)$ is true for all r, $1 \leq r \leq n \Rightarrow P(n+1)$ is true].

Basis Step: Prove that $P(1)$ is true.

Case 1: The task $\Gamma_{k,1}$ was assigned to machine μ_k. Because $\Gamma_{k,1}$ was assigned to μ_k, it will remain assigned to μ_k in the $(k+1)^{th}$ iteration because the makespan machine is removed from consideration; thus, the $P(1)$ statement is true for case 1 of the basis step.

Case 2: The *initial* ready times of the k^{th} and the $(k+1)^{th}$ iterations are identical. Therefore, using Lemma 1, we can prove case 2 is true. Thus the $P(1)$ statement is true for case 2 of the basis step.

Inductive Step: For the inductive step assume that $\forall r$ $P(1 \leq r \leq n)$ is true and prove $P(n+1)$ is true. The assumption that $P(1 \leq r \leq n)$ is true implies that ready times, when $\Gamma_{n+1,k}$ is mapped, are equal in the k^{th} mapping and the $(k+1)^{th}$ mapping.

Case 1: The $(n+1)^{th}$ task to be mapped in the k^{th} iteration was assigned to machine μ_k.

Because $\Gamma_{n+1,k}$ was assigned to μ_k, it will remain assigned to μ_k in the $(k+1)^{th}$ iteration because the makespan machine is removed from consideration; thus, the $P(1)$ statement is true for case 1 of the inductive step.

Case 2: Because of the inductive assumption, all the previous task to machine assignments are identical between the k^{th} and $(k+1)^{th}$ iteration. This implies that the ready times, for all machines in both $M_{k,n}$ and $M_{k+1,\hat{n}}$ are the same. Therefore, using Lemma 1, we prove case 2 is true. Thus the $P(n+1)$ statement is true for case 2 of the inductive step. This proves that all assignments remain identical between iterations. □

GCT IIH like MET, MCT, and Min-Min are indeed IIH when ties are broken deterministically and the mappings do not change with the iterative approach. It is important to note that in practice, depending on precision of completion times, ties can be rare. Also if two machines are tied as the makespan machine, then the machine with the smallest identification number is selected. In the next section, we explore heuristics that may improve the iterative approach.

2.4 Heuristics That May Improve with the Iterative Approach

2.4.1 MET with Random Tie Breaking

2.4.1.1 Example of Reducing the Makespan among Non-Makespan Machines

This is the same MET as described in Section 2.3.1 except that ties are broken randomly. For this example, the initial ready times of machines are 0. Consider the following mapping order: t_1, t_2, and t_3. The resource allocations of the original mapping are shown in Figure 6(a) and 6(c). There are two MET machines for t_1 and t_3. In the original mapping, task t_1 and t_3 will be assigned to m_2.

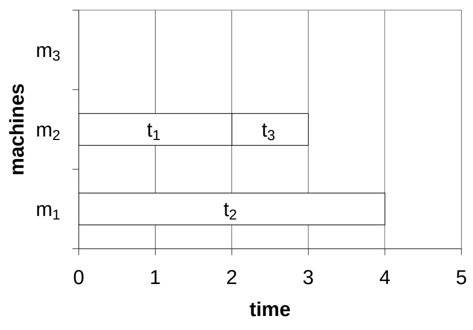
The first iterative mapping (without machine m_1)is shown in Figures 6(b) and 6(d). In the first iterative mapping, t_1 is assigned to m_2 and t_3 is assigned to m_3. This change causes the makespan among the machines in this iteration to decrease. The makespan of the set of machines m_2 and m_3 is decreased from 3 in the original mapping to 2 in the first iterative mapping.

assignment	machine ETC (m_1, m_2, m_3)
$t_1 \rightarrow m_2$	3, $\underline{2}$, $\underline{2}$
$t_2 \rightarrow m_1$	$\underline{4}$, 5, 6
$t_3 \rightarrow m_2$	8, $\underline{1}$, $\underline{1}$

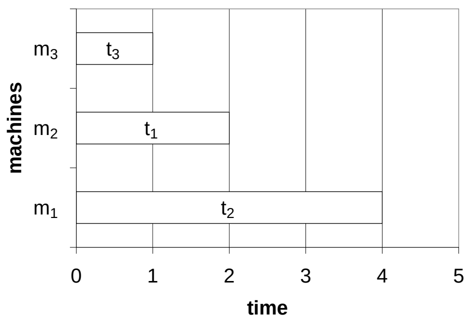
(a)

assignment	machines ETC (m_2, m_3)
$t_1 \rightarrow m_2$	$\underline{2}$, $\underline{2}$
$t_3 \rightarrow m_3$	$\underline{1}$, $\underline{1}$

(b)



(c)



(d)

Figure 6: Example of makespan among non-makespan machines being reduced for MET: (a) details of original mapping, (b) details of first iterative mapping, (c) graphical representation of original mapping, and (d) graphical representation of first iterative mapping.

2.4.1.2 Example of <u>Increasing</u> Overall Makespan

The previous subsection shows we can reduce, from iteration k to $k+1$, the maximum finishing times among the machines available at the $(k+1)^{th}$ iteration. However, it is also possible that the maximum finishing time can be increased even beyond the original makespan. Because of this, we must be careful when applying this method. An example of makespan increasing is shown in this subsection.

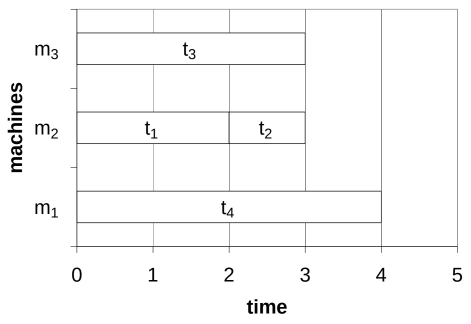
For this example, the initial ready times of machines are 0. Consider the following mapping order: t_1, t_2, t_3, and t_4. The resource allocations of the original mapping are shown in Figures 7(a) and 7(c). There are two MET machines for t_1. Task t_1 will be assigned to m_2 in the original mapping. However, in the first iterative mapping (without machine m_1) shown in Figures 7(b) and 7(d), t_1 is assigned to m_3. This change causes the makespan to become greater than that of the original full mapping. In practice, you can try all possible combination of tie breaking. For this example, we break ties randomly.

assignment	machine ETC (m_1, m_2, m_3)
$t_1 \rightarrow m_2$	3, $\underline{2}$, $\underline{2}$
$t_2 \rightarrow m_2$	4, $\underline{1}$, 4
$t_3 \rightarrow m_3$	5, 4, $\underline{3}$
$t_4 \rightarrow m_1$	$\underline{4}$, 5, 6

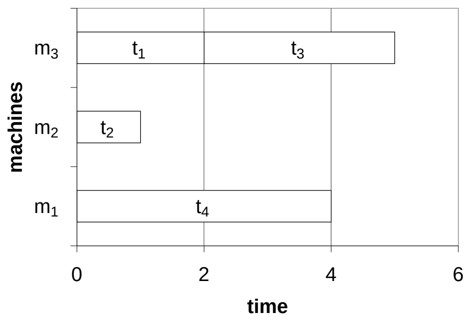
(a)

assignment	machines ETC (m_2, m_3)
$t_1 \rightarrow m_3$	$\underline{2}$, $\underline{2}$
$t_2 \rightarrow m_2$	$\underline{1}$, 4
$t_3 \rightarrow m_3$	4, $\underline{3}$

(b)



(c)



(d)

Figure 7: Example of overall makespan increasing for MET: (a) details of original mapping, (b) details of first iterative mapping, (c) graphical representation of original mapping, and (d) graphical representation of first iterative mapping.

2.4.2 MCT with Random Tie Breaking

2.4.2.1 Example of <u>Reducing</u> the Makespan among Non-Makespan Machines

This is the same MCT as described in Section 2.3.2 except that ties are broken randomly. For this example of improvement, the initial ready times are 0. Consider the following mapping order for the MCT heuristic: t_1, t_2, t_3, and t_4. The ETC matrix used for this example is shown in Figure 8(a). In Figures 8(c) and 8(b), the term "Machine CT" denotes the Machine completion time of the task in the corresponding row, based on previous task assignments. This example relies on a tie in the mapping of task t_3 between m_2 and m_3. In the original mapping, t_3 is assigned to m_2.

In the original mapping shown in Figures 8(b) and Figure 8(d), t_3 is assigned to machine m_3. However, in the first iterative mapping shown in Figures 8(c) and 8(e), t_3 is assigned to machine m_2. Thus, the makespan of the remaining machines m_2 and m_3 went from 5 in the original mapping to 4 in the first iterative mapping.

task	machine (m_1, m_2, m_3)
t_1	8, 9, 10
t_2	3, 2, 5
t_3	5, 2, 4
t_4	3, 3, 1

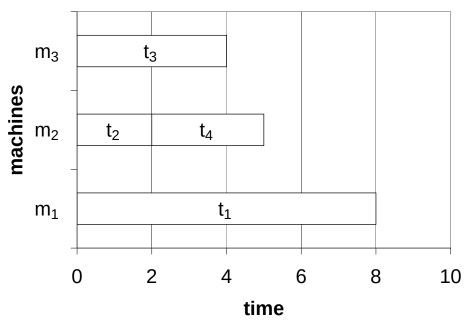
(a)

assignment	machine CT (m_1, m_2, m_3)
$t_1 \rightarrow m_1$	$\underline{8}$, 9, 10
$t_2 \rightarrow m_2$	11, $\underline{2}$, 5
$t_3 \rightarrow m_3$	13, $\underline{4}$, $\underline{4}$
$t_4 \rightarrow m_2$	11, $\underline{5}$, $\underline{5}$

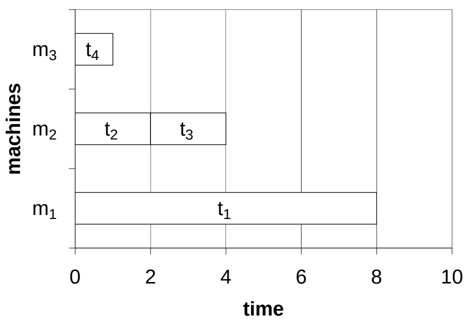
(b)

assignment	machine CT (m_2, m_3)
$t_2 \rightarrow m_2$	$\underline{2}$, 5
$t_3 \rightarrow m_2$	$\underline{4}$, $\underline{4}$
$t_4 \rightarrow m_3$	7, $\underline{1}$

(c)



(d)



(e)

Figure 8: Example of makespan among non-makespan machines being reduced with MCT: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

2.4.2.2 Example of <u>Increasing</u> Overall Makespan

For this example, the initial ready times of machines are 0. Consider the following mapping order for the MCT heuristic: t_1, t_2, t_3, and t_4. The ETC matrix used for this example is shown in Figure 9(a).

The original mapping is shown in Figures 9(b) and 9(d). This example relies on a tie in the mapping of task t_1 between m_2 and m_3. In the original mapping, t_1 is assigned to m_2. The resource allocations for the first iterative mapping are shown in Figures 9(c) and 9(e). For the first iterative mapping, we assign task t_1 to m_3. This change causes the makespan to become greater than that of the original full mapping.

2.4.3 Min-Min with Random Tie Breaking

2.4.3.1 Example of <u>Reducing</u> the Makespan among Non-Makespan Machines

This is the same Min-Min as described in Section 2.3.3 except that ties are broken randomly. For this example of improvement, the initial ready times of machines are 0. The ETC matrix used is shown in Figure 10(a). In Figures 10(b) and 10(c), the term "Machine

task	machines (m_1, m_2, m_3)
t_1	3, 2, 2
t_2	4, 1, 4
t_3	5, 4, 3
t_4	4, 5, 4

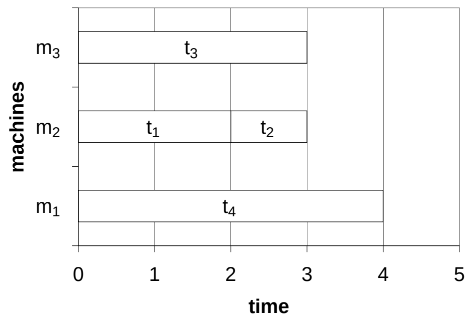
(a)

assignment	machine CT (m_1, m_2, m_3)
$t_1 \rightarrow m_2$	3, $\underline{2}$, $\underline{2}$
$t_2 \rightarrow m_2$	4, $\underline{3}$, 4
$t_3 \rightarrow m_3$	5, 7, $\underline{3}$
$t_4 \rightarrow m_1$	$\underline{4}$, 8, 7

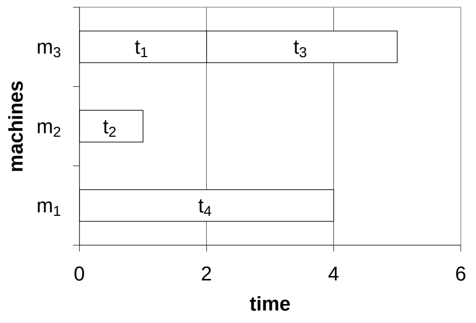
(b)

assignment	machine CT (m_2, m_3)
$t_1 \rightarrow m_3$	$\underline{2}$, $\underline{2}$
$t_2 \rightarrow m_2$	$\underline{1}$, 6
$t_3 \rightarrow m_3$	$\underline{5}$, $\underline{5}$

(c)



(d)



(e)

Figure 9: Example of overall makespan increasing for MCT: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

CT" denotes the Machine completion time of the task in the corresponding row, based on previous task assignments.

The original mapping is shown in Figures 10(b) and 10(d), and the iterative mapping is shown in Figures 10(c) and 10(e). In the second assignment of the example shown in Figures 10(b) and 10(c), task t_2 was assigned to m_3 in the original mapping, and then to m_2 in the first iterative mapping (when makespan machine m_1 is excluded. The makespan of the set of machines m_2 and m_3 is decreased from 6 in the original mapping to 4 in the first iterative mapping.

2.4.3.2 Example of Increasing Overall Makespan

For this example of makespan increasing, the initial ready times of machines are 0. The ETC matrix used is shown in Figure 11(a). The original mapping is shown in Figures 11(b) and 11(d), and the first iterative mapping is shown in Figures 11(c) and 11(e). In the original mapping, we considered that this tie was broken by assigning t_3 to m_2; however, in the first iterative mapping t_2 is assigned to m_3. This change causes the makespan to become greater than that of the original full mapping. This example proves that the makespan can

task	machines (m_1, m_2, m_3)
t_1	8, 9, 10
t_2	10, 3, 4
t_3	11, 1, 4
t_4	8, 5, 4

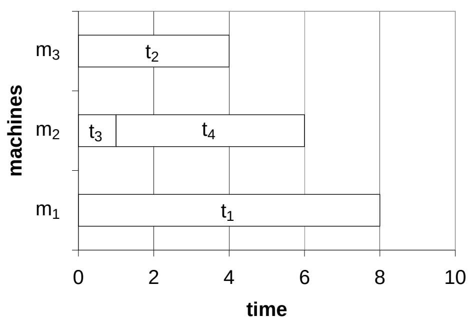
(a)

1st assignment	machine CT (m_1, m_2, m_3)
t_1	8, 9, 10
t_2	10, 3, 4
$t_3 \rightarrow m_2$	11, $\underline{1}$, 4
t_4	8, 5, 4
2nd assignment	(m_1, m_2, m_3)
t_1	8, 10, 10
$t_2 \rightarrow m_3$	10, $\underline{4}$, $\underline{4}$
t_4	8, 6, $\underline{4}$
3rd assignment	(m_1, m_2, m_3)
t_1	8, 10, 14
$t_4 \rightarrow m_2$	8, $\underline{6}$, 8
4th assignment	(m_1, m_2, m_3)
$t_1 \rightarrow m_1$	$\underline{8}$, 15, 14

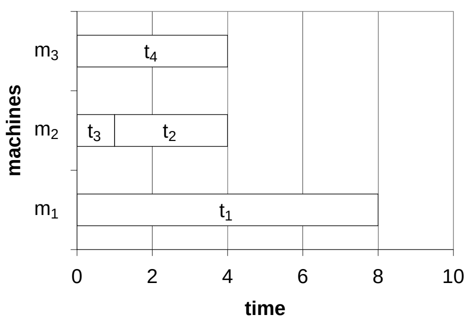
(b)

1st assignment	machine CT (m_2, m_3)
t_2	3, 4
$t_3 \rightarrow m_2$	$\underline{1}$, 4
t_4	5, 4
2nd assignment	(m_2, m_3)
$t_2 \rightarrow m_2$	$\underline{4}$, $\underline{4}$
t_4	6, $\underline{4}$
3rd assignment	(m_2, m_3)
$t_4 \rightarrow m_3$	9, $\underline{4}$

(c)



(d)



(e)

Figure 10: Example of makespan among non-makespan machines being reduced for Min-Min: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

increase if ties are broken randomly using the Min-Min heuristic.

2.4.4 Genetic Algorithm (GA)

Genetic Algorithms (GAs) have been shown to work well for numerous problem domains, including resource allocation and job shop scheduling. GAs use chromosomes to represent possible solutions, e.g., all tasks and the machines to which they are assigned. A GA

task	machines (m_1, m_2, m_3)
t_1	4, 1, 3
t_2	5, 6, 7
t_3	3, 1, 2
t_4	5, 5, 4

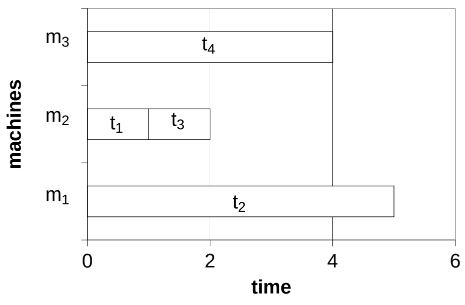
(a)

1st assignment	machine CT (m_1, m_2, m_3)
$t_1 \rightarrow m_2$	$4, \underline{1}, 3$
t_2	5, 6, 7
t_3	$3, \underline{1}, 2$
t_4	5, 5, 4
2nd assignment	(m_1, m_2, m_3)
t_2	5, 7, 7
$t_3 \rightarrow m_2$	$3, \underline{2}, \underline{2}$
t_4	5, 6, 4
3rd assignment	(m_1, m_2, m_3)
t_2	5, 8, 7
$t_4 \rightarrow m_3$	$5, 7, \underline{4}$
4th assignment	(m_1, m_2, m_3)
$t_2 \rightarrow m_1$	$\underline{5}, 8, 11$

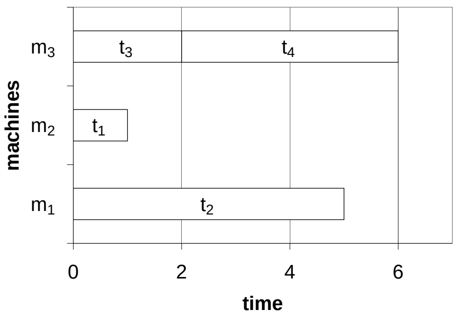
(b)

1st assignment	machine CT (m_2, m_3)
$t_1 \rightarrow m_2$	$\underline{1}, 3$
t_3	$\underline{1}, 2$
t_4	5, 4
2nd assignment	(m_2, m_3)
$t_3 \rightarrow m_3$	$\underline{2}, \underline{2}$
t_4	6, 4
3rd assignment	m_2, m_3
$t_4 \rightarrow m_3$	$\underline{6}, \underline{6}$

(c)



(d)



(e)

Figure 11: Example of overall makespan increasing for Min-Min: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

has a population that consists of multiple chromosomes and typically has two operators to search for better solutions. The first operator is crossover, an operator that combines two chromosomes to produce two new chromosomes. The second operator is mutation, this operator may randomly change tasks assignments within a chromosome. A GA can be summarized by the procedure shown in Figure 29. This variation of a GA is adapted from [71].

```
1 An initial population of mappings is generated.

2 The mappings in the population are ordered based on makespan.

3 While the stopping criteria is not met:

    a An intermediate population is created using a selection mechanism
      (i.e., rank based selection).
    b Two chromosomes in the intermediate population are probabilistically
      selected as parents for crossover.
        i A random cut-off point is generated.
        ii The machine assignments of the tasks below the cut-off point are ex-
           changed.
    c Each offspring has a probability of being mutated. For the chosen
      chromosome, a random task is chosen and its machine assignment
      is arbitrarily modified.
    d The resultant population of the crossover and mutation
      replaces the original population. Because of elitism
      the best chromosome remains in the population.

4 The best solution is output.
```

Figure 12: Summary of one possible procedure that can be used to implement a GA

For each iteration (of the iterative approach), the best mapping found by GA in the previous iteration, excluding the makespan machine and the tasks assigned to it, is included in current population (i.e., it is used as a "seed"). Using elitism in the GA guarantees that the final mapping is either the seeded mapping or a mapping with a smaller makespan, among the machines considered in the current iteration. Thus, for GA the iterative approach will result in either an improvement or no change. The iterative technique also works with steady state GAs such as Genitor [85], because Genitor uses ranking to keep the best chromosome.

2.4.5 K-percent Best Algorithm

2.4.5.1 Overview of K-percent Best Algorithm

The K-percent Best Algorithm [1, 39, 65]. The K-percent Best Algorithm, is a hybrid of MET and MCT. The procedure to implement K-percent Best is shown in Figure 26. If the percentage is (100/number of machines)% then the K-percent Best is identical to the

1 A task list is generated that includes all unmapped tasks in a given arbitrary order.

2 A subset is formed by picking the $M \cdot (\frac{k}{100})$ best machines based on the execution times for the task.

3 The task is assigned to a machine that provides the earliest completion time in the subset.

4 The task is removed from the unmapped task list.

5 The ready time of the machine on which the task is assigned is updated.

6 Steps 2-5 are repeated until all tasks have been assigned.

Figure 13: Procedure for using K-percent Best to generate a resource allocation.

MET heuristic, however, if the percentage is 100% then it is identical to the MCT heuristic. An example of the K-percent Best Algorithm with the iterative approach improving the makespan among non-makespan machines, and in contrast, an example of increasing the overall makespan, can be found even for cases when no special consideration is used to break ties. The percentage K for a given environment is found by experimentation.

The K-percent best heuristic uses the MCT heuristic that Theorem 1 proved would not change mappings unless ties were broken randomly, however its mappings do change. This introduces an important question: how can a heuristic that uses MCT to do its assignments change when ties are broken deterministically? The characteristic that K-percent Best has is that it limits the number of machines considered for assignment depending on the size of the set of machines that are available at that iteration. This implies that if the percentage was such that only one machine is selected, i.e., KPB would work like MET or 100% (K-percent Best would work like MCT) the mapping would not change across iterations. If the value of K is between these two extreme values then the KPB will violate Equation 8. Thus, despite the MCT part, the K-percent Best is not a GCT IIH. For values of K that result in a group of machines whose size is greater than one, the reduction in machines after an iteration can cause the minimum completion time machine in the original mapping to be left out of the machines the heuristic allows for assignment in the first iterative mapping.

task	machines (m_1, m_2, m_3)
t_1	80, 90, 100
t_2	100, 40, 30
t_3	110, 39, 10
t_4	80, 30, 40

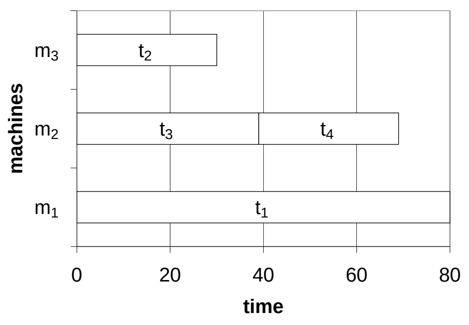
(a)

assignment	machine CT (m_1,m_2,m_3)	K-% machines
$t_1 \rightarrow m_1$	$\underline{80}$, 90, —	m_1,m_2
$t_2 \rightarrow m_3$	—, 40 , $\underline{30}$	m_2,m_3
$t_3 \rightarrow m_2$	—, $\underline{39}$, 40	m_2,m_3
$t_4 \rightarrow m_2$	—, $\underline{69}$, 70	m_2,m_3

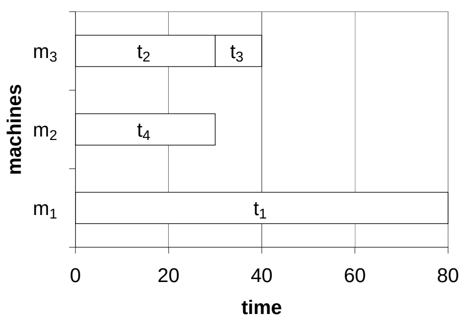
(b)

assignment	machine CT (m_2,m_3)	K-% machines
$t_2 \rightarrow m_3$	—, $\underline{30}$	m_3
$t_3 \rightarrow m_3$	—, $\underline{40}$	m_3
$t_4 \rightarrow m_2$	$\underline{30}$, —	m_2

(c)



(d)



(e)

Figure 14: Example of makespan among non-makespan machines being reduced for K-percent Best: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

2.4.5.2 Example of Reducing the Makespan among Non-Makespan Machines

For this example, the initial ready times of machines are 0. Consider the ETC matrix shown in Figure 14(a) for three machines and the following mapping order: t_1, t_2, t_3, and t_4. The percent for this example is set to 70%. This implies that for the original mapping the best two machines are considered for mapping, and for the first iterative mapping only one machine is considered. Allowing only one machine forces the K-percent Best Algorithm to map tasks to the MET machine in the first iterative mapping.

The original mapping using the K-percent Best heuristic is shown in Figures 14(b) and 14(d). The result of the first iterative mapping is shown in Figures 14(c) and 14(e). In the original mapping, t_3 is assigned to m_2. However, in the first iterative mapping t_3 is assigned to m_3. This change in mapping is because the K-percent Best Algorithm considers only one machine for resource allocation in the first iterative mapping, while it had used two machines in the original mapping. The makespan for machines m_2 and m_3 is reduced from 69 in the original mapping to 40 in the first iterative mapping.

2.4.5.3 Example of Increasing Makespan

For this example, the initial ready times of machines are 0. Consider the ETC matrix shown in Figure 15(a) for three machines and the following mapping order: t_1, t_2, t_3, t_4, and t_5. The percent, for this example, is set to 70%. This implies that for the original mapping the best two machines are used for mapping, and for the first iterative mapping only one machine is considered. This is the critical difference between the first iterative mapping and the original mapping. Considering one machine in the first iterative mapping forces the K-percent Best Algorithm to perform like the MET heuristic.

The results of the original mapping are shown in Figures 15(b) and 15(d). The results of the first iterative mapping are shown in Figures 15(c) and 15(e). In the original mapping t_5 is assigned to m_3; however in the first iterative mapping, task t_5 is assigned to m_2. The overall makespan increased from 60, in the original mapping, to 70 in the first iterative mapping. This is because the number of K-% Best machines went down from two to one. This example shows that for K-percent Best Algorithm the makespan can increase, specifically overall makespan.

2.4.6 Switching Algorithm (SWA)

2.4.6.1 Overview of SWA

The Switching Algorithm (SWA) is adapted from [65]. It was designed for use in dynamic environments, but can be used in static environments as well. The switching algorithm is a hybrid of the MET and MCT heuristics. The procedure for SWA is shown in Figure 16. The high and low thresholds are determined experimentally. Examples of the SWA with the iterative approach both improving the makespan among non-makespan machines (see Figure 17) and increasing the overall makespan (Figure 18) can be found even for cases

task	machines (m_1, m_2, m_3)
t_1	60, 100, 120
t_2	10, 20, 40
t_3	20, 40, 30
t_4	50, 30, 40
t_5	60, 20, 25

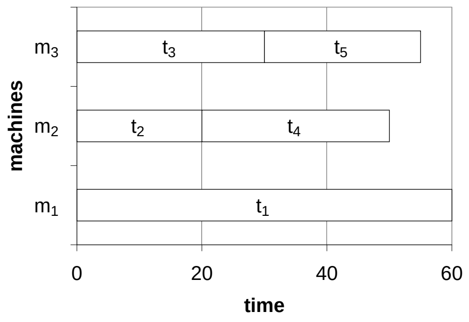
(a)

assignment	machine CT (m_1,m_2,m_3)	K-% machines
$t_1 \rightarrow m_1$	$\underline{60}$, 100, —	m_1,m_2
$t_2 \rightarrow m_2$	70, $\underline{20}$, —	m_1,m_2
$t_3 \rightarrow m_3$	80, —, $\underline{30}$	m_1,m_3
$t_4 \rightarrow m_2$	—, $\underline{50}$, 70	m_2,m_3
$t_5 \rightarrow m_3$	—, 70, $\underline{55}$	m_2,m_3

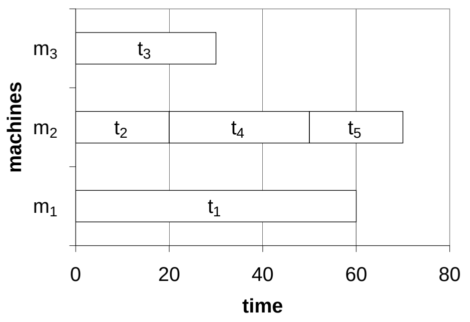
(b)

assignment	machine CT (m_2,m_3)	K-% machines
$t_2 \rightarrow m_2$	$\underline{20}$, —	m_2
$t_3 \rightarrow m_3$	—, $\underline{30}$	m_3
$t_4 \rightarrow m_2$	$\underline{50}$, —	m_2
$t_5 \rightarrow m_2$	$\underline{70}$, —	m_2

(c)



(d)



(e)

Figure 15: Example of makespan increasing for K-percent Best: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

when no special consideration is used to break ties.

The SWA, like the K-percent Best, uses heuristics (MCT and MET) that we proved would not change mappings if ties were broken deterministically (Theorem 1). However, in SWA the heuristic used to assign a task in one iteration (MET or MCT) can change when assigning the same task in the next iteration (e.g., t_4 in Figures 18(b) and 18(c)). In SWA, the choice of using MCT or MET to map as task at a given mapping event depends on the load balance between the machine with the largest computation time, and the machine with the smallest computation time. In general, the calculation of the load balance index when a given task is mapped will change from iteration i to i+1. As a result, different heuristics may be used to map that task at different iterations. Thus, the objective function of this

1 A task list is generated that includes all unmapped tasks in a given arbitrary order.

2 The first task in the list is assigned using the MCT heuristic.

3 The load balance index is calculated for the system
(minimum ready time / maximum ready time over all machines).

4 The heuristic used to map the task is determined as follows:

   i If the load balance index $\geq$ "high threshold," MET is selected to map the next task.
   ii If the load balance index $\leq$ "low threshold," MCT is selected to map the next task.
   iii Otherwise, the current heuristic remains selected.

5 Steps 3-4 are repeated until all tasks have been mapped.

Figure 16: Procedure for using SWA to generate a resource allocation.

heuristic cannot be written as a GCT with fixed λ and η (i.e., λ and η would change as the heuristic used changes between MCT and MET).

2.4.6.2 Example of Reducing the Makespan among Non-Makespan Machines

Consider the ETC matrix shown in Figure 17(a) and the following mapping order: t_1, t_2, t_3, t_4, t_5, and t_6. The initial ready times for all the machines is 0. SWA will switch from MCT to MET when the balance index (BI) is greater than or equal to the high threshold of 0.8, and will switch from MET to MCT when the BI is less than or equal to the low threshold of 0.7.

The original mapping for the SWA is shown in Figures 17(b) and 17(d). In the original mapping, the BI never exceeds 0.8; therefore, all mappings are done using the MCT heuristic.

The first iterative mapping for the SWA example of improvement is shown in Figures 17(c) and 17(e). The difference is that t_4 is assigned to m_3 in the first iterative mapping (instead of m_2). This was different because t_4 was assigned with the MCT heuristic in the original mapping, and in the first iterative mapping t_4 was assigned using MET. The

task	machines (m_1, m_2, m_3)
t_1	40, 70, 50
t_2	60, 50, 80
t_3	70, 90, 60
t_4	100, 30, 21
t_5	110, 30, 30
t_6	60, 120, 110

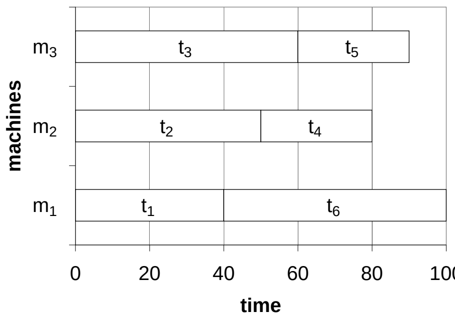
(a)

assignment	machine CT (m_1,m_2,m_3)	ready times (m_1,m_2,m_3)	BI	mapper
$t_1 \rightarrow m_1$	$\underline{40}$, 70, 50	0, 0, 0	x	MCT
$t_2 \rightarrow m_2$	100, $\underline{50}$, 80	*40*, 0, *0*	0	MCT
$t_3 \rightarrow m_3$	110, 90, $\underline{60}$	*40*, 50, *0*	0	MCT
$t_4 \rightarrow m_2$	140, $\underline{80}$, 81	*40*, 50, *60*	2/3	MCT
$t_5 \rightarrow m_3$	150, 110, $\underline{90}$	*40*, *80*, 60	1/2	MCT
$t_6 \rightarrow m_1$	$\underline{100}$, 200, 200	*40*, 80, *90*	4/9	MCT

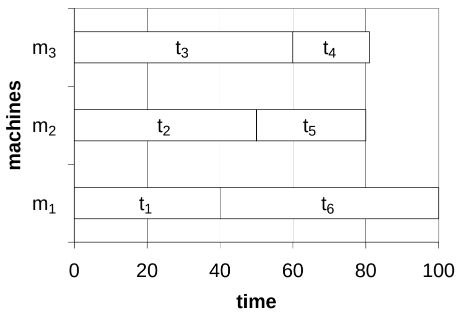
(b)

assignment	machine CT (m_2,m_3)	ready times (m_2,m_3)	BI	mapper
$t_2 \rightarrow m_2$	$\underline{50}$, 80	0, 0	x	MCT
$t_3 \rightarrow m_3$	140, $\underline{60}$	*50*, *0*	0	MCT
$t_4 \rightarrow m_3$	—, $\underline{21}$	*50*, *60*	5/6	MET
$t_5 \rightarrow m_2$	$\underline{80}$, 111	*50*, *81*	50/81	MCT

(c)



(d)



(e)

Figure 17: Example of makespan among non-makespan machines being reduced with SWA: (a) ETC matrix, (b) details of original mapping (largest and smallest RTs are in italic for a given mapping event), (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping (for t_4 mapping event MET is used and CTs do not matter).

makespan among the remaining machines (m_2 and m_3) was reduced from 90 to 81.

2.4.6.3 *Example of Increasing Makespan*

Consider the ETC matrix shown in Figure 18(a) and the following mapping order: t_1, t_2, t_3, t_4, and t_5. The initial ready times for all the machines is 0. SWA will switch from MCT to MET when the BI is greater than or equal to the high threshold of 0.5 and will switch from MET to MCT when the BI is less than or equal to the low threshold of 0.4.

The original mapping is shown in Figures 18(b) and 18(d), and the first iterative mapping is shown in Figures 18(c) and 18(e). Task t_4 is assigned to m_3 (in the first iterative mapping) because the resource allocation of t_4 has a different BI. The makespan machine after the first iterative mapping is m_3 (i.e., the completion time of m_3 becomes greater than the completion time of m_1). The overall makespan increased from 60 in the original mapping to 65 in the first iterative mapping.

2.4.7 Sufferage Algorithm

2.4.7.1 *Overview of Sufferage Algorithm*

The Sufferage Algorithm [14, 23, 31, 38, 44, 47, 52, 65, 69, 72, 84] is shown in Figure 19. In this context, the sufferage value of a task is the difference between its second smallest completion time among all machines and its smallest completion time among all machines. That is, it is the increase in completion time that occurs if the task cannot use its best machine but must use its second best machine instead. Thus, the Sufferage Algorithm is a greedy algorithm that does a limited local search. An example of Sufferage with the iterative approach improving the makespan among non-makespan machines and, in contrast, an example of increasing the overall makespan, can be found even for cases when ties are broken deterministically.

In the Sufferage heuristic, the assignment of a task A to machine B depends on how much it would "suffer" if it was assigned to the machine where A has the second best completion time. The reason why the mapping generated by this heuristic can change is because the machine where A has the second best completion time in the original mapping could easily be the makespan machine. This implies the sufferage value for one or more

task	machines (m_1, m_2, m_3)
t_1	60, 100, 120
t_2	10, 20, 40
t_3	50, 25, 40
t_4	60, 30, 25
t_5	40, 20, 10

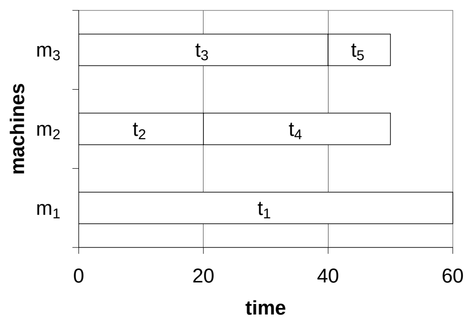
(a)

assignment	machine CT (m_1,m_2,m_3)	ready times (m_1,m_2,m_3)	BI	mapper
$t_1 \rightarrow m_1$	$\underline{60}$, 100, 120	0, 0, 0	x	MCT
$t_2 \rightarrow m_2$	70, $\underline{20}$, 40	*60*, 0, *0*	0	MCT
$t_3 \rightarrow m_3$	110, 45, $\underline{40}$	*60*, 20, *0*	0	MCT
$t_4 \rightarrow m_2$	120, $\underline{50}$, 65	*60*, *20*, 40	1/3	MCT
$t_5 \rightarrow m_3$	100, 70, $\underline{50}$	*60*, 50, *40*	2/3	MET

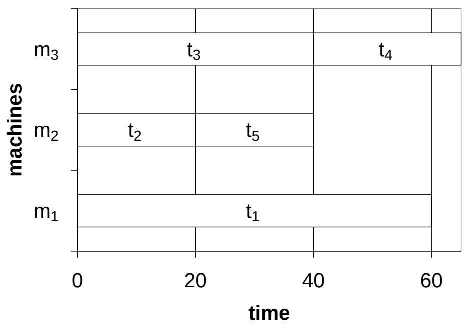
(b)

assignment	machine CT (m_1,m_2,m_3)	ready times (m_1,m_2,m_3)	BI	mapper
$t_2 \rightarrow m_2$	$\underline{20}$, 40	0, 0	x	MCT
$t_3 \rightarrow m_3$	45, $\underline{40}$	*20*, *0*	0	MCT
$t_4 \rightarrow m_3$	—, $\underline{65}$	*20*, *40*	1/2	MET
$t_5 \rightarrow m_2$	$\underline{40}$, 85	*20*, *65*	4/13	MCT

(c)



(d)



(e)

Figure 18: Example of makespan increasing for SWA: (a) ETC matrix, (b) details of original mapping (largest and smallest RTs are in italic for a given mapping event), (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping (for t_4 mapping event MET is used and CTs do not matter).

tasks will be different, and thus change the mapping decision. The Sufferage heuristic is not a GCT IIH, because the function it uses to map cannot be written in the from of Equation 1. Note that for the sufferage algorithm there are two types of ties (a) for a given task to a different machine both have the minimum completion time (step 2.i in Figure 19); and (b) for a given machine two tasks may have the same sufferage values (step 2.i.b in Figure 19).

```
1 A task set ($S$) is generated that includes all unmapped tasks
  in a given arbitrary order.
2 While there are still unmapped tasks:
    i For each machine find the set of tasks that have their
      minimum completion time on this machine.
        a If the set of tasks is size one, then assign the corresponding task to
          the machine and remove the task from $S$.
        b If the size of the set of tasks is greater than one, then assign task with
          the highest sufferage value, and remove that task from $S$.
    ii The ready times for all machines are updated.
```

Figure 19: Procedure for using Sufferage to generate a resource allocation.

2.4.7.2 Example of Reducing the Makespan among Non-Makespan Machines

For this example of improvement, the initial ready times of machines are 0. Consider the ETC matrix shown in Figure 20(a). The original mapping is shown in Figures 20(b) and 20(d), and the first iterative mapping is shown in Figures 20(c) and in Figure 20(e). The difference between the original mapping and the first iterative mapping is that in the original mapping t_4 was assigned to m_1 in the first pass; however, in the first iterative mapping t_4 was assigned to m_2 in the second pass. This is because the sufferage for t_3 increased from 0 to 21 from the original mapping to the first iteration as a result of m_3 being removed. This caused t_3 (instead of t_4) to be mapped to m_1. The makespan for machines m_1 and m_2 is reduced from 60 in the original mapping to 50 in the first iterative mapping.

2.4.7.3 Example of Increasing Makespan

For this example, the initial ready times of machines are 0. Consider the ETC matrix shown in Figure 21(a). The original mapping is shown in Figures 21(b) and 21(d), and the first iterative mapping is shown in Figures 21(c) and 21(e). In the original mapping, t_2 is assigned to m_2 in the first pass. However, task t_2 is assigned to m_3 in the second pass of the first iterative mapping, because the sufferage of t_4 increased in the first iterative mapping as

task	machines (m_1, m_2, m_3)
t_1	50, 20, 50
t_2	71, 80, 70
t_3	50, 71, 50
t_4	10, 30, 20

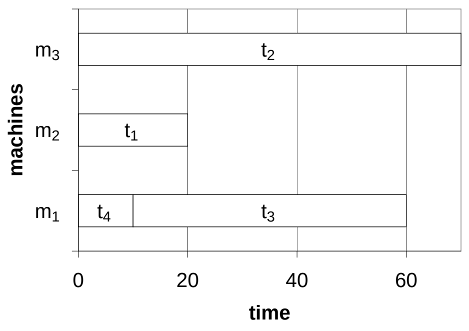
(a)

1st pass	machine CT (m_1, m_2, m_3)	Sufferage
$t_1 \rightarrow m_2$	50, $\underline{20}$, 50	30
$t_2 \rightarrow m_3$	71, 80, $\underline{70}$	1
t_3	$\underline{50}$, 71, 50	0
$t_4 \rightarrow m_1$	$\underline{10}$, 30, 20	10
2nd pass	(m_1, m_2, m_3)	Sufferage
$t_3 \rightarrow m_1$	$\underline{60}$, 91, 120	31

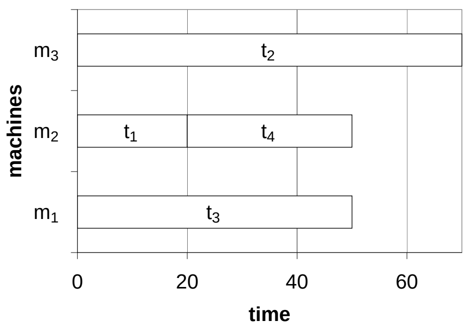
(b)

1st pass	machine CT (m_1, m_2)	Sufferage
$t_1 \rightarrow m_2$	50, $\underline{20}$	30
$t_3 \rightarrow m_1$	$\underline{50}$, 71	21
t_4	$\underline{10}$, 30	20
2nd pass	(m_1, m_2)	Sufferage
$t_4 \rightarrow m_2$	60, $\underline{50}$	10

(c)



(d)



(e)

Figure 20: Example of makespan among non-makespan machines being reduced for Sufferage: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

a result of m_1 being removed. This increase caused t_4 be assigned at a different pass of the heuristic. This in turn forces t_1 to be mapped in the 3^{rd} pass, instead of the second pass, and to a different machine, i.e., m_2 resulting in larger makespan. The overall makespan increases from 100 in the original mapping to 108 in the first iterative mapping.

2.5 Analysis of Characteristics of Heuristics

In the analysis of the heuristics that use the iterative approach, we found that there are three main situations in which a mapping can change. The first situation is when a mapping will change from one iteration to the next if ties are broken randomly. This situation occurred with the Min-Min, MCT, and MET heuristics. If ties are instead broken

task	machines (m_1, m_2, m_3)
t_1	80, 70, 91
t_2	70, 50, 70
t_3	100, 160, 160
t_4	20, 38, 60

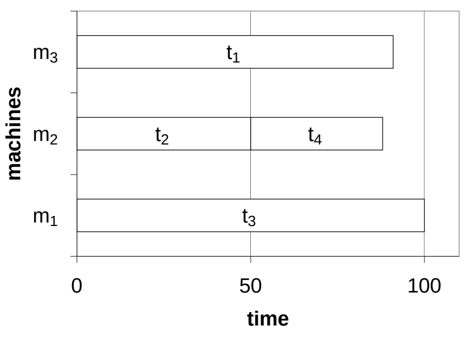
(a)

1st pass	machine CT (m_1, m_2, m_3)	Sufferage
t_1	80, $\underline{70}$, 91	10
$t_2 \rightarrow m_2$	70, $\underline{50}$, 70	20
$t_3 \rightarrow m_1$	$\underline{100}$, 160, 160	60
t_4	$\underline{20}$, 38, 60	18
2nd pass	(m_1, m_2, m_3)	Sufferage
$t_1 \rightarrow m_3$	180, 120, $\underline{91}$	29
t_4	120, 88, $\underline{60}$	28
3rd pass	(m_1, m_2, m_3)	Sufferage
$t_4 \rightarrow m_2$	120, $\underline{88}$, 151	32

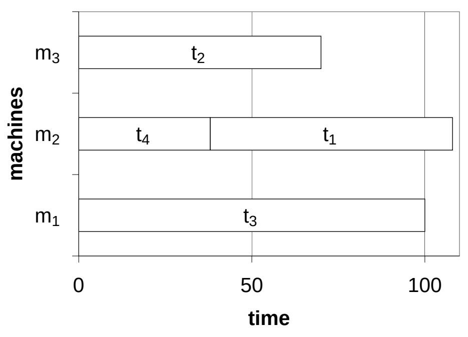
(b)

1st pass	machine CT (m_2, m_3)	Sufferage
t_1	$\underline{70}$, 91	21
t_2	$\underline{50}$, 70	20
$t_4 \rightarrow m_2$	$\underline{38}$, 60	22
2nd pass	(m_2, m_3)	Sufferage
t_1	108, $\underline{91}$	17
$t_2 \rightarrow m_3$	88, $\underline{70}$	18
3rd pass	(m_2, m_3)	Sufferage
$t_1 \rightarrow m_2$	$\underline{108}$, 161	53

(c)



(d)



(e)

Figure 21: Example of overall makespan increasing for Sufferage: (a) ETC matrix, (b) details of original mapping, (c) details of first iterative mapping, (d) graphical representation of original mapping, and (e) graphical representation of first iterative mapping.

deterministically, then these heuristics are GCT IIHs (as proved in Section 2.3.5). It is important to use these observations and apply them to heuristics that were not considered in this study. An example of a different heuristic that is also a GCT IIH is the Opportunistic Load Balancing (OLB) [17, 63, 65] heuristic. OLB assigns each task, in a given arbitrary order, to the machine with the smallest ready time. The OLB heuristic uses a GCT function with $\lambda = 0$ and $\eta = 1$ (recall that MET has $\lambda = 1$ and $\eta = 0$). There are other IIHs, and similar proofs can be derived for them.

The second situation is when a mapping can change from one iteration to the next independently of how ties are broken. This situation occurred with the K-percent Best, SWA, and Sufferage heuristics. The main characteristic of the second category of heuristics is that additional information, other than a GCT function, is used to determine assignments. Intuitively, this means that the assignment of task A to machine B depends on the state of the other available machines; however, the set of available machines changes from one iteration to the next. An additional example of this second type of heuristic is the Max-Min heuristic [13, 42]. In Max-Min, we find the machine that has the minimum completion time for each task and from these task-machine pairs we select the pair that has the maximum completion time. Consider a situation with multiple tasks and machines where t_1 has its minimum completion time on m_1 in the original mapping, and t_2 has its minimum completion time on m_2. In this example, the completion time of t_2 on m_2 is greater than the completion time of t_1 on m_1 ($GCT(t_1, m_1, 1, 1) < GCT(t_2, m_2, 1, 1)$). Therefore, t_2 gets assigned to m_2 ($t_2 \rightarrow m_2$). However, assume m_1 was the makespan machine of the original mapping and t_1 has a new minimum on m_2. This new minimum is greater than the completion time of assigning t_1 to m_1. This could cause $t_1 \rightarrow m_2$ to be greater than $t_2 \rightarrow m_2$ ($GCT(t_1, m_2, 1, 2) > GCT(t_2, m_2, 1, 2)$) causing t_1 to be assigned to m_2 and t_2 to be assigned to a different machine. Max-Min is not a GCTIIH because its task to machine assignment is done with the following equation:

$$\Gamma_{k,n}, \beta_{k,n} = \underset{t \in T_{k,n}}{\operatorname{argmax}} \left\{ \underset{m \in M_{k,n}}{\operatorname{argmin}} \left(GCT(t, m, n, k) \right) \right\}. \tag{14}$$

This equation is different from Equation 2, therefore it is not a GCT IIH.

The third situation is when a mapping will result in either an improvement or no change. The GA based approach was the only heuristic considered in this study where the overall makespan could not increase; however, no improvement can be guaranteed. This was accomplished using the best chromosome from the previous iteration and seeding it into the current iteration. This usage of seeding comes very naturally to the GA heuristic; however, the same concept can also be applied to the other heuristics in this study, e.g., KPB, SWA, Sufferage. If we have heuristics where the mapping can change (with the iterative approach) then we can compare the mapping of the non-makespan machines from iteration k and the mapping from iteration $k + 1$ and keep the mapping with the smallest makespan. This would cause the best solutions to be preserved across iterations, thus changing the mapping only if a better makespan is found.

2.6 Conclusions

An iterative approach for minimizing the finishing times of machines in a heterogeneous computing environment was proposed. The performance of several heuristics was analyzed for such an approach. The greedy heuristics studied (Min-Min, MCT, MET, SWA, K-percent Best, and Sufferage) did not guarantee an improvement in the completion time among non-makespan machines. The Min-Min, MCT, and MET heuristics with deterministic tie breaking were characterized as GCT IIHs, and were mathematically proven to not change mappings. However, the Min-Min, MCT, and MET heuristics with random tie breaking can change the makespan among remaining machines (i.e., improve or get worse, even causing the overall makespan to increase). The Switching Algorithm, K-percent Best, and Sufferage heuristics all produced mappings with the iterative approach where the makespan among remaining machines can change. The main characteristic that SWA, K-percent Best, and Sufferage share is that the assignment of a given task to a given machines depends on the state of other machines. It was also determined that these heuristics use additional information other than the GCT function to assign tasks. The GA-based approach using elitism will keep the same mapping or produce a better mapping and therefore guarantees the overall makespan will not increase. In summary, the most important characteristic to note when implementing the iterative approach with on of the heuristics is that one must apply elitism across iterations to guarantee the makespan among available machines at each iteration does not become worse.

In this study, we aim to minimize the maximum completion time of machines,i.e., minimizing the makespan. Analogously we could try to minimize the maximum utilization or minimize the maximum power consumption of a computer. Thus, this approach can be used in various minmax environments.

CHAPTER III

ROBUST RESOURCE ALLOCATION FOR WEATHER DATA PROCESSING ON A HETEROGENEOUS PARALLEL SYSTEM

3.1 Introduction

A space-based weather monitoring system considered in this work consists of three major components: a satellite positioning system, the satellite with its data collection sensors, and the data processing system (see Figure 22). The satellite positioning system is responsible for requesting (from the satellite) the data that must be collected, and sending the associated processing tasks to the distributed computing platform. Imaging across a variety of spectral bands is collected by the satellite, and is transmitted to the data processing system. This research can also apply to different satellite image processing environments, e.g., entertainment and homeland security.

The weather imaging data sent down by the satellite (the **data set**) must be processed before it is of any value to the users. A new weather data set is received periodically, and the current data set must be processed before the next data set arrives. A similar requirement is used in the satellite image processing in [49]. The tasks to be executed on the data set can be classified into two broad categories: (a) critical telemetry, tracking, and control (**TTC**); and (b) non-critical data processing and data research [67]. Currently systems used for processing the data sets, at a typical site, are divided into three distinct sets of processing elements (dedicated to TTC, data processing, and data research). As a result of this partitioning of a given data set, one system may be overloaded while another is underloaded.

The goal of this research is to develop a resource manager so that a smaller heterogeneous

A preliminary version of portions of this research appeared in [68].

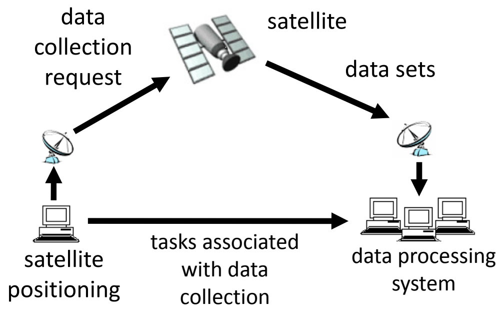


Figure 22: Overview of a space-based weather system.

global bank of shared common resources can replace the three sets of processing elements and operate efficiently. The global bank will reduce the cost of the system, while being financially viable. This platform is a heterogeneous computing system (**HCS**), because machines are typically added or replaced over time with new machines. Therefore, tasks may have different execution times on different machines, and thus have greater affinity to certain machines.

The allocation of tasks to machines is a static mapping problem [2], because all the tasks that need to be executed are known *a priori* (before the data set to be processed arrives). However, it has some characteristics of dynamic mapping in that tasks are known a short time in advance so the resource allocation must be performed in a short amount of time, e.g., 5 minutes.

In an HCS, the assignment of tasks to machines to obtain a near-optimal resource allocation is an important research problem. The act of assigning (**matching**) each task to a machine and ordering (**scheduling**) the execution of the tasks in each machine is known as **mapping**, **resource allocation**, or **resource management**. The mapping problem has been shown, in general, to be NP-complete (e.g., [26, 34, 42]). Hence, the development of heuristic techniques to find near optimal solutions is an active area of research (e.g., [2, 33, 35, 86]).

The performance of computing systems is susceptible to degradation due to unpredictable circumstances. Therefore, it is necessary to allocate resources to tasks so that the robustness of the system in response to unpredictable events is maximized [5]. For this study, the times between the arrival of data sets can vary, i.e., it is uncertain. The next data set may arrive earlier than expected. Because the current data set is discarded when the next data set arrives, it is important for the resource allocation to be robust against an early arrival of the next data set. Thus, it is necessary to develop a performance metric to evaluate the robustness of a mapping produced by the heuristics. The contributions of this research are: (1) derived a formal mathematical model for a proposed real world weather image processing system; (2) defined a new robustness metric for making resource allocation decisions; (3) the development of a two-part approach for scheduling **high priority tasks** (**HPT**), and **revenue generating tasks** (**RGT**) in an oversubscribed system where each part has a different robustness criteria; (4) designed and evaluated new problem domain specific heuristics for developing resource allocations; and (5) the derivation of a bound on the performance of a resource allocation for the proposed HCS.

The remainder of this paper is organized as follows. A detailed overview of the system model is given in Section 3.2. Section 3.3 discusses the related work. The heuristics for HPT and RGT are explained in Sections 3.4 and 3.5, respectively. Section 3.6 describes the simulation setup used for the experiments, and the bounds on the performance of a resource allocation are presented in Section 3.7. The experimental results are discussed in Section 3.8. In Section 3.9, the conclusions are presented.

3.2 System Model

3.2.1 Overview

In this study, there are a set of $\boldsymbol{T}$ tasks that must be executed on $\boldsymbol{M}$ heterogeneous machines for a <u>given</u> data set. It is assumed that all the tasks associated with a data set must arrive at a predetermined time before the expected arrival time of the data set. Therefore, all the tasks associated with a data set are known *a priori*, and the mapping problem is a **static** mapping problem [2, 17]. A new data set arrives from the satellite after an interval of $\boldsymbol{\tau}$ time units. The high priority tasks from this new data set are needed to steer the satellite. We need the computing power to schedule these high priority tasks in order to decide where to move the satellite next. Therefore, when a new data set arrives, all tasks associated with the old data set are dropped, and the machine queues are emptied. In some scenarios dealing with large image data sets, it may difficult to keep multiple data sets from multiple time intervals. Furthermore, for the case where the same geographical location is being monitored, it is better to use the latest data. In our environment, we will need to stop executing revenue tasks to execute the new high priority tasks.

An oversubscribed system is considered because it makes meeting robustness constraints more difficult to accomplish. In this system, not all tasks (for the current data set) can be completed by the expected arrival time of the next data set. The oversubscribed scenario allows the comparison of the heuristics in a different environment.

The expected arrival time of the next data set, $\boldsymbol{\tau_{expected}}$, is only an estimate and the next data set might arrive earlier than expected. The estimated time to compute (**ETC**) values of each task over different data sets on each machine are assumed to be known. The assumption of such ETC information is a common practice in resource allocation research (e.g., [37, 46, 50, 58, 88]). In this system, we consider that task preemption is not allowed. Let the **machine ready time** be the time at which a machine would be able to start the execution of a currently unassigned task.

Execution of the high priority tasks ensure the proper functioning of the system; therefore, it is necessary to provide a guarantee that these tasks are completed. The revenue tasks are important to ensure the system is financially viable. Because of the difference in measures between high priority tasks and revenue tasks, the resource allocation is separated into two parts. Consider that $\boldsymbol{makespan_{HP}}$ is the completion time of the last high-priority task to finish, $\mathbf{Dataset}_{\boldsymbol{i}}$ is the time when data set i arrives, and $\boldsymbol{\Delta\tau}= \tau_{expected} - \tau$. An illustration of this notation is shown in Figure 23.

3.2.2 Robustness

A procedure for deriving a robustness metric for an HCS referred to as **FePIA** (Features, Perturbations, Impact, and Analysis) is introduced in [5]. The FePIA procedure addresses three fundamental questions [4]: (a) What behavior of the system makes it robust? (b) What uncertainties is the system robust against? (c) Quantitatively, exactly how robust is

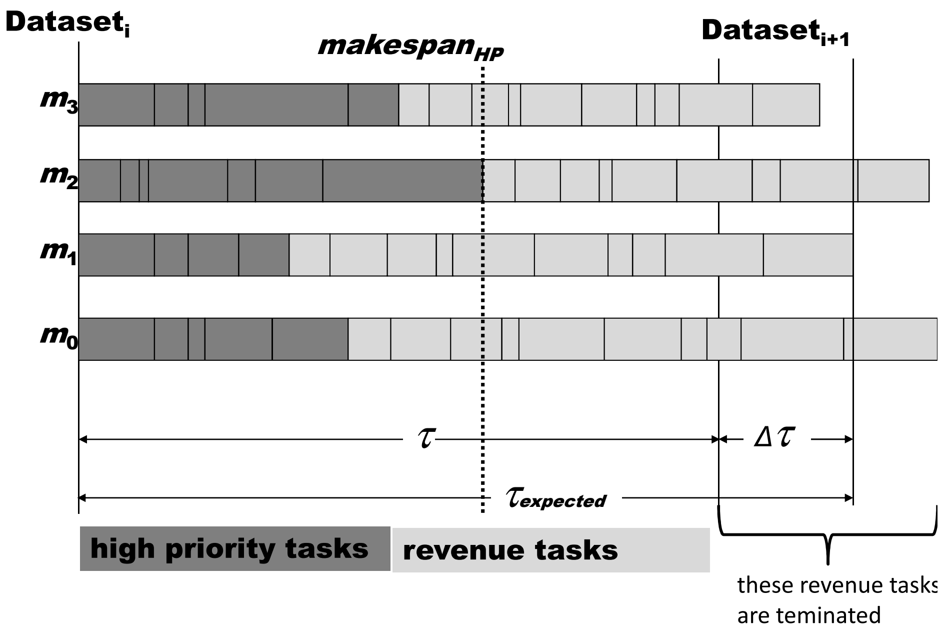


Figure 23: Illustration of a resource allocation that includes both high priority and revenue tasks. Rectangles represents tasks (dark high priority and light shaded areas show revenue tasks)

the system?

Using the FePIA procedure, we define what behavior makes the first part robust. For HPT, a resource allocation is robust if all high priority tasks finish before $\tau_{expected}$, and the robustness is quantified by the difference between $\tau_{expected}$ and the makespan of high priority tasks ($makespan_{HP}$). Therefore, this system needs to be robust against uncertainty in the arrival time of the next data set. By minimizing $makespan_{HP}$, we maximize time difference between the maximum completion time of these tasks and $\tau_{expected}$. The robustness for HPT ($\boldsymbol{\rho_1}$) is quantified as

$$\rho_1 = \tau_{expected} - makespan_{HP}. \tag{15}$$

The part 2 tasks generate a revenue, but are not critical to the operation of the satellite. Each revenue task i ($\boldsymbol{t_i}$) has an associated revenue of $\boldsymbol{R(t_i)}$. The revenue generated by the satellite should cover the costs associated with the continued operation of the satellite (e.g., labor, facilities).

For RGT, we define robustness of a research allocation as follows. The revenue generated by the satellite for each data set needs to be larger than the cost associated with processing that data set for the system to be robust. Let $\boldsymbol{C_{total}}$ be the cost associated with processing a data set, and $\boldsymbol{CTasks(\tau)}$ be the set of revenue tasks completed before the arrival of the next data set; this set is a function of τ. The system is robust if

$$C_{total} \leq \sum_{\forall t_i \in CTasks(\tau)} R(t_i)\ . \tag{16}$$

The uncertainty considered when assigning both HPT and RGT is the arrival time τ of the next data set. The robustness metric for RGT ($\boldsymbol{\rho_2}$) is the maximum value of $\Delta\tau$ ($\tau_{expected} - \tau$) such that the revenue is equal to or exceeds the cost, and is calculated using the following equation:

$$\rho_2 = \max \Delta\tau : C_{total} \leq \sum_{\forall t_i \in CTasks(\tau_{expected} - \Delta\tau)} R(t_i)\ . \tag{17}$$

3.3 Related Work

There are many definitions of robustness for various environments (e.g., [5, 8, 66, 74]). For this study, we use the concept of robustness proposed in [5], where the robustness of a resource allocation in a parallel and distributed computing system is discussed—including the FePIA procedure. Our study applies the robustness concept to an environment that is required to be profitable, where revenue is earned by executing tasks. The two studies in [5] have very different robustness metrics, despite being generated with the same procedure.

The study in [8] discusses a job shop environment that is susceptible to sudden changes that render an existing schedule infeasible. In our study, we use a robustness metric that quantifies the largest change in the arrival time of the next data set that still meets the performance constraints.

The work in [74] defines a robustness metric for systems that use a stochastic model for task execution times. One major difference between this study and the work in [74] is that we model estimated task execution time as a fixed value, while [74] models the execution time as a random variable. The other difference is the use of a task-profit paradigm to generate a robustness metric that guarantees the system is profitable.

The work in [66] describes a dynamic mapping environment, where the resource allocation must be robust against uncertainty in the estimated execution time of the tasks. Two variations are considered: (a) maximizing robustness with a makespan constraint, and (b) minimizing makespan with a robustness constraint. Our study shares some similarities with the goal of variation (a) because we maximize robustness based on the estimated arrival time of data sets. However, it has three main differences: we consider a static rather than dynamic environment, our uncertainty parameter is different, and we use a different robustness metric.

The work in [80] studies two related scenarios. In the first scenario, there is a fixed machine set and the goal is to maximize a robustness metric. In the second scenario, the goal is to maximize robustness with a dollar cost constraint on the set of machines that can be used. In both our study and [80] there is a makespan constraint. However, the uncertainty in our study is the arrival time of the next data set and not the execution time of a task. Also, the robustness metric in this study is based on guaranteeing a profit while the robustness metric in [80] is the smallest collective increase in task execution time that will make the system violate its makespan constraint.

In [10] the authors describe an oversubscribed system for scheduling communications for a satellite range scheduling problem. Each task has a priority and a deadline associated with it, and not all tasks can be scheduled before their deadlines. The goal is to minimize the number of tasks that cannot complete before their deadline. This problem is similar to our study because we schedule based on task priority, i.e., assign high priority tasks first, and also the money generated by revenue tasks can be considered as a priority. However, in our study, the profitability of the system is robust against uncertainties in the arrival time of the next data set. This does not necessarily correlate to minimizing the number of tasks that cannot complete before the deadline.

The work in [55] also discusses an oversubscribed environment of tasks with multiple priorities, but emphasizes that task priorities must be rigidly respected, i.e., a higher priority task can never be traded for a set of low priority tasks. Our study is similar because it is divided into a two-part scheduling problem, where the high priority tasks must be completed

(1) A list is generated that includes all unmapped tasks in a given arbitrary order.

(2) The first task in the list is assigned to its minimum execution time machine.

(3) The next task in the list is selected.

(4) Steps (2)–(4) are repeated until all the tasks have been mapped.

Figure 24: Procedure for using MET to generate a resource allocation.

before the revenue tasks can be considered. However, for revenue tasks there is a trade-off. Another significant difference between our work and the work in [55] is that we study how the system performs when there is uncertainty in the arrival time of the next data set.

3.4 Heuristics for High Priority Tasks

3.4.1 Overview

Six static heuristics are discussed here: five greedy heuristics and a Genitor Algorithm. We consider two types of greedy heuristics: one-phase and two-phase. The one-phase heuristics for HPT are Minimum Execution Time, Minimum Completion Time, and K-Percent Best, and the two-phase heuristics are MinCT-MinCT and MaxCT-MinCT. These heuristics were chosen because they have performed well in similar environments. The Genitor heuristic (a steady state genetic algorithm) was implemented for comparison purposes; this Genitor approach cannot be fielded in a live system because of time constraints on the heuristic runtime.

3.4.2 Minimum Execution Time (MET)

The Minimum Execution Time (**MET**) [17, 42] heuristic considers tasks in an arbitrary order, and maps a task t_i to the machine j that has the smallest $ETC(i, j)$ for that task. The assignment obtained by the MET is independent of the task ordering [20]. Therefore, a random task ordering was used. The procedure for the MET heuristic is shown in q111111 Figure 24.

(1) A list is generated that includes all unmapped tasks in a given arbitrary order.

(2) The first task in the list is assigned to its minimum completion time machine (machine ready time plus estimated computation time of the task on that machine).

(3) The task selected in step (2) is removed from the list.

(4) The ready time of the machine where the task is assigned is updated.

(5) Steps (2)–(4) are repeated until all the tasks have been mapped.

Figure 25: Procedure for using MCT to generate a resource allocation.

3.4.3 Minimum Completion Time (MCT)

The Minimum Completion Time (**MCT**) [17, 42] heuristic considers the tasks in a given random order. Each task is mapped to the machine that completes the task soonest, where the completion time (**CT**) of t_i on machine j is the ready time for machine j plus $ETC(i, j)$. The procedure for the MCT heuristic is shown in Figure 25.

3.4.4 K-Percent Best (KPB)

For the K-Percent Best (**KPB**) [65] heuristic, the "K-percent" of the machines with the smallest execution time for a given task are identified. The task is mapped to the machine in this subset that has the minimum completion time. A "K" value of $1/M\%$ causes this heuristic to be the same as with MET, while 100% implies that the heuristic is the same as MCT. Different values of K were explored, and it was found that the best average results across all ETC consistencies were obtained when K was equal to 50%. A random task ordering of tasks was used for the KPB heuristic. The procedure for the KPB heuristic is shown in Figure 26.

3.4.5 MinCT-MinCT

MinCT-MinCT [17,42,65] is a two-phase greedy heuristic based on the minimum completion time of the tasks. The procedure used for the MinCT-MinCT heuristic is shown in Figure 27.

(1) A list is generated that includes all unmapped tasks in a given arbitrary order.

(2) For the first task, a subset is formed by identifying the $M \cdot (\frac{K}{100})$ machines with the smallest execution times for the task.

(3) The task is assigned to the machine that provides the minimum completion time in the subset (ties are broken randomly).

(4) The task is removed from the list.

(5) The ready time of the machine where the task is assigned is updated.

(6) Steps (2)–(5) are repeated until all tasks have been assigned.

Figure 26: Procedure for using K-Percent Best to generate a resource allocation.

(1) A list is generated that includes all the unmapped tasks.

(2) For each task in the list, the machine that gives the task its minimum completion time (first "Min") is determined (ignoring other unmapped tasks).

(3) Among all task-machine pairs found in (2), the pair that has the minimum completion time (second "Min") is determined.

(4) The task selected in (3) is removed from the list and is assigned to the paired machine.

(5) The ready time of the machine where the task is mapped is updated.

(6) Steps (2)–(5) are repeated until all tasks have been mapped.

Figure 27: Procedure for using MinCT-MinCT to generate a resource allocation.

(1) A list is generated that includes all unmapped tasks.

(2) For each task in the list, the machine that gives the task its minimum completion time is determined (ignoring other unmapped tasks).

(3) Among all task-machine pairs found in (2), the pair that has the maximum completion time is determined.

(4) The task selected in (3) is removed from the list and is assigned to the paired machine.

(5) The ready time of the machine where the task is mapped is updated.

(6) Steps (2)–(5) are repeated until all tasks have been mapped.

Figure 28: Procedure for using MaxCT-MinCT to generate a resource allocation..

3.4.6 MaxCT-MinCT

The MaxCT-MinCT [17,42,65] heuristic is similar to the MinCT-MinCT heuristic. However, instead of selecting the task-machine pair with the smallest completion time, this heuristic selects the task-machine pair that has the largest completion time. The intuition behind selecting the tasks with larger execution times first is to prevent the mapping of a long task on top of a virtually balanced loading [17]. The procedure used for the MaxCT-MinCT heuristic is shown in Figure 28.

3.4.7 HPT Genitor

Genetic Algorithms (**GA**) have been shown to work well for numerous problem domains, including resource allocation and job shop scheduling, e.g., [17, 25, 27]. The Genitor algorithm is a steady-state GA [85]. It uses a ranked population and only does one crossover operation per generation. Genitor (like other GAs) uses **chromosomes** to represent possible solutions, e.g., all tasks and the machines to which they are assigned. The Genitor heuristic implemented here is a variation of Genitor described in [85].

Based on experimentation the population size used for this study was 200 chromosomes. The chromosome is a vector of length T (number of tasks), and the i^{th} element of the vector is the machine to which t_i is assigned. The population is seeded with the best solution generated among the greedy heuristics for each specific simulation. The remaining

199 chromosomes are generated randomly. Multiple copies of a chromosome are not allowed in the initial population to reduce the probability of premature convergence.

The population is sorted in descending order of robustness radius (ρ_1). **Elitism**, the property that guarantees the best solution remains in the population, is implicitly implemented in Genitor by always maintaining a sorted list.

For crossover, two parents are selected using the linear bias approach [85], and two point crossover is used for the selected parents. The machine assignments from the part of the chromosome between the crossover points is exchanged between parents, and two new offspring are generated.

For mutation, each entry in an offspring has a 0.01 probability of being randomly selected. For each mutated entry, a random machine assignment (from 1 to M) is chosen to replace the old machine assignment.

Each offspring is then evaluated and must compete for inclusion in the population. If the new offspring has a larger robustness radius than the worst member in the population, then the offspring is inserted in sorted order into the population, and the worst chromosome is removed. Otherwise, the new offspring is discarded. The heuristic is stopped after one hour (recall the Genitor is used just for comparison), and the best solution is selected.

The procedure for the HPT Genitor heuristic is shown in Figure 29. The population, probability of mutation, and linear bias parameters were determined experimentally.

3.5 Heuristics for Revenue Generating Tasks

3.5.1 Overview

For the second part of the problem, machine ready times are the finishing times found by the best HPT heuristic. Several heuristics were implemented for this part. Of these, MET, MCT, KPB, and MinCT-MinCT are the same as described in Section 3.4 with revenue being used to determine the order in which tasks are assigned. The MaxW-MaxW, MaxWPTU-MaxWPTU, and MaxWPTU-MinCT heuristics are greedy heuristics similar in structure to the MinCT-MinCT heuristic but using different objective functions. A Genitor-based heuristic also was implemented for comparison only, due to its long execution time.

```
1 An initial population is generated.
2 While the stopping criteria is not met (i.e., heuristic execution time
  is less than 1 hour):
    a Two chromosomes in the population are probabilistically
      selected as parents for crossover using the linear bias function.
        i  Two random cut-off points are generated.
        ii The machine assignments of the tasks between
           the cut-off points are exchanged.
    b Each entry in an offspring chromosome has a chance of being mutated.
    c The resultant offspring of the crossover and mutation
      are inserted in the ranked population, and the two
      worst chromosomes are discarded.
3 The best solution is output.
```

Figure 29: Procedure for using Genitor to generate a resource allocation.

3.5.2 Greedy Heuristics

For the one-phase greedy heuristics (MET, MCT, and KPB), three different orderings of tasks were used. These orderings are random orderings (**RAND**), revenue per time unit (**RPTU**), and revenue averaged per time unit (**RAPTU**) among all machines. These orderings are calculated as follows:

(a) RAND: Random ordering of tasks.

(b) RPTU: For each task i that needs to be mapped, $\boldsymbol{RMIN_i}$ is calculated using the following equation:

$$RMIN_i = \frac{R(t_i)}{\min_{1 \le j \le M} ETC(i, j)} \ . \tag{18}$$

The tasks that need to be mapped are sorted in descending order based on $RMIN_i$.

(c) RAPTU: For each task i that needs to be mapped, $\boldsymbol{RAVG_i}$ is calculated using the following equation:

$$RAVG_i = \frac{R(t_i)}{\sum_{1 \le j \le M} (ETC(i, j)/M)} \ . \tag{19}$$

The tasks that need to be mapped are sorted in descending order based on $RAVG_i$.

(1) Generate a list of all the unmapped tasks.

(2) For each task in the list, find the machine j that gives the maximum worth.

(3) For all the task-machine pairs found in step (2), select the pair that has the maximum worth value.

(4) Assign the selected task, remove it from the list, and update the finishing time of the machine.

(5) Repeat steps (2) – (4) until all the tasks have been mapped.

Figure 30: Procedure for using MaxW-MaxW to generate a resource allocation.

3.5.3 MaxW-MaxW

This heuristic is similar in structure to MinCT-MinCT, but instead of minimizing completion time it maximizes a "worth" value. Let $\boldsymbol{F_{ij}}$ be the completion (finishing) time of task i on machine j. For a task i on a machine j, the likelihood ($\boldsymbol{L_{ij}}$) of task i completing on machine j before the deadline is defined as

$$L_{ij} = \frac{\tau_{expected} - F_{ij}}{\tau_{expected}} \ . \tag{20}$$

The worth value ($\boldsymbol{w_{ij}}$) is based on L_{ij} and is calculated using the following equation:

$$w_{ij} = L_{ij} \cdot R(t_i) \ . \tag{21}$$

The procedure used to implement MaxW-MaxW is shown in Figure 30. For the MaxW-MaxW heuristic, the step (2), can be replaced by just finding the minimum completion time machine.

3.5.4 MaxWPTU-MaxWPTU

The MaxWPTU-MaxWPTU heuristic is a Max-Max heuristic similar in structure to MaxW-MaxW. However, this heuristic is based on <u>w</u>orth <u>p</u>er <u>t</u>ime <u>u</u>nit (**WPTU**), calculated as follows:

$$wptu_{ij} = \frac{w_{ij}}{ETC(i,j)}. \tag{22}$$

The procedure for MaxWPTU-MaxWPTU is the same as the procedure shown in Figure 30; however, worth per time unit is substituted for worth.

3.5.5 MaxWPTU-MinCT

The MaxWPTU-MinCT heuristic is similar to the MaxW-MaxW heuristic previously described. MinCT finds for each unmapped task the task/machine pair with the smallest completion time, then MaxWPTU selects the task/machine pair with the maximum WPTU.

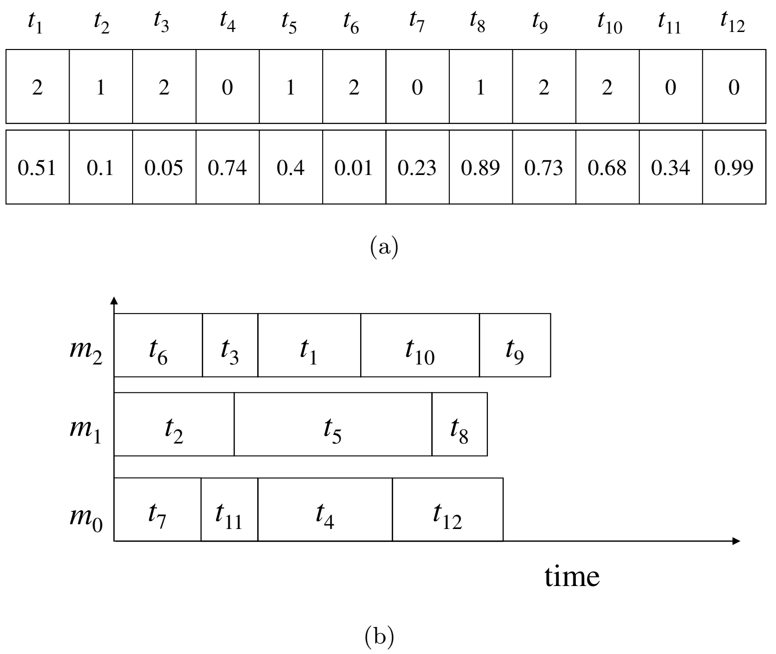
3.5.6 RGT Genitor

The RGT Genitor is similar to the HPT Genitor shown in Figure 29. However, the RGT Genitor requires information about the assignment of tasks to machines, and the ordering of these tasks in the machine queue. This information is represented with two chromosome strings that are illustrated in Figure 31(a). The top string represents the assignment of t_i to machine j. The bottom string is a real number from 0 to 1 that represents the relative ordering of a task in a machine queue. In Figure 31(b), the string from Figure 31(a) is converted to a mapping. To understand how the chromosome string is converted into a mapping, we can observe the tasks assigned to machine 0. The task-real number pairs assigned to machine 0 are: $t_4 - 0.74$, $t_7 - 0.23$, $t_{11} - 0.34$, and $t_{12} - 0.99$. The real number is used to arrange the tasks in ascending order within the machine queue (e.g., t_{11} executes before t_4 because $0.34 < 0.74$).

This implementation of the RGT Genitor uses 200 chromosomes. The best solution generated among the greedy heuristics was used as a seed in the RGT Genitor. The rest of the population is created by generating a random assignment of tasks to machines, and a uniform random variable ($U(0,1)$) for the relative ordering for each task. The chromosomes are sorted in a ranked list based on descending order of ρ_2.

The crossover for the RGT Genitor is done by selecting two parents using linear bias [85]. For the two selected parents, two crossover points are randomly generated. Between these two crossover points, the machine assignments and the real numbers are exchanged among parents. This crossover procedure generates two new offspring.

The mutation is done on both offspring. For each entry i in the machine assignment string of the offspring, with X% probability of mutation (determined experimentally) reassign task i to a randomly selected machine. After the machine assignment string is mutated,


t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
2 1 2 0 1 2 0 1 2 2 0 0
0.51 0.1 0.05 0.74 0.4 0.01 0.23 0.89 0.73 0.68 0.34 0.99
(a)
m_2 t_6 t_3 t_1 t_{10} t_9
m_1 t_2 t_5 t_8
m_0 t_7 t_{11} t_4 t_{12}
time
(b)
Figure 31: Chromosome representation for RGT Genitor: (a) chromosome strings, (b) machine queues.

the random number string is mutated. For each entry i of the random number string of the offspring, with $X\%$ probability of mutation (determined experimentally), a new random number ($U(0, 1)$) is generated and it replaces the entry for task i.

After the crossover and mutation operations are done, the offspring are evaluated and inserted into the sorted population; the two worst chromosomes are discarded from the population (i.e., the size of the population remains constant). This process is repeated until the stopping criteria is met (i.e., heuristic execution time reaches one hour).

3.6 Simulation Setup

The simulation environment was intended to represent a typical satellite data processing system. The environment we used to evaluate and compare the heuristics had 8 machines ($M = 8$), and 2048 total tasks (subdivided into: 512 high priority tasks, and 1536 revenue tasks). In this environment, users submit requests to process a provided data set by one of a set of well-known tasks. For this simulation, the estimated time to compute a task t_i on machine j ($ETC(i, j)$) was generated using the coefficient of variance (**COV**) based method, described in [3].

Three different kinds of ETC matrices were generated for this simulation; to represent a variety of different types of actual heterogeneity: consistent, inconsistent, and partially-consistent. For this study, 100 different ETCs were generated for each type of consistency (consistent, inconsistent, and partially consistent). The results presented are averaged for each consistency over the 100 runs. For a consistent ETC matrix, if t_i has a lower execution time on machine x than machine y, then the same is true for any t_k. For an inconsistent ETC matrix, if t_i has a lower execution time on machine x than machine y, then there exists a task t_k with a lower execution time on machine y than machine x. A combination of these two cases is the partially-consistent ETC matrix, which is an inconsistent matrix with a consistent sub-matrix [3]. For the partially-consistent matrices simulated here, the consistent sub-matrix was 50% of the tasks for 50% of the machines.

To simulate the diverse task mixtures in a real system, the COV for task heterogeneity was 0.1 and machine heterogeneity was set to 0.4, i.e., low task - high machine heterogeneity. The simulation parameters were configured to ensure an oversubscribed system, and provide a sufficient challenge for the mapping heuristics. The mean time to execute the tasks was set to 7.5 seconds. Also, the estimated arrival time of a new data set was $\tau = 900$ seconds. The e̲xecution r̲atio of a t̲ask ($\mathbf{ERT}_i$) is the average execution time of a task i over all machines divided by the average execution time of all tasks across all machines.

In an actual system, revenue for each task is negotiated between the system provider and the user. For our simulation studies, the $R(t_i)$ values for revenue task i are computed by multiplying the ERT_i and a sample from a Gamma distribution (mean of 200 and standard deviation of 50).

The simulations were run on an Intel Core 2 Duo T8100 (2.1GHz), with 4Gb RAM running a Windows Vista OS. The code was written in C++ and run on cygwin.

3.7 *Bounds*

3.7.1 HPT Upper Bound (UB1)

To compute an upper bound (**UB1**) on the robustness radius of the HPTs, let $\boldsymbol{Set_{HP}}$ be the set of high priority tasks. The UB1 for the high priority tasks is given by

$$UB1 = \tau - \frac{\sum_{t_i \in Set_{HP}} \min_{\forall j} ETC(i,j)}{M} \; . \tag{23}$$

The calculation for the bound assumes that each task can be executed using its minimum execution time, and that a single task can be split across multiple machines [17]. These assumptions are unrealistic and depending on consistency and heterogeneity of an HCS the bound can be fairly loose.

3.7.2 RGT Upper Bound (UB2)

The upper bound for RGT (**UB2**) uses the best result found from among the HP heuristics to set the initial machine ready times for RGT. This method is used (instead of using the result of UB1) to make the bound tighter. For similar reasons to those discussed in 3.7.1, this bound will also be loose.

Let $\boldsymbol{Set_{revenue}}$ be the set of revenue tasks. For each revenue task ($t_i \in Set_{revenue}$), the revenue per minimum execution time ($\boldsymbol{RPMET_i}$) is calculated using the following equation:

$$RPMET_i = R_i / \min_j ETC(i,j) \; . \tag{24}$$

A list ($\boldsymbol{UB_{list}}$) of the revenue tasks sorted based on $RPMET_i$ in descending order is created. To calculate UB2, we define $\boldsymbol{CT_j}$ as the completion time of machine j, and task i (t_i) as the ith task from UB_{list}. The following equations also define terms ($\boldsymbol{\tau_a}$, $\boldsymbol{\tau_b}$, in Figure 32, and the minimum execution time of t_i $\boldsymbol{MET(t_i)}$) that are used in the description of the bound:

$$\tau_a = \sum_{j=0}^{M-1} (makespan_{HP} - CT_j) \ , \tag{25}$$

$$\tau_b = (\tau_{expected} - makespan_{HP}) \cdot M \ , \tag{26}$$

$$MET(t_i) = \min_j ETC(i,j). \tag{27}$$

The $\boldsymbol{N_{min}}$ is the minimum number of revenue tasks from UB_{list} needed for the revenue to be larger than the cost, and $\boldsymbol{\tau_{total}}$ is the minimum total amount of time units that the minimum execution time machines would need to execute these tasks. That is:

$$N_{min} = \min Y : cost \leq \sum_{i=0}^{Y} R(t_i), \tag{28}$$

$$\tau_{total} = \sum_{i=0}^{N_{min}-1} [MET(t_i)] + \frac{cost - \sum_{i=0}^{(N_{min}-1)} R(t_i)}{R(t_{N_{min}})} \cdot MET(t_{N_{min}}). \tag{29}$$

Equations 28 and 29 assume that the system is oversubscribed, and that it is possible for the system to be robust (can have a revenue that exceeds cost). Based on the previous definitions, it is possible to calculate the bounds using the following conditions:

$$\rho_2 = \rho_1 \quad \text{if} \quad \tau_{total} \leq \tau_a, \tag{30}$$

$$\rho_2 = \tau_{expected} - \frac{\tau_a + \tau_b - \tau_{total}}{M} \quad \text{if} \quad \tau_a < \tau_{total} \leq \tau_a + \tau_b. \tag{31}$$

If the system is not robust (i.e., $\tau_{total} > \tau_a + \tau_b$), then

$$\rho_2 = \frac{\tau_a + \tau_b - \tau_{total}}{M} \quad \text{if} \quad \tau_{total} > \tau_a + \tau_b, \tag{32}$$

can be used to determine the minimum time past the deadline needed to make revenue equal to cost.

3.8 Results

3.8.1 Results of Consistent ETCs

The robustness results for the HPT consistent ETC matrices are shown in Figure 33, and the execution time of the heuristics is shown in Table 1. The MinCT-MinCT heuristic was the best among the greedy heuristics; its performance was within 80.9% of UB1, and within 99.9% of the best solution found by the Genitor heuristic. It is important to remember that

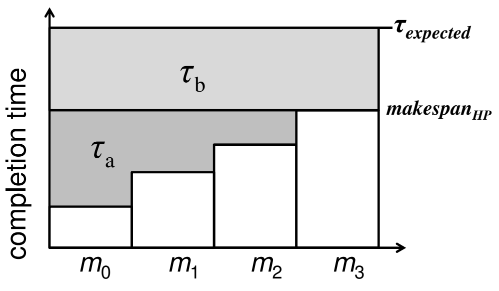


Figure 32: Illustration of τ_a and τ_b.

	heterogeneity		
heuristic	consistent	partially-consistent	inconsistent
MET, MCT, KPB	[0.02-0.03]	[0.02-0.03]	[0.02-0.03]
MinCT-MinCT	0.20	0.19	0.20
MaxCT-MinCT	0.26	0.26	0.26

Table 1: Execution times for HPT.

the Genitor was implemented only for comparison, because of its long execution time. For the parameters used in this study, it is possible to obtain a good result in a short amount of time with a greedy heuristic (0.01 seconds versus 1 hour of runtime for Genitor). The MET had the worst results; the average robustness generated by the MET heuristic had -1070.18 time units of robustness. The reason behind the poor performance of the MET heuristic is that all tasks are assigned to one machine. The KPB heuristic outperformed the MET heuristic, but did not perform as well as the MCT heuristic. Because the KPB heuristic shares characteristics with MET and MCT, it is not surprising to see that it shares the negative aspects of the MET heuristic when doing resource allocations with consistent ETC matrices.

The results for the RGT consistent ETCs are shown in Figure 34, and the execution times are shown in Table 2. The best greedy heuristic was the MaxWPTU-MaxWPTU with a robustness of 201.7 time units; this robustness was within 98% of robustness value generated by the Genitor heuristic.

The RAPTU variation of MCT outperformed all the other one-phase greedy heuristics. In terms of robustness, the robustness obtained by MCT-RAPTU was about 84.2% of the

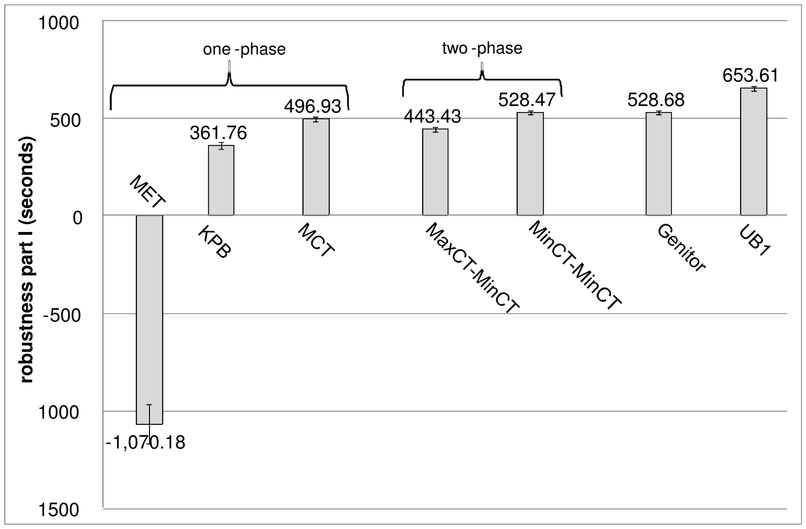


Figure 33: Consistent heterogeneity robustness for HPT, averaged over 1000 trails. The 95% confidence intervals are shown.

	heterogeneity		
heuristic	consistent	partially-consistent	inconsistent
MET, MCT, KPB	[0.02-0.03]	[0.02-0.03]	[0.02-0.03]
MinCT-MinCT	1.46	1.42	1.48
MaxW-MaxW	2.10	2.11	2.00
MaxWPTU-MinCT	2.11	2.01	2.11
MaxWPTU-MaxWPTU	3.55	3.54	3.56

Table 2: Execution times for RGT.

robustness obtained by the MaxWPTU-MaxWPTU heuristic. In terms of execution, the one-phase greedy mappings had an average of execution time between 0.02-0.03s, compared to the fastest two-phase heuristic (MaxWPTU-MaxWPTU) it was 178 times faster.

The best variation for the KPB was the RAPTU; both the RAPTU and RPTU produced mappings that were robust. However, the RAND variation did not produce robust results. For RGT, all variations of the MET heuristic have a negative robustness. The variation MET-RPTU was the best performing among the tested orderings for MET. This is an obvious result as the minimum execution time machine is the only machine that matters when considering the assignment of the MET heuristic.

The MinCT-MinCT heuristic performed better than the MET and KPB heuristics that used revenue to determine the task assignment order. However, the MinCT-MinCT did not

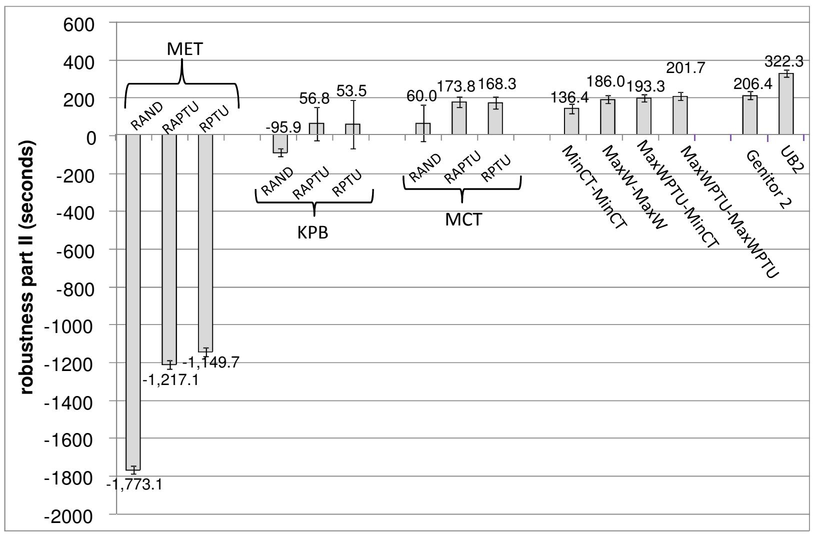


Figure 34: Consistent heterogeneity robustness for RGT, averaged over 1000 trails. The 95% confidence intervals are shown.

perform better than two of the MCT variations. Because the KPB and MET only used a small set of machines, it is better to use a greedy two-phase heuristic instead of these heuristics even though it does not have considerations for the revenue. Because the MCT heuristic distributes the load across all machines, the added revenue consideration makes it perform better than the MinCT-MinCT heuristic. The RAND variation of all one-phase greedy heuristics (MET, KPB, and MCT) performed the worst, because it does not consider the revenue when ordering tasks. However, the results from the RAND illustrate how much improvement is obtained by using the revenue-based orderings.

3.8.2 Results of Inconsistent ETCs

The results of HPT for the inconsistent matrices are shown in Figure 35, and the execution time of the heuristics is shown in Table 1. The MinCT-MinCT heuristic was the best performing greedy heuristic with an average robustness of 552.35. This result was over 99.9% of robustness value generated by the Genitor heuristic that was used for comparison.

The results of RGT for the inconsistent matrices are shown in Figure 36, and the execution time of the heuristics is shown in Table 2. The MaxWPTU-MaxWPTU was the best performing heuristic. This result was over 98.7% of the robustness value generated

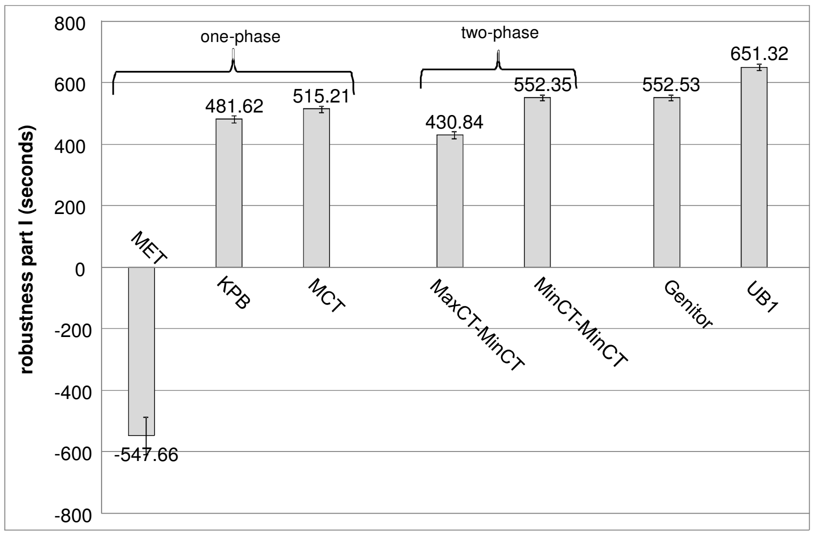


Figure 35: Inconsistent heterogeneity robustness for HPT, averaged over 1000 trails. The 95% confidence intervals are shown.

by the Genitor heuristic. Even though Genitor is seeded with the best of the heuristics and has a runtime of 60 minutes, It was interesting to observe that the MET heuristic did not perform well with inconsistent heterogeneity. In previous studies (e.g., [17]), MET performed similarly to MCT with inconsistent heterogeneity. However, in this experiment, its performance was significantly inferior to MCT's performance. Despite how poorly the MET performed in the inconsistent case, it did improve when compared to its performance in the consistent case.

By observing a sample histogram of the MET machine for each task in the inconsistent case (Figure 37(a)), it is easy to understand why the MET heuristic is not performing well. In the low-task/high-machine heterogeneity case, a small subset of machines have the minimum execution time for the majority of tasks, i.e., machines 1, 2 and 3. Thus because of the low-task/high-machine heterogeneity, many tasks share the same MET machine.

One-phase greedy heuristics were significantly faster than the two-phase heuristics. The average execution time of the one-phase heuristics was less than 0.03s, while the quickest two-phase greedy heuristic had a run time of 1.4s. Both MaxW-MaxW and MaxWPTU-MinCT had a runtime of approximately 2.11s; this shows that calculating the worth per time unit, as opposed to just the worth, does not increase the runtime significantly.

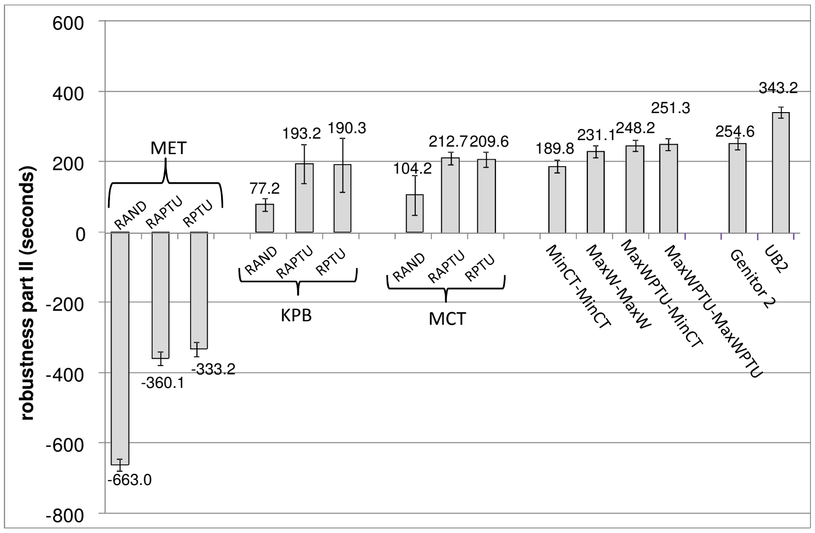
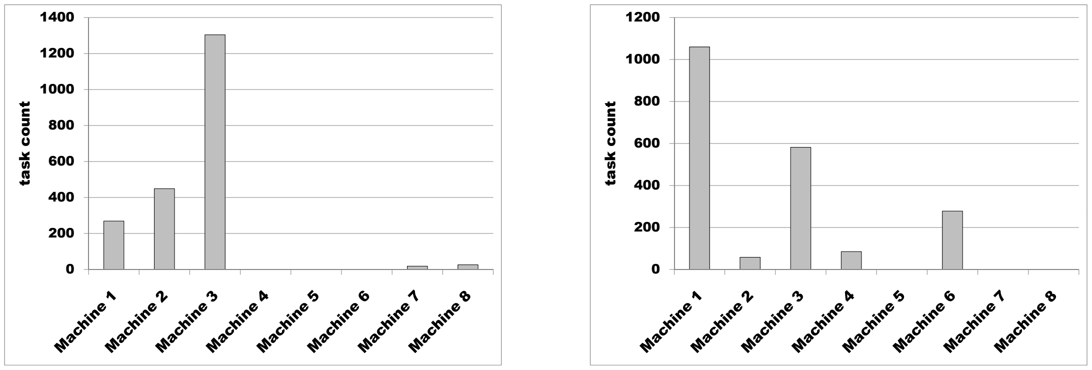


Figure 36: Inconsistent heterogeneity robustness for RGT, averaged over 1000 trails. The 95% confidence intervals are shown.



(a) Inconsistent heterogeneity.

(b) Partially consistent heterogeneity.

Figure 37: Sample histogram of number of tasks that have machine i as its minimum execution time machine.

3.8.3 Results of Partially-Consistent ETCs

The results of HPT for the partially-consistent matrices are shown in Figure 38, and the execution time of the heuristics is shown in Table 1. The best performing greedy heuristic was MinCT-MinCT, and it was over 99.9% on average of the solution Genitor was able to produce. Both the MinCT-MinCT and Genitor are very close to UB1 at about 85.8%. The

MCT and KPB heuristic both had good performance and overlapping confidence intervals; both were approximately 79.7% of UB1.

By observing a sample histogram of the minimum execution time machine (Figure 37(b)) for the partially consistent heterogeneity case, it easy to understand why the MET heuristic is performing better than it did in the inconsistent case. The partially consistent case has a better distribution of the minimum execution time machine than the inconsistent case, because of the algorithm, used to create the partially consistent matrix [3], creates a consistent sub-matrix (composed of half the tasks and half the machines). This allows the minimum execution time machine to be spread across more machines, thus marginally improving the results. This explains why the MET did better in the partially consistent heterogeneity than in the inconsistent heterogeneity.

It is interesting to observe that, in general, the results are closer to UB1 in the partially consistent case than in the inconsistent case. This can be explained by considering the method used to calculate UB1, i.e., it assumes all machines to be the MET machine. The reason why this bound is tighter for the partially consistent case is the same reason why the MET results are better for the partially consistent case.

The results of RGT for the partially-consistent matrices are shown in Figure 39, and the execution time of the heuristics is shown in Figure 2. The best performing greedy heuristic for RGT was the MaxWPTU-MaxWPTU with a robustness of 246.00 time units, which was 73.5% of UB2. The RGT Genitor gave an average improvement of 1.6% time units, and had overlapping confidence intervals with MaxWPTU-MaxWPTU. Among the one-phase heuristics MCT-RAPTU had the best performance with a robustness of 208.8, which was about 84.9% of the performance obtained by the best performing greedy heuristic (MaxWPTU-MaxWPTU).

3.9 Conclusions

The environment considered for this study is a computer system that processes weather data. Additionally, this research could also apply to different types of satellite image processing, e.g., homeland security. The satellite data system has two parts that need to be robust

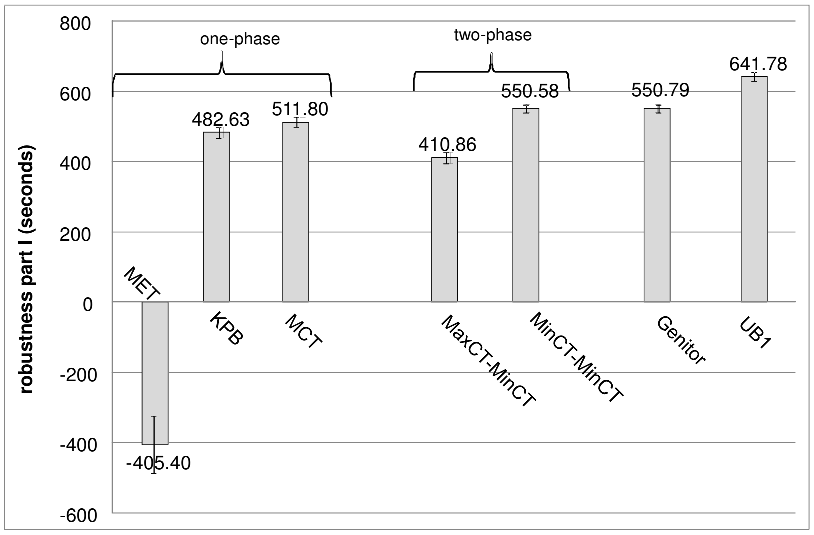


Figure 38: Partially-consistent heterogeneity robustness for HPT, averaged over 1000 trails. The 95% confidence intervals are shown.

against uncertainty. The first part is the execution of critical tasks to ensure the operation of the satellite, and the second part ensures financial viability of the system. The uncertainty considered for this system is the variability in the time interval between the arrival of successive data sets.

Several heuristics and their variations were implemented for each of the parts. For all the evaluated scenarios, the best HPT heuristic was MinCT-MinCT, and MaxWPTU-MaxWPTU was either the best for RGT or had comparable performance. Therefore, these are recommended as the resource allocation heuristics for this environment. In all scenarios, MinCT-MinCT was within 99.9% of Genitor for HPT, and MaxWPTU-MaxWPTU performed on average about 98% of Genitor for RGT. This is a good result considering that the Genitor was seeded with the best greedy heuristic, and executes for an hour instead of seconds. The MCT-RAPTU is the best one-phase heuristic for RGT robustness, across all consistencies. For a few test cases, the Genitor was allowed to run for six hours with only a 1% additional improvement (over the Genitor with a one hour run time). This indicates that very little improvement was observed from letting the Genitor heuristic execute for a longer time.

An example of possible extensions could be: tasks execution times could be vary based

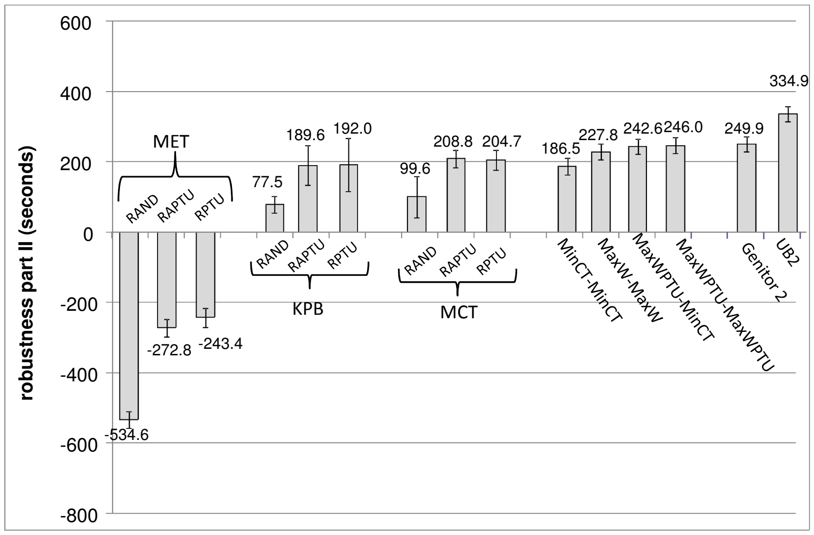


Figure 39: Partially-consistent heterogeneity robustness for RGT, averaged over 1000 trails. The 95% confidence intervals are shown.

on the input data sets, possible machine failures, representing the entries in an ETC as a probability mass function, use of different revenue models, computing platforms with different heterogeneities, variation in the amount of tasks that need to be executed, and applications to other important sensor problems. Another extension to this work is to incorporate good will or fairness when assigning resources. This fairness could become part of the robustness metric or a Quality-of-Service constraint. Additionally, the same model could be used to simulate a multicore or multi-threaded environment. However, to do this accurately, we would need to incorporate memory hierarchy, and sharing. This could also be an area for future research.

CHAPTER IV

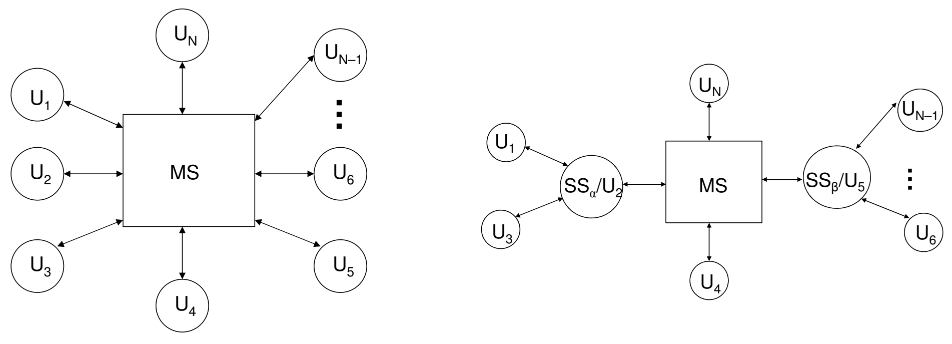
RESOURCE ALLOCATION IN A CLIENT/SERVER MASSIVE MULTI-PLAYER ONLINE GAMING ENVIRONMENT

4.1 Introduction

The environment considered in this research is a massive multiplayer online gaming (MMOG) environment. In an MMOG environment, each user controls an *avatar* (an image that represents and is manipulated by a user) in a virtual world and interacts with other users. An important aspect of an MMOG environment is maintaining fairness among users (i.e., not giving an unfair advantage to users with faster connections or more powerful computers). The experience (positive or negative) the user has with the MMOG environment is dependent on how quickly the game world responds to the user's actions. There are various methods that can be used to quantify the responsiveness of the game world to a user's action. One possible method is to minimize the maximum response time (time needed to send an action to the server and receive the result of that action from the server) among all users; another option is to reduce the differences between the response times so that the system is "fair."

In general, most MMOG environments use a client/server architecture to control the virtual game world. The client/server architecture has some disadvantages: the initial procurement of servers is expensive, server administration is required, customer service is necessary, and the architecture is hard to scale based on demand. In addition to the initial development cost, other factors such as the popularity of a game, and unexpected technical problems during and after the launch, also can affect the final cost and success of the game [73].

A preliminary version of portions of this research appeared in [21, 22].



(a) (b)

Figure 40: (a) Client/server architecture, using a single server to do processing; and (b) secondary server architecture, using users' computers to assist the Main Server in processing.

The environment we are interested in is massive on-line first-person shooters. These types of games usually involve a large group of people (e.g., 256 people in Massive Action Game (MAG) developed by Zipper Interactive) competing in a virtual world attempting to complete specific goals . A problem may occur when considering interaction with other users. For example, consider a war game where two users are shooting at each other. One way of determining the winner of this contest is to determine who shot first. However, determining who shot first in the game world can be difficult. It is possible for the game to process these users' actions in the incorrect order.

This study focuses on using secondary servers to modify the system based on demand. Consider an environment where there is a main server ($\underline{MS}$) that controls the state of the virtual world, and each user ($\underline{N}$ is the total number of users) produces a data packet that needs to be processed by the MS. If the performance falls below acceptable standards, the MS can off-load calculations to secondary servers ($\underline{SS}$s). An SS is a user's computer that is converted into a server to avoid degradation in the performance of the MMOG environment (see Figure 40). The purpose of using the users' computers as SSs is to create a distributed ad-hoc system that will keep the response times low and fair.

The allocation of users as SSs has similar security requirements as distributed servers and peer-to-peer based MMOG systems. These issues are studied in [12, 45] and will not be discussed here because we consider it to be a separate research problem.

The introduction of SSs causes the game-state to be handled differently than with a single MS. Each SS handles conflicts among the players attached to it, and sends conflict-free information to the MS. However, this information may conflict with information from another SS. If there is a conflict between SSs then it will be resolved by the MS.

This study assumes all players are willing to become SSs. Our approach could easily be adapted to account for having a subset of players who are not willing to be an SS, i.e., we can have a list of players eligible to become SSs.

A session in the MMOG environment is assumed to last for an extended period of time, with a small break between sessions [54]. These assumptions make a static resource allocation heuristic viable [2]. This study evaluates the MMOG environment with two different optimization criterion. The first criteria is the minimization of the response time, while the second is the maximization of robustness. In both cases, the heuristics determine the number of SSs, which users are converted to SSs, and how users are distributed among the SSs and the MS. The assignment of users to SSs and the MS is related to the assignment of tasks to machines (e.g., [11,17,75,86]) with the SSs and the MS as machines and the users as tasks.

This paper is separated into two parts; for the first part (in Section 4.3), we propose heuristics that minimize the time it takes to process the game-world update requests from all users. A mathematical lower bound is derived to evaluate the performance of the heuristics for minimization of response time in this MMOG. For the second part (in Section 4.4), we consider that the number of new players joining an on-going game is unknown, and we derivate a “robustness” metric to make the resource allocation resilient against this uncertainty. The goal of the heuristics is to provide a “fair” environment for all the users, and to be “robust” against this uncertainty. The difference between the first part and the second part is that the second part uses a metric based on fairness to quantify the quality of a resource allocation.

The contributions of this paper are: (a) mathematically modeling an MMOG environment, (b) designing heuristics to minimize the response time, (c) studying and simulating an MMOG environment where an unpredictable number of players may want to join an ongoing session, (d) creating parameters to quantify the robustness of a system against the uncertainty of the number of players that will try to join an ongoing session, and (e) deriving resource allocation heuristics that maximize the number of players that can join an existing game session while still maintaining a fair system.

This paper is organized as follows. Section 4.2 provides the common environment. In Section 4.3, we analyze the MMOG environment when the optimization criteria is to minimize the maximum response time. This section includes the heuristics used to do resource allocation, the bound on performance, and the results for this simulation setup. Section 4.4 focuses on the proposed heuristics for maximizing the robustness, the upper bound on the number of new players that can join the game, and results for this section. We provide the related work in Section 4.5, and in Section 4.6 we present our conclusions.

4.2 Environment

4.2.1 Overview

For both problem domains considered in this study, the aspect of the MMOG environment we can control is (a) which users are converted into secondary servers, and (b) how users are assigned to the MS and SSs. In the client/server solution shown in Figure 40(a), all users connect to the MS, therefore the MS is the only machine performing computation. In the SS solution shown in Figure 40(b), the MS and SSs perform computation and the MS resolves conflicts among users and SSs connected to it.

The time it takes the system to respond to a user's request (latency) is very important [7]. The communication time between different pairs of nodes (user computer, SS, or MS) will vary. To simplify the calculation of a server's response time to a user, the following assumptions are made about the communication model in this system. The communication times among the users, SSs, and the MS do not change during a session. These times are independent of the number of users connected to an SS or the MS. These assumptions are used to reduce the complexity of the simulations.

4.2.2 Computational Model for Main Server and Secondary Servers

To simplify the simulation study, the level of activity in the MMOG environment of all the users is considered identical (i.e., the frequency of interaction with the MMOG environment is the same for all players). Thus, the computational load is based on the number of users (i.e., they have the same computational needs). To model the computation times of the MS and SSs we need to consider how the computation time increases with the increase in the number of users. In [43], latency in an MMOG environment shows a "weak exponential" increase with an increase in players; we approximate this by using a constant communication time and a quadratic factor for the computation.

Let $\underline{n_\alpha}$ be the number of users connected to secondary server $\underline{\alpha}$ ($\underline{SS_\alpha}$), and $\underline{\mu_\alpha}$ be a computational constant for $\underline{SS_\alpha}$ that represents the heterogeneity in the computing power of the users' computers (each user has a different constant). The computation time for an SS_α ($\underline{Comp_\alpha}$) can be modeled as:

$$Comp_\alpha = \mu_\alpha \cdot (n_\alpha)^2. \tag{33}$$

Let $\underline{n_{secondary}}$ be the total number of users connected to all the SSs, $\underline{n_{nss}}$ be the number of SSs, $\underline{n_{main}}$ be the number of users connected to the MS, and $\underline{b}$ and $\underline{c}$ be computational constants of the MS. The computation time of the MS ($\underline{Comp_{MS}}$) is:

$$Comp_{MS} = c \cdot n_{secondary} + b \cdot (n_{main} + n_{nss})^2. \tag{34}$$

We assume that the state of the game world is updated every $Comp_{MS}$ time units.

4.2.3 Objective Functions RT_{max} and RT_{min}

Let $\underline{RT_x}$ represent the R̲esponse T̲ime (R̲T̲) of a packet (representing an action in the game world) sent by the computer of user x ($\underline{U_x}$) to the MS (possibly through an SS) and returning to U_x with the corresponding consequence of that action in the game world. Let $\underline{Comm(A,B)}$ be the communication time between node A and node B, and $\underline{\Delta}$ is the time packet has to wait before being processed. The equation used to calculate RT_x if U_x is connected directly to the MS is:

$$RT_x = Comm(U_x, MS) + Comp_{MS} + Comm(MS, U_x) + \Delta. \tag{35}$$

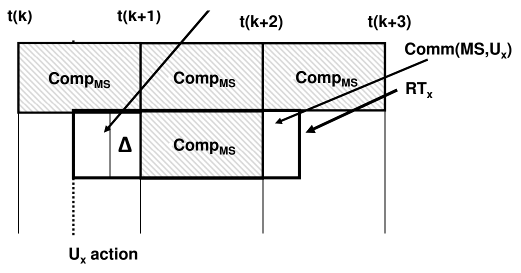


Figure 41: This figure illustrates the return time for a user x (U_x) connected directly to the MS.

A graphical representation of this equation is shown in Figure 41. If a user is connected to an SS_α then the equation is:

$$\begin{aligned} RT_x &= Comm(U_x, SS_\alpha) + Comp_\alpha + Comm(SS_\alpha, MS) + Comp_{MS} \\ &+ Comm(MS, SS_\alpha) + Comm(SS_\alpha, U_x) + \Delta. \end{aligned} \tag{36}$$

A graphical representation of this equation is shown in Figure 42. If U_x is SS_α then Equation 36 is used with $Comm(U_x, SS_\alpha) = Comm(SS_\alpha, U_x) = 0$. To calculate the maximum response time (i.e., $\underline{RT_{max}}$) we use:

$$RT_{max} = \max_{\forall U_x} (RT_x), \tag{37}$$

with $\Delta = Comp_{MS}$. As shown in Figure 43, this time represents the maximum time any user will have to wait for a response from the MS. Let $\underline{RT_{min}}$ represent the fastest any user can interact with the MMOG environment, and it is calculated as follows:

$$RT_{min} = \min_{\forall U_x} (RT_x), \tag{38}$$

with $\Delta = 0$ (shown in Figure 44).

4.2.4 Heuristic Requirements

All heuristics were limited to a maximum execution time of 10 minutes. We assume that the players wait in a game lobby while the game fills up, therefore the resource allocation can be done while the players are in the lobby. In this context, resource allocation implies assigning a user in one of three ways: (1) attaching it directly to the MS without making

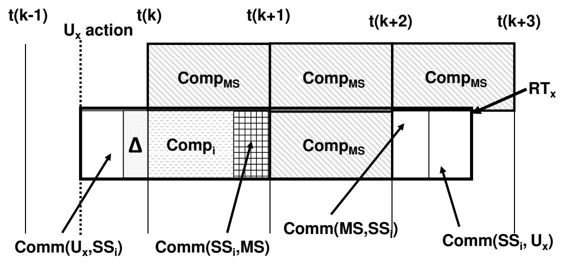


Figure 42: This figure illustrates the return time for a user x (U_x) connected to the MS through an SS.

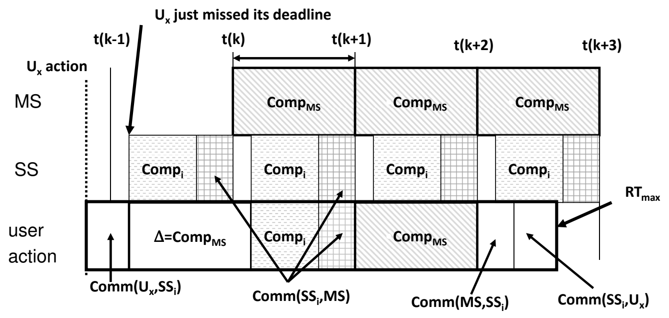


Figure 43: This figure illustrates RT_{max} when the user with just *misses* the deadline for sending computation at the SS, i.e., $\Delta = Comp_{MS}$.

it an SS (although it can become one), (2) attaching it to the MS and making it an SS, or (3) attaching it to an existing SS. An unassigned user is one that has not been assigned yet, and directly connected users ($\underline{DCU}$s) are users that are connected directly to the MS.

4.3 Response Time Minimization

4.3.1 Problem Statement

The purpose of this section is to maintain an acceptable system performance (despite the MS used to maintain the MMOG environment being oversubscribed) without increasing the processing power of the MS. The proposed solution is to convert users to SSs that assist the MS in computation. In the client/server solution shown in Figure 40(a), all users connect to the MS, therefore the MS is the only machine performing computation. In the

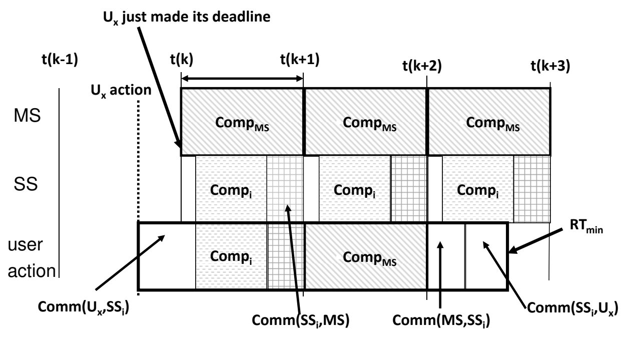


Figure 44: This figure illustrates RT_{min} when the user with just *makes* the deadline for sending computation at the SS, i.e., $\Delta = 0$.

(1) Given a predetermined set of DCUs, all users that are not in the set of DCUs are marked as unassigned.

(2) For each unassigned user, the DCU that gives the minimum RT is determined (first minimum).

(3) The best paired user/server (i.e., with smallest RT) among all the pairs generated in (2) is selected (second minimum).

(4) The user in the best pair selected in (3) is then assigned to its paired server.

(5) Steps (2) through (4) are repeated until all tasks are assigned.

Figure 45: Procedure for using Min-Min RT to generate a resource allocation.

SS solution shown in Figure 40(b), the MS and SSs perform computation and the MS resolves conflicts among users and SSs connected to it. The objective, of this section, is to minimize RT_{max} (i.e., the resource allocation that produces the smallest RT_{max} is the best).

4.3.2 Heuristics for Response Time Minimization

4.3.2.1 Min-Min RT

The Min-Min RT sub-heuristic, used by the other heuristics, is based on the concept of Min-Min heuristic [42]. The Min-Min heuristic is widely used in the area of resource allocation (e.g., [20,31,39,42,44,47,51,86]). Min-Min RT requires a set of DCUs to generate a resource allocation; therefore, it is not an independent heuristic. The procedure to implement the Min-Min RT is shown in Figure 45.

4.3.2.2 Min-Min SS

The Min-Min SS heuristic is similar to the Min-Min RT heuristic. The difference is that the Min-Min SS does not require an initial set of DCUs. The heuristic will determine the set of SSs by allowing users to consider connecting to the MS in addition to DCUs in step (2) of Figure 45.

4.3.2.3 Iterative Minimization

In Iterative Minimization (IM), a resource allocation is represented in the form of an vector whose i^{th} element indicates the way user i is connected to the MS, either directly or the SS to which it is connected. A potential host (PH) is a user that is not connected to the MS through an SS; i.e., either it is a DCU or is unassigned at this point. This heuristic considers assigning an unassigned user to all PHs or the MS and picks the PH or MS that provides the minimum RT_x. If a PH provides the minimum RT_x and is not already an SS then it is converted to one. The pseudo-code is shown in Figure 46. An iterative minimization procedure is run on the resource allocation from the IM heuristic. This iterative minimization attempts to move the user with RT_{max} to a different secondary servers or to the MS to find a better resource allocation. The procedure for this iterative minimization is shown in Figure 47.

4.3.2.4 Tabu Search

Tabu Search concept [41] enhances the performance of a global search method by storing the previously visited areas in the search space using a tabu list so they are not revisited. To make the size of the tabu list reasonable, only the last n (set empirically to 20) visited neighborhoods are saved [41]. The resource allocation is represented by vectors as in the IM heuristic.

Local moves (or short hops) explore the neighborhood of the current resource allocation, searching for the local minimum. All the moves that we use in the Tabu Search are considered greedy in the sense that we accept a resource allocation if it has a smaller RT_{max} (better objective function value); however, applying greedy moves may cause the Tabu Search to

```
(1) Mark all users as unassigned.
(2) For each unassigned user ($\underline{u}$) in a fixed arbitrary order.
    (a) Define $\underline{minRT}$ as the $RT$ if $u$ is connected directly to the $MS$.
    (b) Among all $PH$s, find the $PH$ that minimizes $RT$ of $u$ connected to the
        $MS$ through $PH$ ($\underline{RT_{u \rightarrow PH \rightarrow MS}}$) .
        (i) If $RT_{u \rightarrow PH \rightarrow MS}$ is less than $minRT$ then attach $u$ to $PH$, and convert
            $PH$ to an $SS$ if it is not already one. Because of heterogeneity in
            computation and communication the $RT_u$ might be smaller through a
            $PH$ instead of the $MS$.
        (ii) Else, attach $u$ directly to the $MS$.
    (c) Mark user $u$ as assigned.
(3) Output final resource allocation.
```

Figure 46: Pseudo-code for generating the initial resource allocation of the Iterative Minimization heuristic.

```
(1) $RT_{best}$ is equal to the $RT_{max}$ value of the resource allocation generated in Figure
    46.
(2) For each user ($\underline{U_x}$) connected to an $SS$
    (a) Connect $U_x$ to the $MS$ and connect the user with $RT_{max}$ to $U_x$.
    (b) Find the $RT_{max}$ of this configuration.
    (c) If $RT_{max} < RT_{best}$ then save this resource allocation as the best.
(3) For each user ($U_x$) connected to an $SS$
    (a) Swap $U_x$ with the user with $RT_{max}$.
    (b) Find the $RT_{max}$ of this configuration.
    (c) If $RT_{max} < RT_{best}$ then save this resource allocation as the best, otherwise
        undo the swap.
(4) For each $SS$ ($\underline{SS_x}$)
    (a) Connect the user with $RT_{max}$ to $SS_x$.
    (b) Find the $RT_{max}$ of this configuration.
    (c) If $RT_{max} < RT_{best}$ then save this resource allocation as the best, otherwise
        undo the change.
(5) Output best resource allocation.
```

Figure 47: Procedure for using the Iterative Minimization heuristic to generate a resource allocation.

```
(1) Create the tabu list of size $\underline{size_{tabu}}$.
(2) While the execution time is less than 10 minutes
    (a) Generate a random set of $DCU$s ($\underline{rand_{DCUs}}$). If this random set is not in
        the tabu list then continue to step (b), otherwise repeat step (a).
    (b) Use the Min-Min RT heuristic with $rand_{DCUs}$ to generate a full mapping.
    (c) Use the short hop procedure.
    (d) Update the tabu list by adding the set of $DCU$s from step (c) and removing
        the oldest set of $DCU$s.
```

Figure 48: Procedure for using the Tabu Search heuristic to generate a resource allocation.

reach a local minimum that it cannot escape. The global move (or long hop) is used to escape local minima by producing a random resource allocation with a new set of SSs that is not in the tabu list. Intuitively the short hops try to find better resource allocations within the same neighborhood (same set of DCUs) by moving the user with RT_{max} to other SSs, or by reducing the computation of the SS where the user with RT_{max} is connected. The sum of both long hops and short hops were limited to a 10 minute execution time limit. The number of short hops allowed per long hop was limited to a maximum$\underline{MAX_{SHORTHOPS}}$ hops (determined empirically to be 100), or 10 short hops without improvement. The procedure for Tabu Search is shown in Figure 48 and the procedure for the short hops is shown in Figure 49.

In our experiments, the Tabu Search heuristic was seeded with the results from other heuristics. This was accomplished by replacing the first long hop (first long hop is generated in steps 2(a) and 2(b) in Figure 48) with the solution from the *seed* heuristic.

4.3.2.5 Discrete Particle Swarm Optimization

Discrete Particle Swarm Optimization (DPSO) is based on the particle swarm optimization in [48]. The authors in [70] implemented a discrete version of the particle swarm optimization in [48], upon which we base our implementation. Intuitively, this algorithm samples the search space of possible SS configurations, and then uses the Min-Min RT algorithm to generate a complete mapping from a set of SSs.

In DPSO, the position of a particle represents a solution (resource allocation). Each

```
(1) Set $short_{hops}$ to 0 and set $MAX_{SHORT\ HOPS}$ to the maximum allowed short hops.
(2) Given the resource allocation found in the long hop, we determine $RT_{max}$ of this resource allocation.
(3) While $short_{hops} < MAX_{SHORT\ HOPS}$
    (a) Find the user with $RT_{max}$ ($U_{max}$).
    (b) For each server $s$ ($DCU$ or $MS$), reconnect $U_{max}$ to $s$,
        and if the move decreases $RT_{max}$ then
          (i) accept the move
         (ii) update $RT_{max}$
        (iii) go to step (c)
    (c) Increase $short_{hops}$ by one.
    (d) Find the $SS$ that has the user with $RT_{max}$.
    (e) Select a random user that is connected to this $SS$ ($U_{random}$).
    (f) For each $DCU$ $s$ , reconnect $U_{random}$ to $s$,
        and if the move decreases $RT_{max}$ then
          (i) accept the move,
         (ii) update $RT_{max}$
        (iii) go to step (g).
    (g) Increase $short_{hops}$ by one.
```

Figure 49: Procedure for using the short hops to improve a resource allocation.

particle is composed of N entries (each entry represents a user). Let $X_{ij} \in \{0, 1\}$ represent whether user j is a DCU ($X_{ij} = 0$), or a non-DCU ($X_{ij} = 1$) in particle i. Particles move around through different possible solutions based on how their velocity is composed. The direction of the velocity will determine whether user j changes to a DCU or a non-DCU. Let V_{min} represent the minimum and V_{max} represent the maximum allowed velocity for a particle. A particle i will have a velocity in each direction j ($V_{ij} \in [V_{min}, V_{max}]$). A sigmoid function is used to probabilistically convert the real value of V_{ij} into a position of either 0 or 1 for X_{ij}. A coefficient ($w \leq 1$) is used to slow the current velocity of the particle over time. A particle is allowed to "move" for a pre-determined number of iterations ($iter_{max}$)—determined based on the maximum allowed execution time.

Each particle i will keep a record of its best solution (P^i), where each P^i has an entry for

```
(1) Initialize an array of $P$ particles by $N$ dimensions randomly with 0 or 1 (a value
    of 0 indicates a user is not a $DCU$ and 1 indicates the user is a $DCU$).
(2) Determine $RT_{max}$ using the Min-Min RT heuristic for each particle.
(3) Initialize the global and each particles best positions.
(4) For $i = 1$ to number of particles do
    (a) For $j = 1$ to number of dimensions do
          (i) $R_1 = U(0,1)$
         (ii) $R_2 = U(0,1)$
        (iii) $R_3 = U(0,1)$
         (iv) $V_{ij} = w \cdot V_{ij} + p_w \cdot R_1 \cdot (P^i_j - X_{ij}) + g_w \cdot R_2 \cdot (G_j - X_{ij})$
          (v) If $(V_{ij} < V_{min})$ then $V_{ij} \leftarrow V_{min}$.
         (vi) If $(V_{ij} > V_{max})$ then $V_{ij} \leftarrow V_{max}$.
        (vii) If $(R_3 < Sigmoid(V_{ij}))$ then $X_{ij} = 1$, else $X_{ij} = 0$.
    (b) Determine $RT_{max}$ using the Min-Min RT algorithm to generate a complete
        mapping.
(5) Set each particle's best position ($P^i$) using the particles in (4),and set the best
    global position to the best $P^i$ over all $i$.
(6) Repeat (4) and (5) until the number of iterations is equal to $iter_{max}$.
```

Figure 50: Procedure for using the DPSO heuristic to generate a resource allocation.

each user j ($\underline{P^i_j} \in \{0,1\}$). The particle i will be attracted back to P^i with a given personal weighting coefficient ($\underline{p_w}$). This coefficient will attract this particle to explore areas of the search space close to P^i.

The system as a whole will keep a best global solution ($\underline{G}$). This best global solution has an entry for each user j ($\underline{G_j} \in \{0,1\}$). All the particles in the system are attracted to the best global solution. The force of the attraction is determined by a global weighting coefficient ($\underline{g_w}$). The coefficient promotes the exploration around the best known solution. The values of the coefficients w, g_w, and p_w were selected by experimentation to optimize the performance. The pseudo-code for our implementation of DPSO is shown in Figure 50.

In our experiments, the DPSO heuristic was seeded with the results from other heuristics. This was accomplished by creating a particle that has the same set of SSs as the resource allocation generated by the *seed* heuristic. The set of SSs determined by the seed is evaluated using the Min-Min RT heuristic, and the associated RT_{max} is determined.

1) An initial population of 200 chromosomes is generated and evaluated.

2) While there are less than 1000 iterations without improvement or 10 minutes have not elapsed.

    a) A pair of parents is selected using roulette wheel selection.
    b) Two offspring are generated using two-point crossover.
    c) For each offspring there is a 3% probability of mutating each field in the chromosome.
    d) The offspring are evaluated and inserted into the ordered population displacing the worst chromosomes.

3) The output is the best resource allocation.

Figure 51: Procedure for using the Genitor RT to generate a resource allocation.

4.3.2.6 Genitor RT

The Genitor RT heuristic is based on the Genitor heuristic [85]. Genitor is a steady state heuristic that only does one crossover and mutation operation per iteration. The results of the crossover and mutation are evaluated and inserted in the ordered population based on their rank. The heuristic uses the ranked population to keep the best chromosomes in the population (of size 200 determined empirically).

This heuristic uses a chromosome that represents a full mapping. A chromosome is a vector of size N where the value of the i^{th} entry represents where the i^{th} user is connected. The value j in this entry indicates that user i connected to the MS through user j (if $1 \leq j \leq N$), and directly to the MS (if $j = 0$). While this a clear and natural representation, the disadvantage of this representation is that the crossover and mutation operations can cause invalid resource allocations that need to be fixed. The complete procedure for the Genitor RT heuristic is shown in Figure 51.

The first operator is crossover; for the crossover, we randomly select two points (from 1 to N) in the two parent strings and exchange the entries of the parents between these two points. If the crossover causes a user x to be mapped to another user that is no longer an SS, then user x is assigned to an existing SS that gives it the smallest RT_x time. The procedure for the crossover is shown in Figure 52.

(1) Select two parents for crossover (parent 1 and parent 2) using a linear bias function.

(2) Generate two random numbers between 1 and N ($\underline{R_1}$ and $\underline{R_2}$ with $R_1 < R_2$)

(3) The entries between R_1 and R_2 in parent 1 are exchanged with the value the entries have in parent 2 generating a child.

(4) For each entry (i.e., user assignment) in the child:

  (a) Check if the entry has a valid assignment.

  (b) If the entry has an invalid assignment (e.g., assigned to a user that is not an SS) then assign it to the server (MS or an SS) that gives the user the minimum RT_x.

(5) The entries between R_1 and R_2 in parent 2 are exchanged with the value the entries have in parent 1 generating a child, and repeat step (4).

Figure 52: Procedure for using crossover to generate new resource allocations.

The second operator is mutation; for the mutation, we determine with a fixed probability if the assignment of a user is mutated. The mutation is done by randomly selecting if a user should be connected directly to the MS, an SS, or user i (not an SS). If the user is an SS assigned to a user i, then reassign it to user i and reassign the players connected to it to other existing SSs (selected randomly). If as a result of the mutation the user is connected directly to the MS or is an SS then no further repairs need to be made to the assignment; however, if the randomly selected user is connected to another user i then i needs to be converted into an SS. The procedure for mutation is shown in Figure 53.

In our experiments, the Genitor RT heuristic was seeded with the results from other heuristics. This was accomplished by using the resource allocation generated by the *seed* heuristic as a chromosome in the population.

4.3.3 Lower Bound

The primary purpose of deriving a mathematical lower bound was to evaluate the experimental results of our proposed heuristics for the minimization of RT_{max}. The bound has two components that can be calculated independently. The first component finds the minimum possible *computation time* of the MS and SSs (by performing an exhaustive search

(1) Set k to 1.

(2) Based on a fixed probability, determine if the k^{th} entry in the chromosome is mutated.

(3) If the entry is mutated, then:

  (a) Generate a random assignment (connected to the MS, SS, or user i).

  (b) If the entry being modified is an SS, then reassign the players assigned to this SS to existing SSs (selected randomly).

  (c) If this is an assignment to a user that is not an SS convert that user to an SS.

(4) Increase k by 1.

(5) If $k \leq N$ then go to (2).

Figure 53: Procedure for using mutation to change a resource allocation.

of all possible *computation times*). This component has two simplifying assumptions that are consistent with generating a lower bounds: (a) all users have the same computational constant ($\mu_{min} = \min_{\forall U_x} \mu_x$), and (b) users connected to SSs are evenly distributed among SSs. Component (a) removes the heterogeneity in computing power of the SSs, and (b) minimizes the maximum computation time among SSs. Given the assumptions above, we set $\Delta = Comp_{MS}$, and Equations 33, 34, and 36 we can calculate the computation f_{comp} with the following equation:

$$f_{comp}(n, n_{n_s s}) = Comp_\alpha + 2 \cdot Comp_{MS} \;\; with \;\; \Delta = Comp_{MS} \tag{39}$$

$$f_{comp}(n, n_{nss}) = \mu_{min} \cdot \left\lceil \frac{N-n}{n_{nss}} \right\rceil^2 + 2 \cdot \left[c \cdot (N-n) + b \cdot n^2 \right] \; ; \tag{40}$$

The second component is the lower bound on the communication time. This bound is calculated by finding the minimum time each user requires to connect to the MS (either connected directly to the MS or through another user), and then finding the minimum among these times.

Let $\underline{n = n_{ss} + n_{main}}$ be the total number of users that are connected to the MS. Based on Equation 33, and assuming user x connects to the MS through user y gives user x's communication time f_{comm}. The communication time f_{comm} is calculated as follows:

$$f_{comm}(U_x, U_y) = Comm(U_x, U_y) + Comm(U_y, U_x) + Comm(U_y, MS) + Comm(MS, U_y)\ . \quad (41)$$

The case where $U_x = U_y$ is considered to account for the case when U_x is connected to the MS, i.e., $Comm(U_x, U_x) = 0$. The lower bound (LB) on RT_{max} is given as:

$$LB = \min_{1 \leq n \leq N} \left(\min_{0 \leq n_{nss} \leq n} \left(f_{comp}(n, n_{nss}) \right) \right) + \min_{U_x \in all\ users} \left(\min_{U_y \in all\ users} \left(f_{comm}(U_x, U_y) \right) \right)\ . \quad (42)$$

Proof. The proof will be divided into two parts. The first part will be to prove that the computational bound is minimum and the second part will be to prove the communication minimum.

The first part of the bound does an exhaustive evaluation of all possible configurations for n_{nss} and n. This will give us all the possible computations times. It will move n from 1 (only one user connected to the MS) to N (all users connected to the MS). For each of these values of n it will attempt all possible configurations of $n_{nss} \leq n$. It is important to note that $n_{nss} = 0$ is an invalid configuration unless $n = N$ (i.e., the only scenario where we do not have SSs is when all users are connected directly to the MS), and in this case we consider $(N - N)/(0) = 0$. Because we are considering all the possible configurations it is not possible to get a smaller computation time.

The second part of the bound finds the smallest communication time for each user, then it finds the minimum among these times. This method does an exhaustive search of the possible communication times (through an SS or directly connected to the MS). Therefore, this is a user with this minimum communication time. To this user's communication time we add the smallest possible computation time to get a lower bound on RT_{max}.

□

4.3.4 Simulation Results

4.3.4.1 Simulation Setup

The simulation had 200 users interacting in the MMOG environment. The constants for these simulations were $b = 0.03$ and $c = 0.01$ (the values for these constants were set to approximate realistic values for latencies in an MMOG environment). The communication times between nodes were allowed to vary from 0 to 40 ms with a uniform distribution. The computational constant (μ_α) at each user node was allowed to vary between 0.5 and 1 with a uniform distribution. For this study, 100 scenarios were created with varying communication times and μ_α for each user. For the purpose of comparison, each heuristic was limited to a maximum execution time of 10 minutes per scenario.

4.3.4.2 Results for Minimization of Response Times

Figure 54 shows the results averaged over the 100 scenarios. We can observe that the DPSO had the best performance in all cases (unseeded, seeded with Min-Min SS, and seeded with IM). When we seeded the heuristics with the Min-Min SS seed, the DPSO with the Min-Min SS seed was the best overall performing variation. Additionally all heuristics were able to improve upon the allocation done by the Min-Min SS seed. For the heuristics seeded with the IM heuristic, we can see that the DPSO did not perform as well as the DPSO seeded with the Min-Min SS. Genitor and Tabu were able to improve on the result of the IM heuristic. The performance of Tabu Search when it was seeded with IM was comparable to its performance when seeded with Min-Min SS. The Genitor did have better performance with the Min-Min SS seed (as opposed to the IM seed).

The LB was calculated to compare the performance of the heuristics to a mathematical bound on performance. The lower bound is about 44.7 time units less than the best performing heuristic (DPSO with Min-Min seed).

If all users were connected to the MS then the RT_{max} would be approximately 1200 time units (i.e., $200^2 \cdot b + \max_{\forall U_x} 2 \cdot Comm(U_x, MS) \approx 1200$). The use of the secondary server based approach in our simulations leads to an improvement of an order of magnitude (i.e., 110 time units versus 1200 time units).

4.4 Robustness to Additional Players Joining the Game

4.4.1 Problem Statement

The purpose of this section is to determine an allocation that will allow the maximum number of new players to join an on-going game, i.e., be robust to additional players. The concept of robustness is described in detail in Section 4.4.2.

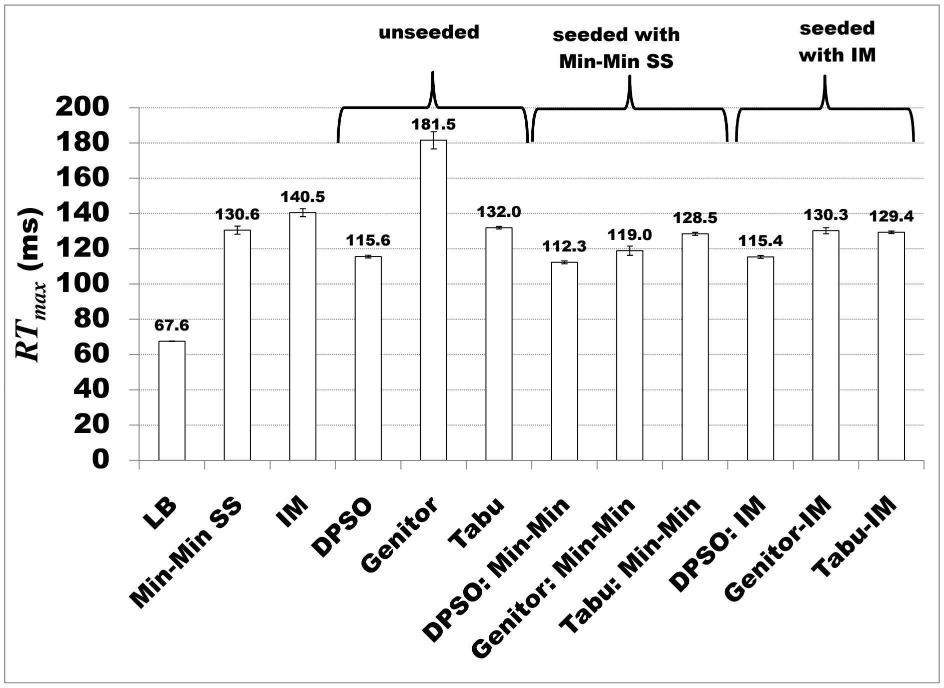


Figure 54: Results for response time minimization. The computational parameters of the MS were set to: $b = 0.03$ and $c = 0.01$, values are averaged over 100 scenarios, and the error bars show the 95% confidence intervals.

The goal of the heuristics presented in this section is to provide an environment where the differences in latency among all users are bounded by a quality of service (QoS) constraint. This QoS constraint is based on human perception (i.e., the difference in response times between players is imperceptible). If the QoS is met then the environment provides a high-quality interactive experience. New players are users that join the game after the initial resource allocation and are connected to the MS. The new players are not aware of the initial configuration of SSs, and therefore we assume that they can only connect to the MS. The latency for original users may increase above the QoS bound as new players join the game. The heuristics will provide a resource allocation that maximizes the number of new players that can be connected to the MS, while still maintaining the QoS for all users.

4.4.2 Robustness Metric

4.4.2.1 Overview

Using the FePIA (Performance Features, Perturbation Parameters, Impact, and Analysis) procedure described in [4], we define the characteristics that make the system robust. The FePIA procedure should respond to three fundamental robustness questions [4, 6]. First, what behavior of the system makes it robust? Second, what uncertainties is the system robust against? Quantitatively, exactly how robust is the system?

4.4.2.2 Performance Feature

The first step of the FePIA procedure is to describe quantitatively the QoS requirement that makes the system robust. The requirement that makes the system robust is that all the RTs are within a pre-determined range. The maximum RT time the system can allow is $\underline{\beta_{max}}$:

$$RT_{max} \leq \beta_{max}. \tag{43}$$

However, to maintain fairness RT_{min} also has a constraint. A time window ($\underline{\Delta_{max}}$) is used to specify the allowable range of RT_x for all users. The constraint that RT_{min} must meet is:

$$RT_{max} - RT_{min} \leq \Delta_{max}. \tag{44}$$

For the system to be robust the constraints shown in Equations 43 and 44 need to be satisfied.

4.4.2.3 Perturbation Parameter

The second step of the FePIA procedure is to determine the perturbation parameter that represents the uncertainty in the system. For this study, the perturbation parameter is the number of new players joining the game after the initial resource allocation is done.

4.4.2.4 Impact of Perturbation Parameter on the QoS Performance Features

In this study, it is assumed that new players joining a game in progress connect to the MS. When new players join, the computation at the MS will increase quadratically. This increase in time will make the RT of users that are already in the game increase, and hence RT_{max} will increase. Thus, if the initial resource allocation satisfies Equation 44, then it will remain satisfied.

Let $\underline{RT_{new}}$ be the RT for a new player. We assume the system does not allow new players whose response time exceeds RT_{max} (i.e., $RT_{new} < RT_{max}$); or violates the fairness criteria (i.e., $RT_{max} - RT_{new} \leq \Delta_{max}$). Therefore new players have comparable time to other connected users.

4.4.2.5 Analysis

The number of new players that can be added to the system before RT_{max} and RT_{min} violate the QoS constraints can be calculated exactly if $RT_{max} \leq \beta_{max}$. For a given resource allocation and an associated set of secondary servers, we can calculate how many new players can join the game before the QoS constraints are violated. We define $\underline{\Gamma}$ as the components of the RT equation that do not depend on the number of players connected to the MS. When U_x is connected to the MS, Γ is given by:

$$\Gamma = Comm(U_x, MS) + Comm(MS, U_x), \tag{45}$$

and if U_x is connected to SS_α then Γ is:

$$\begin{aligned}\Gamma &= Comm(U_x, SS_\alpha) + Comp_\alpha + Comm(SS_\alpha, MS) \\ &+ Comm(MS, SS_\alpha) + Comm(SS_\alpha, U_x). \end{aligned} \tag{46}$$

Therefore,

$$RT_{max} = \Gamma + 2 \cdot Comp_{MS}. \tag{47}$$

The system will be at the boundary of robustness when RT_{max} is equal to β_{max} with $\Delta = Comp_{MS}$, that is

$$\beta_{max} = \Gamma + Comp_{MS} + \Delta, \ or \tag{48}$$

$$\beta_{max} = \Gamma + 2 \cdot Comp_{MS} \ . \tag{49}$$

Let n_{new} represent the number of new players that can be added, and n be equal to $n_{main} + n_{nss}$. This implies that

$$\beta_{max} = \Gamma + 2 \cdot (c \cdot n_{secondary} + b \cdot (n + n_{new})^2) \ . \tag{50}$$

The quadratic term can be expanded so that

$$\beta_{max} = \Gamma + 2 \cdot (c \cdot n_{secondary} + b \cdot ((n)^2 + 2 \cdot n \cdot n_{new} + n_{new}^2)). \tag{51}$$

Using Equation 47, this can be simplified to

$$\beta_{max} = RT_{max} + 2 \cdot b \cdot (2 \cdot n \cdot n_{new} + n_{new}^2). \tag{52}$$

This can be re-written in standard quadratic form:

$$2 \cdot b \cdot n_{new}^2 + 4 \cdot b \cdot n \cdot n_{new} + (RT_{max} - \beta_{max}) = 0. \tag{53}$$

With the roots given by the quadratic formula, the <u>robustness</u> <u>metric</u>, the maximum number of new players that can be added, is quantified as:

$$n_{new} = -n \pm \sqrt{n^2 - \frac{RT_{max} - \beta_{max}}{2 \cdot b}}. \tag{54}$$

This result requires some interpretation, because it has two roots. If Equation 54 has two real roots, then the largest value is selected. If the largest value is positive then this is the number of players the current resource allocation can add without violating the QoS constraints. If the largest value is negative then this is the number of players that need to be removed for the system to become robust. If the roots generated by Equation 54 are complex then the robustness cannot be achieved due to excessive communication or computation at an SS. The value of the robustness metric is based on RT_{max} which is determined by the given resource allocation; hence, better resource allocation will result in larger values for the robustness metrics.

For some heuristics, it is necessary to give a "robustness" value to all resource allocations. If the resource allocation cannot achieve robustness (i.e., Equation 54 has two complex roots), then we approximate the robustness. In this case, the robustness is calculated as:

$$n_{new} = \frac{-\sqrt{RT_{max} - \beta_{max}}}{\sqrt{2 \cdot b}}. \tag{55}$$

This gives a negative bias to all the resource allocations that cannot reach robustness.

4.4.3 Heuristics for Maximizing Robustness to Additional Players Joining the Game

4.4.3.1 Recursive Optimization Algorithm for Robustness (ROAR)

The Recursive Optimization Algorithm for Robustness (ROAR) iteratively adds SSs and uses the Min-Min RT algorithm to assign non-DCUs to DCUs. Initially, the ROAR heuristic creates a sorted list ($COMM_{list}$) of users in ascending order of communication time to the MS. The first element of this list is added as a DCU, and the Min-Min RT heuristic is used to assign the non-DCUs. If the constraints are met by this resource allocation, then the robustness is calculated, and compared against the best known robustness. If the constraints are not met, then the next element in $COMM_{list}$ is also added as a DCU. This procedure continues until a stopping criteria is met, which is the number of iterations without improvement, or we have added all the users as SSs. The procedure for the ROAR heuristic is shown in Figure 55.

4.4.3.2 Robust Tabu Search

The Robust Tabu Search is very similar to the Tabu Search in Section 4.3.2.4. The differences between the two heuristics are: in Figure 48, if this mapping does not meet the

```
(1) Create a list with all users sorted in ascending order based on communication
    time to the $MS$ ($COMM_{list}$).
(2) Set $best\ robustness$, $i$, and $j$ to 0.
(3) While $j <$ stopping criterion and $i < N$ then:
    (a) Add the $i^{th}$ entry in the list to the set of $DCU$s ($\underline{DCU_{set}}$). Note that when
        a user is added to $DCU_{set}$, it will remain in $DCU_{set}$ until the heuristic
        finishes executing.
    (b) Use the Min-Min RT heuristic with $DCU_{set}$ to generate a full mapping.
    (c) If the QoS constraints are not met then go to step (f), otherwise continue
        to step (d).
    (d) Calculate the robustness of the current mapping ($\underline{R_x}$).
    (e) If $R_x > best\ robustness$ then
        (i) $best\ robustness \leftarrow R_x$ and $j \leftarrow 0$.
        (ii) The current mapping is stored as the best known resource allocation.
        Otherwise, increment $j$ by 1
    (f) Increment $i$
(4) The best known resource allocation is output.
```

Figure 55: Procedure for using the ROAR heuristic to generate a resource allocation.

fairness constraints (based on β_{max} and Δ_{max}) in step 2(b) then we go to step 2(a), otherwise continue to step 2(c), and we switch from minimizing RT_{max} to maximizing the robustness in steps 2, 3(b), 3(b).ii, 3(f), and 3(f).ii of Figure 49.

4.4.3.3 Robust Discrete Particle Swarm Optimization

The Robust Discrete Particle Swarm Optimization (Robust DPSO) is very similar to the Discrete Particle Swarm Optimization in Section 4.3.2.5. The differences between the two heuristics are the change in the objective function (from minimizing RT_{max} to maximizing robustness in step 2 and 4(b) in Figure 50), and checking the resource allocation to insure the fairness constraints are met in step 1 and 4(b). Note, that in step 5 the best position is based on maximum robustness.

4.4.3.4 Robust Genitor

The Robust Genitor heuristic is very similar to the Genitor RT (Section 4.3.2.6). The differences are that the chromosomes are ranked based on decreasing robustness instead of increasing RT_{max} and chromosomes are only allowed entry into the population if both fairness constraints are met in step 2(d) in Figure 51.

4.4.4 Upper Bound

The primary purpose of deriving a mathematical upper bound was to evaluate the experimental results of our proposed heuristics for the maximization of robustness. This bound is based on the lower bound in Section 4.3.3, and uses the same simplifying assumptions. The basic idea of the upper bound is (1) to find a lower bound on RT_{max} ($\underline{RT_{bound}}$) for each *specific configuration* (i.e., values of n,n_{nss}, and n_{main}), and (2) using RT_{bound} with the number of users connected to the MS ($\underline{n = n_{nss} + n_{main}}$) to calculate a true upper bound using Equation 54.

The function f_{comp} from Section 4.3.3 is used to calculate the computation required given n and n_{nss}, and Equation 56 is the communication part of Equation 42.

$$Comm_{min} = \min_{U_x \in all\ users} \left(\min_{U_y \in all\ users} (f_{comm}(U_x, U_y)) \right). \tag{56}$$

$$RT_{bound}(n, n_{nss}) = f_{comp}(n, n_{nss}) + Comm_{min} \ . \tag{57}$$

The robustness of the bound can be calculated with Equation 54 by substituting RT_{bound} for RT_{max}. If the robustness of a particular configuration can be calculated with the quadratic equation shown in Equation 54, then the discriminant $\underline{f_{quad}}$ will be a positive value, i.e.,

$$f_{quad}(n, n_{nss}) = cdotn^2 - \frac{RT_{bound}(n, n_{nss}) - \beta_{max}}{2 \cdot b}. \tag{58}$$

If f_{quad} is positive, then

$$rob_{max}(n, n_{nss}) = -n + \left| \sqrt{f_{quad}(n, n_{nss})} \right| \ . \tag{59}$$

Otherwise,we have complex roots and the game cannot be played in this configuration. Therefore, there roots would be ignored.

For each configuration there will be a rob_{max}, the upper bound ($\underline{UB}$) will be the maximum rob_{max} over all possible configurations.

$$UB = \max_{1 \leq n \leq N} \left(\max_{0 \leq n_{nss} \leq n} (rob_{max}(n, n_{nss})) \right) \tag{60}$$

To show that the following bound is true, we must first prove that RT_{bound} is a lower bound on RT_{max} for a *specific* n and n_{nss} of DCUs. The value of RT_{bound} is composed of the communication bound and the computation bound.

The bound on communication does an exhaustive search of the possible communication times (through an SS or directly connected to the MS). Therefore, no user can have a smaller communication time than $Comm_{min}$ independent of the configuration.

The bound on the computation will calculate the minimum computation given a specific configuration. It consider values of n from 1 (only one user connected to the MS) to N (all users connected to the MS). For each of these values of n, all possible values of $n_{nss} \leq n$ are considered. Each combination of n and n_{nss} we generate a value of f_{comp}, and the sum of f_{comp} and the bound on communication will give us RT_{bound}. For each RT_{bound}, the robustness can be calculated using Equation 59. The maximum of these robustness values is the UB.

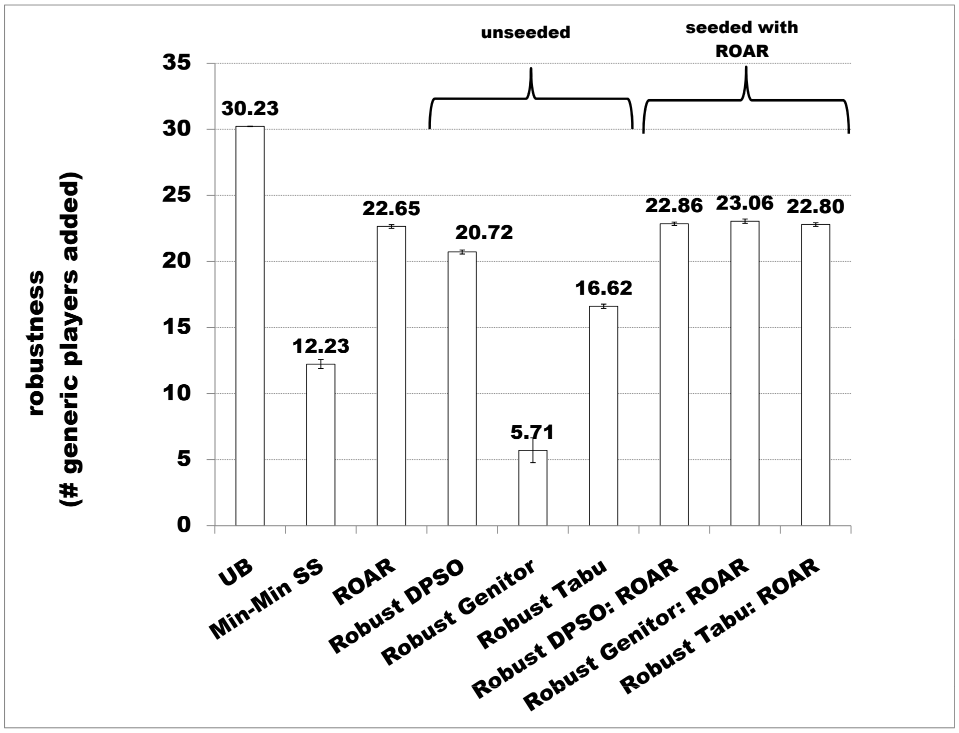


Figure 56: Results for maximizing the robustness of the system against additional players joining the game. The computational parameters of the MS were set to: $b = 0.03$ and $c = 0.01$, values are averaged over 100 scenarios, and the error bars show the 95% confidence intervals. The values for β_{max} and Δ_{max} are 200 and 150 milliseconds, respectively.

4.4.5 Simulation Results

4.4.5.1 Simulation Setup

The simulation setup is the same as in Section 4.3.4.1. The values for β_{max} and Δ_{max} are 200 and 150 milliseconds, respectively.

4.4.5.2 Results for Maximization of Robustness

The Robust Tabu Search, Robust DPSO, and Robust Genitor heuristics were run with and without a seed as shown in Figure 56, the results shown were the best results for each heuristic found after doing parameter sweeps on controlled parameters, e.g., probability of mutation in the Robust Genitor, and velocity weighting parameters in DPSO. The Robust Tabu Search, Robust DPSO, and Robust Genitor heuristics had an execution time of 10 minutes, while the ROAR and Min-Min SS heuristics had an execution time of less than 1 minute.

The performance of the seeded Robust Genitor and Robust DPSO had similar performance (about 22-23 players could be added). The unseeded Robust Genitor did not perform well, this could due to the method used for generating random resource allocations, i.e., resource allocation with a negative robustness were not screened out of the initial population.

The Min-Min SS heuristic (described in Section 4.3.2.2) was used as a comparison to see how heuristics that optimize RT_{max} perform when considering robustness as the optimization criterion. It had a performance that was not able to add as many users as the ROAR heuristic (on average it could add approximately 8 less players).

For the Robust Tabu, the average result from the long hop was a robustness of 9.06 users (a total of 3458 long hops were executed). The average improvement obtained by the local search was 24.45% upon the initial resource allocation with an average of 24.5 short hops. This shows that the short hops are able to improve the resource allocation by exploring the neighborhood.

The Robust Tabu, Robust Genitor, and Robust DPSO significantly improved with the introduction of the seed. However, this improvement in performance was mostly due to the high robustness generated by the ROAR heuristic. The DPSO heuristic had a 0.21 ($\approx$ 1%) improvement, Tabu Search had a 0.15 ($\approx$ 1%) improvement, and Robust Genitor had a 0.41 ($\approx$ 2%) over the ROAR heuristic. The results of the ROAR seeded heuristics were (on average) 7.4 time units less than the UB (about 76% of the UB).

The results from the heuristics for the maximization show that with the constraints set for this environment, a large number of users can be added while maintaining the fairness conditions (approximately 10% more users). The results from the heuristics for robustness maximization are not very close to the UB. The closest result generated by the heuristics was on average approximately 7 users less than the UB.

4.5 Related Work

Various MMOG architectures are reported in the literature (e.g., client/server [24,29], peer-to-peer [15,43,54], mirrored server [28]). Each architecture has its own advantages. For example, the client/server and mirrored server allows the company that develops the MMOG environment to maintain tight control of the game state. However, there is a significant monetary cost associated with maintaining a large-scale MMOG environment. In a peer-to-peer architecture, because of the absence of a centralized game state controller, no peer has full control over the game state making it difficult to maintain a consistent MMOG environment. The advantage of using a peer-to-peer architecture is that there is no single point of failure and the MMOG environment can be maintained without a significant monetary cost. The use of the centralized server in the hybrid approach may have a single point of failure, however it allows the game developer to control the MMOG environment and uses peers to reduce the computation of the main server. Our work is different from [28, 29] because it considers converting users to secondary servers. Our work is also different from [15, 43, 54] because it has a "non-peer" centralized server, and fairness is not directly addressed.

Maintaining a seamless interactive experience for the users is an important factor in MMOG because an increase in latency within the system can lead to deterioration in the gaming experience [7, 29]. In [43], the authors show that the latency follows a "... weak exponential increase ..." as the number of users in the system increases. Our study focuses on latency as a critical performance parameter that must be maintained and uses the results in [43] to model the relationship between latency and the number of users.

In [24], a hybrid approach is presented where peers are clustered together and they update movement information independent of the main server. Because movement information can make up a significant amount of the traffic generated by an MMOG [59], this off-loading can reduce the main server's computational load. The work in [24] shares similarities with our work, however the focus of our study is resource allocation considering latency and heterogeneity in a hybrid MMOG environment. In regard to heterogeneity, [24] considers the scenarios where the peers are either able or unable to computationally contribute to the system. Our algorithm uses heterogeneity of computational capabilities as part of the information used to make resource allocation decisions. Also, the study in [24] does not use a robustness metric to evaluate resource allocation.

This study proposes a hybrid client/server architecture to combine the best elements of both the centralized client/server and peer-to-peer architectures, and guarantee a robustness criteria that creates a fair environment. Our work is similar to [60], where a distributed system uses intermediate servers (analogous to our definition of secondary servers) to reduce the communication latency to the central server. The main differences between our study and theirs is that in [60] the intermediate servers are predefined and do not participate as users in the MMOG, and we have a robustness criterion to guarantee fairness.

4.6 Conclusions

In this study, we created a detailed mathematical model of a hybrid MMOG environment, and derived metrics to analyze the performance of the system. For the first part of the study, we designed heuristics that minimize the maximum response time (RT_{max}) among every player in this environment. Heuristics for this environment need to determine (a) how many users are converted into secondary servers, (b) which users are converted into secondary servers, and (c) how the remaining users are connected among the secondary servers and the main server. For this environment, we derived a mathematical lower bound on RT_{max}, and showed it to be a true lower bound. We used a simulation study to compare heuristics against each other and to the lower bound. In this part of the study, we saw that we could decrease the response time from approximately 1.2s (with all users connected to the main server) to about 112ms using the proposed resource allocation heuristics.

The mathematical model of the MMOG environment was additionally used to address the problem of adding players to an on-going game session. The problem of adding players was modeled in terms of fairness and robustness. We designed heuristics to maximize a robustness metric, i.e., number of player that can join an on-going game, that guarantees (using QoS constraints) the configuration of the system is fair. We derived a mathematical upper bound on the number of players that can be added, and used this bound to evaluate the performance of the heuristics. In this part of the study, using the proposed resource allocation heuristics, we were able to add approximately 10% more players (≈ 220) while maintaining a system that is fair. If users are connected directly to the main server, then we would able to support a maximum of 81 users within the fairness constraints. This shows that, in our environment, the hybrid client/server configuration found by the resource allocation heuristics can duplicate the number of players that can interact the system.

A possible extension to this study is to make the model more realistic, consider that players may leave (or be banned) during an on-going game session, consider that new players joining can be assigned anywhere (as opposed to only the main server), and determine an allocation given a large set of users that allows the maximum number players among these users to participate in the game session.

CHAPTER V

ROBUST RESOURCE ALLOCATION IN A SATELLITE IMAGING SYSTEM

5.1 Introduction

In this study, we consider a heterogeneous computing (**HC**) system based on multicore chips used to analyze satellite data. The data processing applications used in the analysis typically require computation on large data sets, and their execution may be subject to a completion deadline. Multiple applications, each represented as a directed acyclic graph (**DAG**) of tasks, are to be assigned to an HC system for execution. The goal of this study is to assign tasks to processors in such a way that all applications complete before a common deadline, and the application completion times are robust against uncertainties in task execution times. We define a measure of robustness in this context, and we design, compare, and evaluate five resource allocation heuristics that attempt to maximize robustness, and test them under a variety of situations.

The simulation environment used to compare and evaluate these heuristics is motivated by similar systems in use at DigitalGlobe$^{\pm}$ [1] and the National Center for Atmospheric Research [2] (**NCAR***). In these systems, data from a satellite is received and distributed to storage units in the satellite data processing HC system, where there are: (a) heterogeneous hard drive (**HD**) access rates, (b) different computational capabilities across compute nodes, and (c) different data set sizes. This satellite data processing system has the following characteristics: (a) the initial allocation of satellite data to HDs is determined by the heuristic, (b) there is limited RAM available at each compute node, (c) data items have to be explicitly staged to and removed from RAM, (d) some tasks can be executed using

A preliminary version of portions of this research appeared in [18].

[1] $\pm$ `http://www.DigitalGlobe.com/`

[2] $\star$ `http://www.ncar.ucar.edu/`

parallelization, and (e) transfer times between HD and RAM must be taken into account. The simulation environment models this HC system and the applications.

Each application requires only a subset of the total collection of data that is downloaded from the satellite. Resource allocation in this environment requires both selecting a location within the system to store each satellite data item and mapping tasks to compute nodes for execution; the mapping includes a decision about using data parallelism and multiple cores to each task. All applications and their required satellite data items are known prior to the satellite collecting the data, so this is an instance of a static resource allocation problem [2]. The general mapping problem is NP-complete [26, 34, 42]; therefore, heuristics are required to obtain a near-optimal allocation in a reasonable time.

Our contributions are: (a) a model and simulation of a complex multicore-based data processing environment that executes data intensive applications, (b) a robustness metric for this environment, and (c) resource allocation heuristics to maximize robustness using this metric.

In the next section, we will describe the problem statement. Five heuristics are defined in Section 5.3. The related work is discussed in Section 5.4. Section 5.5 and 5.6 provide the results and conclusions, respectively.

5.2 Problem Statement

5.2.1 System Model

This HC system is composed of $\boldsymbol{N}$ compute nodes, where each compute node j has dedicated storage ($\boldsymbol{DS_j}$)—composed of RAM_j and HD_j. Each compute node also has one to eight processing elements (**PEs**), where each PE in our model may correspond to a core in a real system. We make the simplifying assumption that each PE may only execute one task at a time, i.e., no multi-tasking. The PEs within a compute node are homogeneous, but are assumed to be heterogeneous across compute nodes. Each compute node j has $\boldsymbol{\nu_j}$ PEs where the x^{th} PE is denoted $\boldsymbol{PE_{j,x}}(1 \leq x \leq \nu_j)$, and the total number of PEs across all compute nodes is $\boldsymbol{M}$ ($M = \sum_{j=1}^{N} \nu_j$). The composition of a compute node j is shown in Figure 57. The goal of resource allocations in this environment is to complete all applications before a

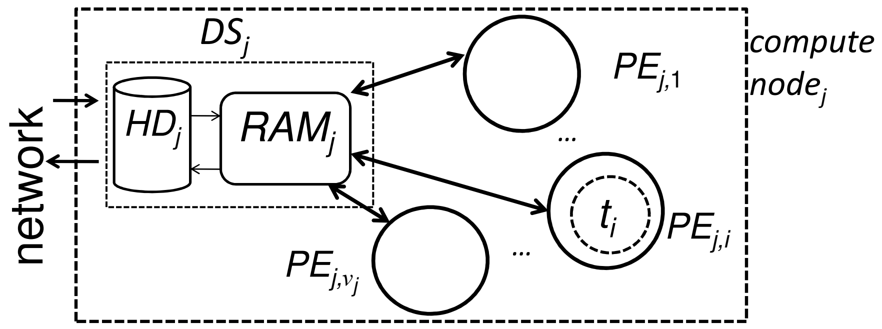


Figure 57: The composition of compute node j is illustrated in this figure.

common deadline ($\boldsymbol{\Delta}$). Let $\boldsymbol{app_k}$ be the k^{th} application. Each app_k is divided into $\boldsymbol{T_k}$ tasks, represented by a DAG. In the DAG, entry tasks require only satellite data and exit tasks produce final results that must be stored on an HD. Without loss of generality, we assume that there is an entry task and an exit task for each DAG. Tasks may require both satellite data sets, denoted $\boldsymbol{SD_i}$, and data sets produced by other tasks, denoted $\boldsymbol{TD_j}$. Within a compute node α, a data set can be located in either RAM or the HD. The location of a data set (TD or SD) within a compute node α is denoted $\boldsymbol{loc_\alpha}$, i.e., $loc_\alpha \in \{RAM_\alpha, HD_\alpha\}$. An example of a resource allocation of a DAG is shown in Figure 58.

In each compute node, the RAM storage space is limited and may be unable to store simultaneously data sets currently needed by tasks assigned to the PEs; however, each HD is assumed to be large enough to store any SDs and TDs assigned to it. If a TD in RAM is to be released, but is required later as an input to a task, then it must be copied to the HD. Because all SDs are initially stored on HDs, SDs do not need to be saved to the HD before being overwritten in RAM. Two methods for initial placement of satellite data are considered: the first one (**ARB**) uses a fixed arbitrary placement, and the second method (**HEU**) lets the heuristic map satellite data sets to compute nodes.

All the input data sets required by a task must be in local RAM before the task can start executing, and must remain in RAM until its execution is finished. The storage space in RAM for the output of a task must be reserved locally before it begins execution. For a given compute node, if there is space currently available in RAM then the required data

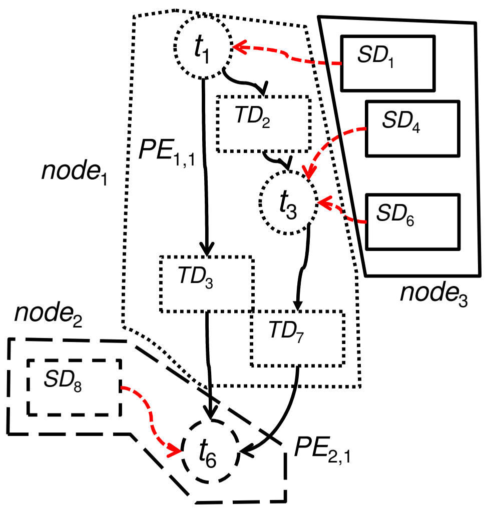


Figure 58: This figure shows a diagram of a DAG. The PE shown in compute node 1 ($PE_{1,1}$) is executing t_1 that requires SD_1 from compute node 3. In this case, TD_3 and TD_7 on compute node 1 need to be transmitted to compute node 2 for t_6. The result of t_6 must be stored in an HD of the system, and the time to store the result must be considered when calculating the makespan.

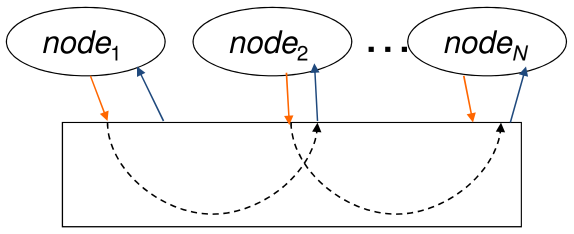


Figure 59: An illustration of HiPPI network is shown in this figure.

sets are sent directly to the RAM on the destination compute node. If there is no space currently available in RAM, then the required data is sent to the HD of the destination node. When a task finishes execution, we evaluate which data sets can be removed from RAM, we select for removal data sets that are not currently being used, and have the fewest remaining tasks that require them (TDs are copied to the HD to avoid lost data). In this model, all PEs on the same compute node share dedicated storage and network access. The time to access local RAM from a PE is assumed to be part of the estimated task execution time. HD access must be considered explicitly, and is different for a read or a write.

The network topology used for this study is a high performance parallel interface

(**HiPPI**) crossbar switch (Figure 59). Each compute node may simultaneously transmit and receive one data set at a time, but may not broadcast. The transfer rate of data from one compute node to another depends on the data's location in both the source and the destination, i.e., RAM or HD.

Because there are two locations where data may be stored and two compute nodes involved in the transfer, there are four cases of data transfer within this system. In the first case, we wish to transfer data from RAM on a source compute node to RAM on a destination compute node. This transfer is only limited by network bandwidth (the same for all compute nodes) because the bandwidth to RAM is always greater. In the second case, the transfer rate of data from the HD on the source compute node to RAM on the destination compute node is limited by the smaller of the network bandwidth and the read bandwidth of the source HD. In the third case, the transfer rate of data from RAM on a source compute node to HD on a destination compute node is limited to the smaller of the network bandwidth and the write bandwidth of the destination HD. In the fourth case, the transfer rate of data from a HD on the source compute node to the HD on the destination compute node is limited by the smaller of the network bandwidth, the read bandwidth from the source HD, and the write bandwidth to the HD on the destination compute node.

For each task $\boldsymbol{t_i}$, we assume that an estimated time to compute on each compute node j has been provided, denoted $\boldsymbol{ETC(i,j)}$, possibly determined from past task execution times or experimentation, which is a common assumption (e.g., [9,30,37,46,50,78,88]). The goal of this study is to assign tasks to PEs so that unexpected increases in the estimated task computation times do not cause the total time required to complete all applications (makespan) to exceed Δ.

A subset of the tasks are designed to be decomposable for parallel execution within a single compute node. These decomposable tasks are grouped into good parallel tasks and poor parallel tasks, based on how amenable the task is to parallel processing. For both "good" and "poor" parallel tasks, a **divisor** value is used to scale the execution time depending on how many PEs a task is being parallelized across. This *parallel* execution time is denoted $\boldsymbol{ETC_{parallel}(i,j)} = \frac{ETC(i,j)}{divisor}$, and the divisor values we use in the simulations

	number of PEs in use							
types of parallelism	1	2	3	4	5	6	7	8
divisor for good parallel tasks	1	1.75	2.5	3.25	4	4.75	5.5	6.25
divisor for poor parallel tasks	1	1.5	2	2.5	3	3.5	4	4.5

Table 3: Table showing the divisor for parallel tasks

are shown in Table 3. In this work, any time a parallelizable task is assigned to a compute node all idle PEs were assigned to execute the task. In our simulation environment, the amount of RAM in the compute node is a constraint that can prevent all PEs from executing different tasks. Therefore, parallelizing tasks makes more efficient use of the memory.

5.2.2 Robustness

A resource allocation is robust if it meets a given performance criterion and is able to maintain this performance despite unexpected perturbations [5, 74, 79]. To quantitatively compare robustness among different possible resource allocations, three questions about robustness must be answered [4]: *(1) What behavior of the system makes it robust?* Our system is robust if all applications complete before a common deadline Δ. *(2) What uncertainties is the system robust against?* The uncertainty is the relationship between the estimated execution time of each task and the actual data dependent execution time of each task. *(3) Quantitatively, exactly how robust is the system?* In this study, we define robustness of a given resource allocation to be the smallest common percentage increase ($\boldsymbol{\rho}$) for all task execution times that causes the makespan to be equal to the deadline Δ. Thus, the robustness metric ρ is maximized for this study. An example of a resource allocation with a robustness of 50% is shown in Figure 60.

5.2.3 Performance Metric

The performance metric for this study is ρ. Due to the complexity of the environment, it is difficult to identify a closed-form expression for ρ. Robustness is different than just increasing the makespan by ρ, because of the complexity introduced by the inter-compute node data transfers. Thus, an iterative search procedure is used.

In a real system, the execution times of all tasks will not be increased by the same percentage. However, ρ can be used as a suitable measure for robustness in this environment—it

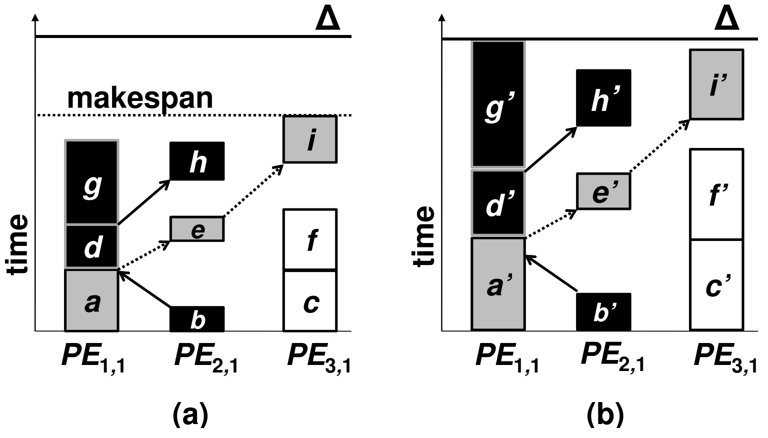


Figure 60: In this figure, an example of a resource allocation is shown. Each of the execution times for the tasks in (a) is increased by 50% with the results shown in (b). Note that the communication times do not increase, and the makespan for (b) (equal to Δ) is much less than makespan in (a) increased by 50%. In (a), the makespan PE (the PE that determines the makespan) is $PE_{3,1}$. After all task execution times are increased by 50%, $PE_{3,1}$ is no longer the makespan PE. This example intuitively shows how the makespan is not a good measure of robustness.

can be viewed as a worst-case guarantee.

We define the makespan with a λ% percent of the execution times as $\boldsymbol{makespan_{\lambda}}$. One procedure to calculate λ, such that $makespan_{\lambda} = \Delta$, is to multiply all the estimated task execution times by λ. Then, using a binary search, the value of λ is found to the nearest percent. The starting upper value of λ for the binary search is an upper limit on λ ($\boldsymbol{UL_{\lambda}}$).

UL_{λ} for the binary search is calculated as follows. For each PE, sum the ETC values of the tasks assigned to that PE. Let μ be the maximum value of these sums among all PEs. The starting value for the binary search is:

$$UL_{\lambda} = \frac{\Delta}{\mu}. \tag{61}$$

The binary search will go between 0% (all computation times are set to 0 time units), and UL_{λ}. For each iteration of the binary search, we calculate the makespan (including communication) until you find the value of λ (to the nearest percent) gives $makespan_{\lambda} = \Delta$.

(1) Let $transfer_{time} = 0$.

(2) For a given task t_i on PE j (PE_j)

   (a) For each data set dat (satellite and inter-task) required by t_i at PE_j:

      (i) Determine the smallest transfer time from all possible sources to move dat to the RAM of the compute node associated to PE_j.

      (ii) Add the transfer time from 1(i) to $transfer_{time}$ (ignoring availability of communication channels of compute nodes).

(3) For each dependency of the destination task, determine the maximum completion of among all the predecessor tasks ($\boldsymbol{max_{source}}$).

(4) Estimated completion time of t_i on PE_j is equal to the sum of $transfer_{time}$ and the maximum of max_{source} and the ready time of PE_j.

Figure 61: Procedure for estimating a completion time in our environment.

(1) A random task $\boldsymbol{t_{rand}}$ is selected for mutation.

(2) A valid range where t_{rand} can be moved in the ordering without violating any precedence constraint is determined (i.e., after all predecessor and before all successor tasks).

(3) A random position within this range is selected, and t_{rand} is moved to this new position.

Figure 62: Procedure for modifying a total ordering.

5.3 Heuristics

5.3.1 Multicore Minimum Completion Time

The Multicore Minimum Completion Time (**MC MCT**) heuristic estimates the completion time of each resource allocation, and assigns each task to the machine that gives it its minimum completion time. The MC MCT uses a known total ordering to generate a resource allocation. The procedure used to calculate the estimated completion time is shown in Figure 61, the procedure for modifying a total ordering (based on the mutation in [83]) is used to create multiple MC MCT solutions is shown in Figure 62, and the complete procedure for MC MCT is shown in Figure 63.

(1) Generate a valid total ordering for all tasks (DAG_{list}).

(2) For 100 iterations the heuristic we execute steps i-ix

   (i) The first task in DAG_{list} is assigned to its estimated minimum completion time PE (Figure 61).
   (ii) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
   (iii) The RAM of the compute node is allocated for this task's data sets (input and ouput) as soon as possible.
   (iv) If it is possible then the task is parallelized across multiple PEs.
   (v) The task used in step (i) is removed from DAG_{list}.
   (vi) The ready time of the PE on which the task is assigned is updated.
   (vii) Steps (i)-(vi) are repeated until all the tasks have been mapped.
   (viii) Calculate the robustness of the resource allocation, and keep the best solution across iterations.
   (ix) Mutate the total ordering using the procedure shown in Figure 62.

(3) Output the best solution.

Figure 63: Procedure for generating a resource allocation using Multicore MCT.

5.3.2 Multicore Random Resource Allocation

The Multicore Random Resource Allocation heuristic uses an arbitrary total ordering, and assigns the tasks (in order) to a randomly selected PE. The procedure is the same as that of MC MCT except that in step 2(i) of Figure 63 we assign tasks to PEs randomly.

5.3.3 Heterogeneous Robust Duplication (HRD)

5.3.3.1 Overview

Heterogeneous Robust Duplication (**HRD**) is based on the Highest Critical Parents with Fast Duplication heuristic introduced in [40]. The algorithm has a listing phase, where tasks have their priority computed and are inserted into a queue based on that priority. The algorithm then has a scheduling phase, where the tasks are assigned to PEs in order of the queue.

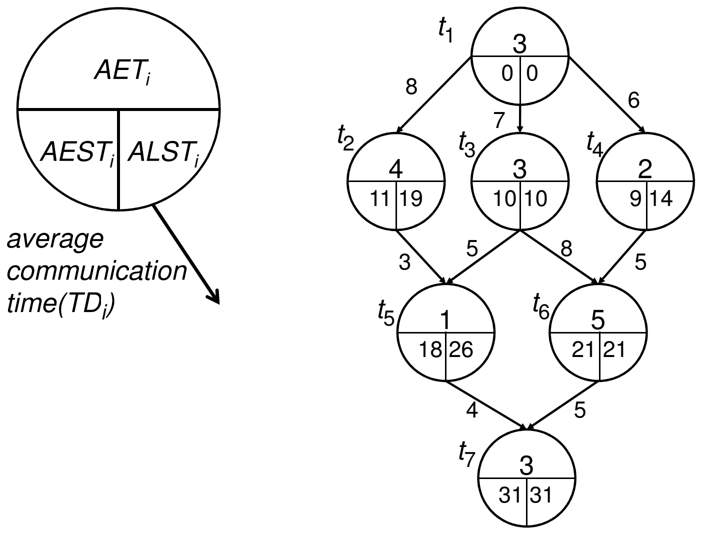


Figure 64: Figure showing an example of the AEST and ALST computation. The AEST is calculated first starting from t_1 to t_7. After all the AESTs are calculated, the ALST of t_7 is set to the AEST of t_7 (in this case 31), and the ALST of the all the tasks from t_7 to t_1 is calculated.

```
(1) For each application DAG
    (a) Traverse the DAG downward (starting at $t_{entry}$), computing the AEST for
        each task.
    (b) Traverse the DAG upward (starting at $t_{exit}$), computing the ALST for each
        task.
    (c) Identify all critical nodes (where $ALST = AEST$).
    (d) Push the critical nodes on the stack ($\boldsymbol{S}$) in descending order of their ALST.
    (e) While $S$ is not empty do
        (i) If the task at the top of the stack $S$ has a parent that is not in $L$ then
            push the parent on $S$ (so that the parent is at the top of the stack).
        (ii) Else pop $S$ and enqueue on L.
```

Figure 65: Procedure used to generate the HRD list.

5.3.3.2 The Listing Phase

We denote the average communication time for the transferring a data item ($\boldsymbol{dat_{i,j}}$) created by t_i and consumed by t_j between any pair of compute nodes as $\boldsymbol{ACT(dat_{i,j})}$, and the transfer time of $dat_{i,j}$ between the HD or RAM of the source compute node k and the HD or RAM of the destination node l as $\boldsymbol{TT(dat_{i,j}, k, l)}$. These parameters are calculated as follows:

(1) Generate the HRD list using the procedure in Figure 65.
(2) For each task in the HRD list
    (a) A task (t_{map}) is de-queued from L
    (b) For t_{map} on each PE_j
        (i) Calculate the robustness (considering communication, memory allocation, and parallelization) of assigning t_{map} to PE_j.
        (ii) Calculate the completion time (considering communication, memory allocation, and parallelization).
        (iii) We compare this allocation of t_{map} to the best allocation of t_{map}.
        (iv) If the DTS is large enough to hold the execution time of $t_{critical}$, then attempt duplication. Keep the duplicated critical task if the robustness is improved.
        (v) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
        (vi) The RAM of the compute node is allocated for this task's data sets (input and output) as soon as possible.
        (vii) If it is possible then the task is parallelized across multiple PEs.
        (viii) The ready time of the PE on which the task is assigned is updated.
(3) Output the solution.

Figure 66: Procedure used to map tasks to machines for the HRD heuristic.

$$\begin{aligned} TT_{average}(dat_{i,j},\alpha,\beta) &= \frac{TT(dat_{i,j},RAM_\alpha,HD_\beta)+TT(dat_{i,j},HD_\alpha,RAM_\beta)}{4} \\ &+ \frac{TT(dat_{i,j},HD\alpha,HD_\beta)+TT(dat_{i,j},RAM_\alpha,RAM_\beta)}{4} \quad (62) \\ ACT(dat_{i,j}) &= \frac{\sum_{\alpha=1}^{N}\sum_{\beta=1}^{N}\left[TT_{average}(dat_{i,j},\alpha,\beta)\right]}{N^2}. \quad (63) \end{aligned}$$

This algorithm calculates two values for each task, the Average Earliest Start Time (**AEST**) and the Average Latest Start Time (**ALST**). We traverse down the DAG computing $AEST(t_i)$ for each task t_i. We define the average execution time of t_i as $\boldsymbol{AET(t_i)}$, and calculate it as follows:

$$AET(t_i) = \frac{\sum_{j=1}^{N} ETC(i,j)\cdot\nu_j}{M}. \quad (64)$$

Let $\mathbf{pred}(t_i)$ be the set of predecessor tasks for t_i in the DAG, and $\mathbf{succ}(t_i)$ be the set of successor tasks. We can calculate $AEST(t_i)$ recursively as follows:

$$AEST(t_{entry}) = 0, \tag{65}$$

$$AEST(t_i) = \max_{t_j \in pred(t_i)} [AEST(t_j) + AET(t_j) + ACT(dat_{i,j}]. \tag{66}$$

To compute $ALST(t_i)$ recursively, we traverse up the DAG, for each task t_i.

$$ALST(t_{exit}) = AEST(t_{exit}), \tag{67}$$

$$ALST(t_i) = \min_{t_j \in succ(t_i)} [ALST(t_j) - ACT(dat_{i,j}] - AET(t_i). \tag{68}$$

Tasks along the critical path (**critical tasks**) have ALST equal to AEST. An example of this is shown in Figure 64.

In the next part of the listing phase, a prioritized queue L is built for the mapping phase, using the procedure in Figure 65. The goal of this procedure is to consider the critical task path as soon as possible.

5.3.3.3 Mapping Phase

For the mapping phase, a task $\boldsymbol{t_{map}}$ is dequeued from L. Initially, assign t_{map} to the PE(s) that results in the maximum robustness value (**MRV**). Multiple PEs may have the same robustness value. If ties occur then we select the PE with the minimum completion time (**MCT**), equal to the estimated ready time of the PE plus the execution time of t_{map} on that PE. We repeat this until the queue is empty.

The idle time slot just prior to the execution of a task is denoted as the duplicate time slot (**DTS**). The DTS for each PE on the same compute node (as the PE selected for assignment) is the start time of t_{map} minus the time when the PE finishes executing the task it has before t_{map}. If the DTS is large enough to hold the execution of the critical parent then duplication is attempted; thus, eliminating the need to incur the communication cost of the critical parent's data set. For this study, we attempted the duplication if the DTS was large enough. However, we only kept the duplication if it improved the robustness. The procedure for the HRD is shown in Figure 66.

```
(1) Let $t_i = t_{exit}$
(2) Calculate average task execution time ($AET(t_i)$).
(3) For each $SD_x$ needed by $t_i$, calculate the estimated transfer time (using
    $ACT(SD_x)$ Equation 63).
(4) For each $TD_j$ needed by $t_i$, calculate the estimated transfer time to
    successor nodes (using $ACT(TD_j)$ Equation 63).
(5) Determine the maximum time from any successor (child) node
    to the $t_{exit}$ ($\boldsymbol{max_{time}}$).
(6) The critical path value is the sum of each individual $TD$ and $SD$ transfer time,
    $max_{time}$, and $AET(t_i)$.
(7) Select another task (whose successors have a calculated critical path value) and
    go to step (2) until all tasks are processed.
```

Figure 67: Procedure used to calculate the critical path of the DATCP heuristic.

```
(1) Calculate the critical path for each application (using the procedure shown in
    Figure 67).
(2) Dynamically create a list of all tasks available for mapping (i.e., predecessors
    have been mapped).
(3) Determine the task with the longest critical path from the list of available tasks.
(4) For task $t_i$ determined in (3),
    (a) For each PE $k$ in each compute node $j$,
        (i) Calculate the robustness of assigning $t_i$ to $PE_{j,k}$ (including communi-
            cation, memory, and task parallelization).
    (b) Find the compute node $j$ with the highest number of maximum-robustness-
        value PEs (it is possible for other PEs to have the same robustness value).
        This compute node is denoted $\boldsymbol{node_{max}}$.
    (c) Map task to the PE with the maximum robustness in $node_{max}$.
    (d) Communication is scheduled from the compute node (that has a copy of
        the data set) that can start the communication the earliest.
    (e) The RAM of the compute node is allocated for this task's data sets (input
        and ouput) as soon as possible.
    (f) If it is possible then the task is parallelized across multiple PEs.
    (g) The ready time of the PE(s) on which the task is assigned is updated.
    (h) Remove task $t_i$ from list.
(5) Repeat steps (2)–(4) until all tasks are mapped.
```

Figure 68: DATCP heuristic procedure used to generate a resource allocation.

5.3.4 Dynamic Available Tasks Critical Path (DATCP)

5.3.4.1 Base Version

The Dynamic Available Tasks Critical Path (**DATCP**) heuristic is based on the Dynamic Critical Path (DCP) heuristics [57]. It is based on computing the critical path for every arbitrary task, and choosing the ordering based on this critical path value. Without loss of generality, assume that all of the entry nodes of all application DAGs have a common predecessor that is the pseudo-node entry task $\boldsymbol{t_{entry}}$. Similarly, assume there is a unique exit node $\boldsymbol{t_{exit}}$. The calculation of the critical path is a recursive process that begins with t_{exit} and finishes at t_{entry}. Each task calculates its estimate of the average time to reach t_{exit} and then passes this time value to its predecessor tasks. The execution time for a task is calculated using the average values across all compute nodes.

The pseudo-code for the critical path calculation method is in Figure 67. After calculating the critical path, the next step of the heuristic determines the list of available tasks, which are entry tasks or tasks whose predecessors have been mapped. The available task with the longest critical path time is then mapped to the compute node that maximizes the robustness value. That task is then removed from the list of mappable tasks and the process is repeated until all tasks are mapped. The pseudo-code for this algorithm is listed in Figure 68.

5.3.4.2 Variation

A variation of this heuristic (**DATCP-V**) was implemented to determine the effect of breaking ties between two allocations with the same robustness. DATCP assigns the tasks to a PE on the compute node with the maximum number of PEs with the same robustness, and HRD assigns it to the PE that has the minimum completion time. This variation is identical to the DATCP heuristic except that steps 4(a) through 4(c) in Figure 68 are replaced with the steps from HRD that assign a task to the PE with the maximum robustness using minimum completion time to breaks ties (steps 2(b).i and 2(b).iv from Figure 66). The procedure for DATCP-V is shown in Figure 69.

(1) Calculate the critical path for each application (using the procedure shown in Figure 67).

(2) Dynamically create a list of all tasks available for mapping (i.e., predecessors have been mapped).

(3) Determine the task with the longest critical path from the list of available tasks.

(4) For task t_i determined in (3),

  (a) Calculate the robustness (considering communication, memory allocation, and parallelization) of assigning t_{map} to PE_j.

  (b) Calculate the completion time (considering communication, memory allocation, and parallelization).

  (c) We compare this allocation of t_{map} to the best allocation of t_{map}.

  (d) If the DTS is large enough to hold the execution time of $t_{critical}$, then attempt duplication. Keep the duplicated critical task if the robustness is improved.

  (e) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.

  (f) The RAM of the compute node is allocated for this task's data sets (input and ouput) as soon as possible.

  (g) If it is possible then the task is parallelized across multiple PEs.

  (h) The ready time of the PE(s) on which the task is assigned is updated.

  (h) Remove task t_i from list.

(5) Repeat steps (2)–(4) until all tasks are mapped.

Figure 69: DATCP-V heuristic procedure used to generate a resource allocation.

5.3.5 Multicore Dynamic Levels

The Multicore Dynamic Levels **MC DL** heuristic is based on the concept [77]. The static level of a task is computed as an approximate time from the task node to the exit node along the worst-case path on a heterogeneous system. A data arrival time for a task is defined as the time when all required input data arrives for a task at a destination node. The dynamic level is the static level minus the maximum of data arrival time and the time when the processing element is ready to execute a new task. The mappable task/machine pair with the highest dynamic level value is selected for mapping.This dynamic level depends on the state of the system due to previous task-to-PE assignments. The difference between our work and [77] is that this version assigns memory, schedules data transfers, and employs task level parallelism in a multicore environment.

The static level $SL(t_i)$ is calculated recursively using the following equation:

$$SL(t_{exit}) = AET(t_{exit}) \tag{69}$$

$$SL(t_i) = \max_{t_j \in succ(t_i)} [SL(t_j)] + AET(t_i) \tag{70}$$

For each task (t_i) on a PE (PE_j), the current dynamic level is calculated before every task is assigned. Let $\boldsymbol{S}$ be the state of processing and communication resources, $\boldsymbol{DA(t_i, PE_j, S)}$ be the earliest time that all the data transfer to t_i (on PE_j) from its predecessors are guaranteed to be completed, and $\boldsymbol{TF(PE_j, S)}$ be the time when the task assigned to PE_j finishes. The dynamic level of t_i on PE_j at state S ($\boldsymbol{DL(t_i, PE_j, S)}$) is calculate as follows:

$$DL(t_i, PE_j, S) = SL(t_i) - \max\left[TF(PE_j, S), DA(t_i, PE_j, S)\right]. \tag{71}$$

The procedure for the Multicore Dynamic Levels is shown in Figure 70. Despite the MC DL heuristic not considering robustness explicitly it does perform well for certain scenarios.

5.3.6 Multicore Genitor

The Multicore Genitor heuristic is based on the concept of the Genitor heuristic [85]. The Genitor is a steady state heuristic that only does one crossover and mutation operation

(1) Calculate the static levels for all tasks (Equation 70).
(2) While tasks remain unassigned
    (a) For each task-PE pair calculate the dynamic level (Equation 71).
    (b) Using the task-PE pair with the highest dynamic level, assign this task to the corresponding PE.
    (c) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
    (d) The RAM of the compute node is allocated for this task's data sets (input and ouput) as soon as possible.
    (e) If it is possible then the task is parallelized across multiple PEs.
    (f) The ready time of the PE(s) on which the task is assigned is updated.
(3) Output the solution.

· **Figure 70:** Procedure used to assign tasks to machines using dynamic levels.

1) A random position in the scheduling string is selected for a one-point crossover.

2) We create a copy of parent A (denoted **offspring A**), and parent B (denoted **offspring B**).

2) For offspring A, create an ordered list $\boldsymbol{list_A}$with the tasks from the bottom part of its scheduling string.

3) For offspring B, create an ordered list $\boldsymbol{list_B}$with the tasks from the bottom part of its scheduling string.

4) For offspring A, we change its scheduling string so that the tasks in $list_B$ follow the same relative ordering in $list_B$ and offspring A.

5) For offspring B, we change its scheduling string so that the tasks in $list_A$ follow the same relative ordering in $list_A$ and offspring B.

Figure 71: Procedure used for crossover in the Multicore Genitor heuristic.

per iteration, and uses an ordering population to keep the best chromosomes (encountered during the simulation) in the population. The results of the crossover and mutation are evaluated and inserted in the population based on their rank.

The chromosome used in the Multicore Genitor is based on the work in [83]. The chromosome in [83] has two strings: a mapping string and a scheduling string. The mapping string is a vector of length $LN = \sum_{\forall k} T_k$ (recall that T_k is the number of tasks in application k), where the i^{th} entry in the chromosome represents task t_i (assume LN task are uniquely numbered from 1 to LN), and the value of the entry represents the PE where t_i is assigned.

1) An initial population of chromosomes (amount is determined empirically) is generated and evaluated.

    (i) Seeds of Multicore Dynamic Levels heuristic and DATCP are generated.

    (ii) The remaining seeds are created by using the total ordering generated by the Multicore Dynamic Levels heuristic as an initial ordering for the MC MCT heuristic (shown in Figure 63).

(2) While there are less than 1000 iterations without improvement, repeat the following procedure.

    (i) A pair of parents are selected for crossover and mutation using linear bias (1.5 determined experimentally) [85].

    (ii) Two offspring are generated using one-point crossover (for both the mapping and scheduling string). For the scheduling string, the crossover procedure is shown in Figure 71.

    (iii) For each offspring, there is a 1% probability (determined empirically) of mutating each field (determined empirically) in the chromosome. For the scheduling string, the procedure for mutation is shown in Figure 62.

    (iv) The offspring are evaluated and ranked into the population displacing the worst chromosome.

3) The output is the best solution.

Figure 72: Procedure used for resource allocation by the Multicore Genitor.

The scheduling string is also of length LN; however, the j^{th} entry represents the j^{th} task in the total ordering. The difference between the Genitor heuristic in [83] and this paper is that Genitor in this paper maximizes robustness, stages data sets to RAM, parallelizes tasks, and determines satellite placement. The procedure for crossover is shown in Figure 71, and the full procedure for the Multicore Genitor is shown in Figure 72.

5.4 Related Work

The research that discusses scheduling DAGs on multi-processor systems is extensive, e.g., [53,57,61,77,81,82,87]. Resource allocation of operations research and computer literature. In this section, examples of existing heuristics are presented.

The Modified Critical Path (**MCP**) algorithm developed in [87] was designed for homogeneous systems and considers only one application DAG. The heuristic determines the latest possible start time of each task (constrained by the critical path length) and then creates a list of tasks in increasing order of these times. Tasks are selected for mapping in the order of the list. The selected task is then mapped to the machine that allows the earliest start time. Thus, the heuristic attempts to start critical path tasks as early as feasible. There are several differences between [87] and this study. We focus on robustness against uncertainty in execution times, and consider a heterogeneous computing environment.

The authors in [77] developed the Dynamic Level Scheduling (**DLS**) heuristic. The static level of a task is computed as an approximate time from the task node to the exit node along the worst-case path on a heterogeneous system. A data arrival time for a task is defined as the time when all required input data arrives for a task at a destination node. The dynamic level is the static level minus the maximum of data arrival time and the time when the processing element is ready to execute a new task. The mappable task/machine pair with the highest dynamic level value is selected for mapping. The performance metric in [77] was makespan, and the paper did not consider uncertainties or robustness. As indicated earlier, minimizing makespan is not the same as maximizing our robustness measure for DAGs.

The Dynamic Critical Path (**DCP**) heuristic developed in [57] calculates the Absolute Earliest Start Times (**AbEST**) and Absolute Latest Start Time (**AbLST**) for each task. A task is defined to be on the critical path if its AbEST equals its AbLST. At each mapping event, mappable tasks update their AbEST and AbLST to determine which task is on the critical path. The critical path task is then mapped to a compute node that minimizes the Earliest Start Time (**EST**) of the task and the EST of its successor tasks. The DATCP heuristic differs from the DCP heuristic in that the DCP heuristic does not have to deal with uncertainty in task execution times and its performance goal is makespan rather than robustness.

The work in [76] considers a heterogeneous *ad hoc* grid used to compute an application composed of communicating sub-tasks. Both this and our study consider the mapping problem of DAGs in a heterogeneous computing environment; the heuristics in [76] minimize the average battery power consumed while meeting a makespan constraint. This paper focuses on maximizing the ability of a resource allocation to tolerate the uncertainty of execution times, while the work in [76] does not consider uncertainty. The minimization of battery power consumed, as studied in [76], is different than maximizing robustness presented in this study.

5.5 Results

5.5.1 Simulation Setup

Each simulation run (50 total) has 50 unique applications, and each application is composed of 8 to 16 tasks when the number of tasks is chosen with uniform probability. We generate DAGs with unique entry and unique exit tasks. The DAGs have a maximum fanout of 3, and a maximum fan in of 3, similar to the example in Figure 64. This type of DAG was used because it is similar to the DAGs encountered in real-world applications like DigitalGlobe.

The size of the SD and TD sets used in this simulation varies from 1 to 20 GBytes in size. We assume the network bandwidth is 512 Mbytes per second, and that the HC system is composed of four compute nodes. Compute node 1 has eight PEs, compute node 2 has four PEs, and finally compute nodes 3 and 4 have two PEs. All compute nodes have 160 GBytes of RAM (152 are used for staging data and the remaining 8 are assumed to be used to buffer data in and out of the local HD). Half of the tasks in the applications are parallelizable, with half of those having "good" parallelism and half having "poor" parallelism (see Table 3 in Section 5.2). The ETC values of a task are calculated by adding all the incoming data sets (SD and TD in GBytes), and multiplying this sum by an ETC-per-GByte value generated using the coefficient of variation based method described in [3]. Consider that all compute nodes are sorted in descending order based on the number of PEs. If k is greater than j, then the ETC of computing t_i on compute node j (PE running individually) is less than or equal to its ETC on compute node k. Note that even if two nodes have the same number of PEs, they may still have different ETC values. The intuition behind ordering the ETC-per-GByte in this manner is that newer systems will have more cores per multiprocessor, and a PE in this multiprocessor will be faster than a PE in an older multiprocessor. A high-task/high-compute-node heterogeneity [3] is used to simulate the tasks within the DAGs. The heterogeneity values used to generate the ETCs are $V_{task} = \frac{\sigma_{task}}{\mu_{task}} = 0.4$ and

trial	α_{comp}	α_{comm}	Δ
1	Medium (1)	High (1)	Loose (15,500)
2	Medium (1)	High (1)	Loose (10,500)
3	Medium (1)	High (3)	Loose (8,000)
4	Medium (1)	High (3)	Loose (6,000)
5	Medium (2)	High (6)	Loose (8,000)
6	Medium (2)	High (6)	Loose (6,000)

Table 4: Table showing the simulation trials and their respective configurations.

$V_{machine} = \frac{\sigma_{machine}}{\mu_{machine}} = 0.3$, and the mean ($\mu_{task}$) is 1 second/Gbyte.

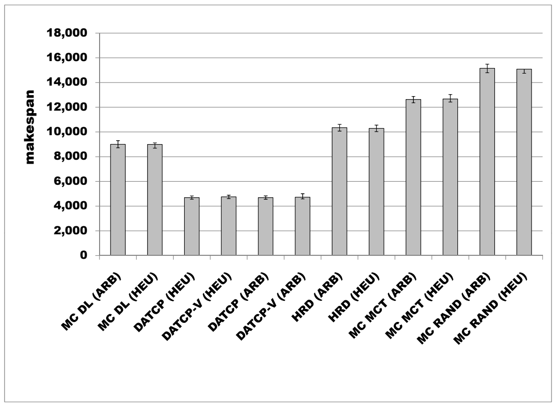
Additional parameters were created to scale the mean computation time and communication bandwidth. The first parameter is $\alpha_{comp} \in \Re$, used to scale the computation by a factor of α_{comp}; and the second parameter is $\alpha_{comm} \in \Re$, used to scale the communication bandwidth by a factor of α_{comm}. The different trials simulated in this study are shown in Table 4.

5.5.2 Simulation Results

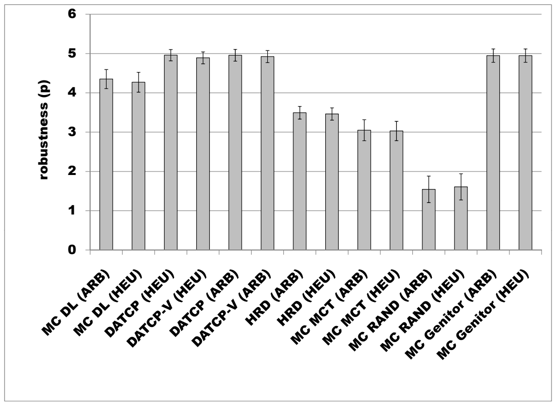
5.5.2.1 Medium Computation and Large Communication

In Figure 73, we can observe that both DATCP variations have the best performance in terms of makespan and robustness. It is important to remember that the fitness functions used to assign tasks to PEs in DATCP and DATCP-V are very different; however, their performance is comparable. The scenario with α_{comp} and $\alpha_{comm} = 1$ has long communication times compared to computation times. The MC Random heuristic had the worst performance and the highest makespan, because it does not consider communication or computation. Therefore, it will often assign tasks from one application across multiple compute nodes, and these tasks may be unable to start executing quickly because of large communication delays in transferring required data.

The results with $\alpha_{comp} = 1$, $\alpha_{comm} = 1$, and $\Delta = 10,500$ shown in Figure 74 show the DATCP and DATCP-V heuristics significantly outperforming the other heuristics (both in makespan and robustness). In this scenario, the HRD was barely able to keep a performance of 1 (makespan equal to deadline), the MC MCT was on average not able to meet the deadline line constraint, and the MC Random heuristic did not make the deadline on any run.



(a) Makespan



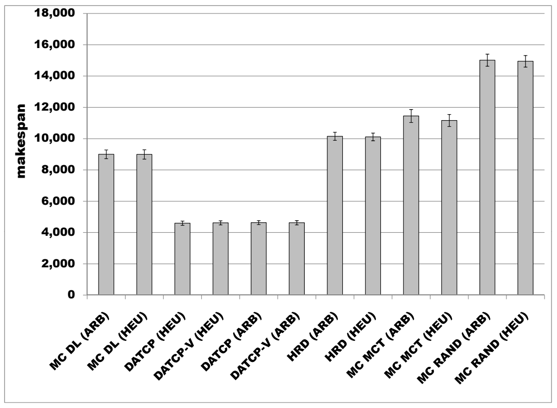
(b) Robustness

Figure 73: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = \alpha_{comm} = 1$ and $\Delta = 15{,}500$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

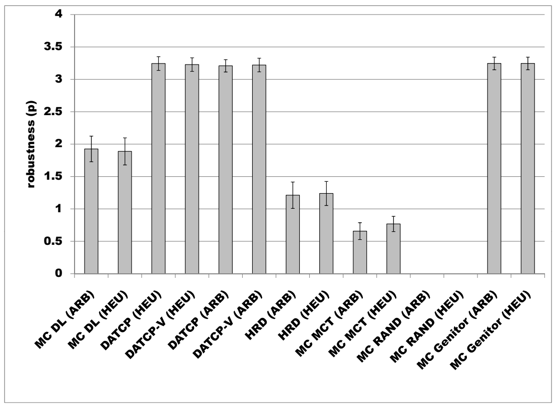
5.5.2.2 Medium Computation and Medium Communication

For the results in Figures 75 and 76, we reduced the average communication times by a factor of 3. This causes the scheduling gaps created by cross-compute-node communication to be reduced significantly.

In Figures 75, the DATCP and DATCP-V outperform the other heuristics in regards to makespan; however, in terms of robustness the DATCP and DATCP-V have comparable performance to MC DL heuristics. The results in Figure 76 show that even with the reduced communication times, when the deadline is tightened, the DATCP and DATCP-V outperform the other heuristics (both in makespan and robustness). The MC DL heuristic



(a) Makespan



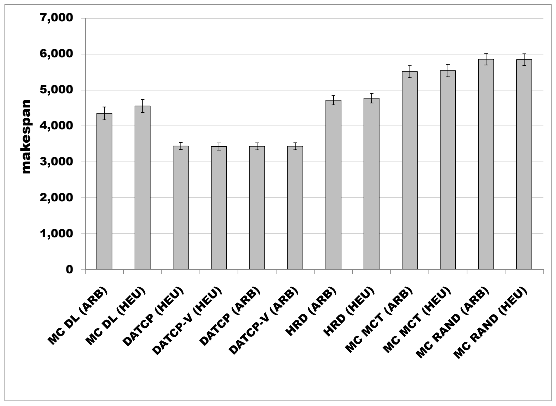
(b) Robustness

Figure 74: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = \alpha_{comm} = 1$ and $\Delta = 10{,}500$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

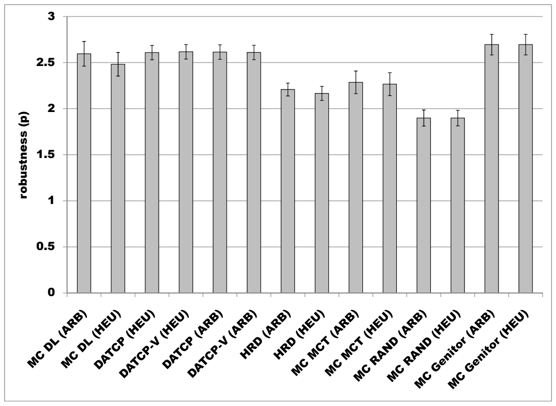
was the second best performing heuristic, and the difference in performance between it and DATCP was reduced in this scenario (compared to the two scenarios in Section 5.5.2.1). In this scenario, the MC Random heuristic was the worst performing heuristic; however the performance in the tight deadline case improved significantly. This was due to a reduction in the penalty of communicating.

5.5.2.3 Loose and Tight Deadlines with α_{comp}=2, and $\alpha_{comm} = 6$.

For the results in Figures 77 and 78, we increased the computation time by a factor of two and reduced the average communication times by a factor of six. This drastically reduces the gaps created by cross-compute-node communication. This causes the makespan and



(a) Makespan



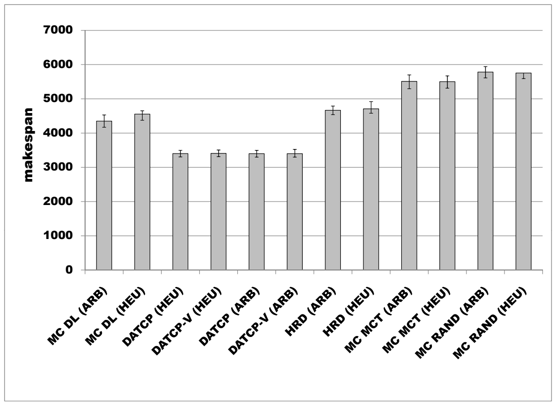
(b) Robustness

Figure 75: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = 1$, $\alpha_{comm} = 3$, and $\Delta = 8000$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

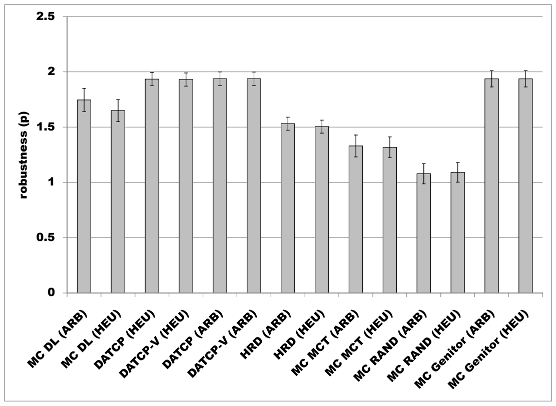
robustness to be even more closely correlated, because a change in robustness causes a proportional linear increase in the makespan. This will be shown in Section 5.5.2.4.

For this scenario, the MC DL heuristic and the MC MCT heuristics on average perform better (shown in Figure 77). However, the MC MCT, DATCP, and DATCP-V heuristics have overlapping confidence intervals, and MC DL has overlapping confidence intervals with MC MCT. In this scenario, the HRD and MC random are the worst performing heuristics.

In Figure 78, we see that the performance of DATCP is comparable to the best performing heuristics when we make the deadline tighter. In this case, the performance of MC DL, MC MCT, DATCP, and DATCP-V is comparable.



(a) Makespan



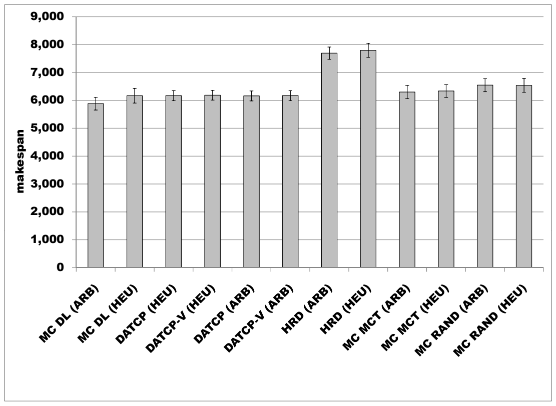
(b) Robustness

Figure 76: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = 1$, $\alpha_{comm} = 3$, and $\Delta = 6000$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

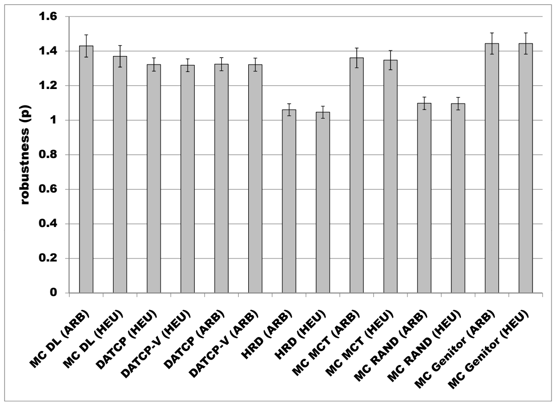
5.5.2.4 Comparison of Makespan vs. Robustness with Loose Deadlines

In Figure 79, the MC Random heuristic to generate 20,000 resource allocations, and the deadlines were made loose to avoid the random heuristic from only finding solutions that had a makespan that was close to the deadline. The three cases shown represent a scenario where communication is dominant 79(a), communication and computation are comparable 79(b), and a scenario where computation is dominant 79(c).

It is interesting to see that as the computation increased (higher computation to communication ratio) the robustness and makespan follow the relationship of $\Delta/makespan$. However, when communication is predominant then there is a large deviation between



(a) Makespan



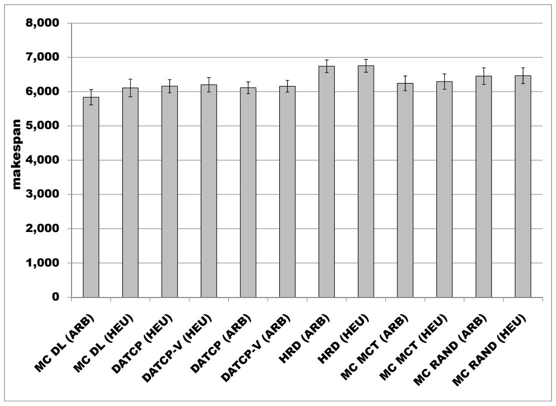
(b) Robustness

Figure 77: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = 2$, $\alpha_{comm} = 6$, and $\Delta = 8000$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

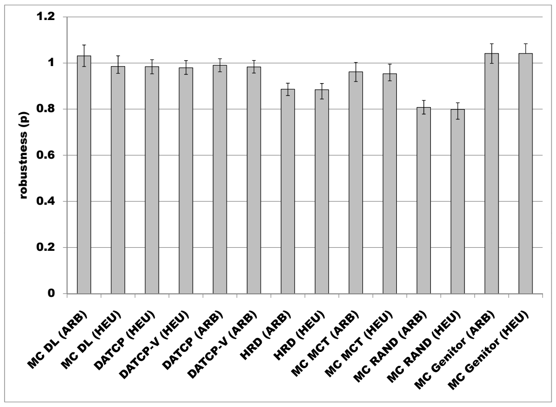
$\Delta/makespan$ and robustness. The results in Figure 79 indicate that using our robustness metric in systems where communication is greater or comparable to computation is better than using it when communication times are small or close to negligible compared to computation times.

5.6 Conclusions

This study focused on using heuristics to allocate resources for multiple applications (formed by DAGs of tasks) to an HC environment based on multicore chips. The goal of resource allocations is to meet the deadline constraint while being robust against uncertainty in execution times. We modeled a heterogeneous computing system used for satellite image processing,



(a) Makespan

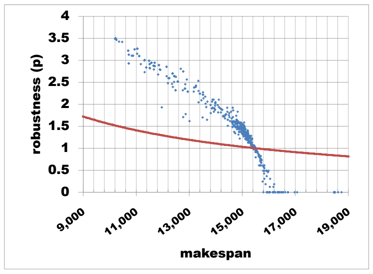


(b) Robustness

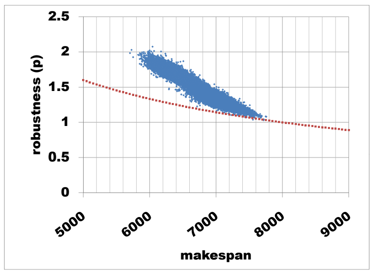
Figure 78: Results for the heuristics defined in Section 5.3 with $\alpha_{comp} = 2$, $\alpha_{comm} = 6$, and $\Delta = 6000$. Recall that the two variations of satellite placement are arbitrary (ARB) and heuristic driven (HEU). The results are shown with a 95% confidence interval.

defined a mathematical robustness metric, and created and evaluated resource allocation heuristics that maximize this robustness metric. We showed that there is an important advantage in using robustness to guide the resource allocation done by the heuristics.

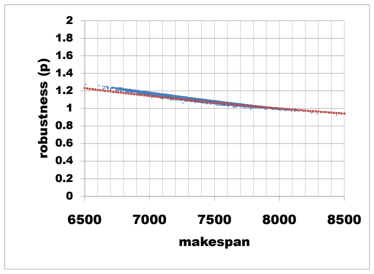
The full ordering of tasks is an important factor for making a good allocation of resources in the system. The experiment that substituted the fitness function used to assign tasks in the DATCP heuristic with the function used for the HRD proved that there was not a significant difference in how the two fitness functions performed. As computation becomes dominant, the differences between heuristics that minimize makespan and those that increase robustness become less significant. Because the makespan is more dependent



(a)



(b)



(c)

Figure 79: Scatter plot of makespan vs. robustness for 10,000 resource allocations of the Multicore MCT heuristic in three different scenarios: (a) α_{comp} =1, α_{comm} = 1, and Δ = 15,500; (b) α_{comp} = 1, α_{comm} = 3, and Δ = 8,000; and (c) α_{comp} = 2, α_{comm} = 6, and Δ = 8,000. Additionally, a line showing $\Delta/makespan$ (square markers) was graphed in (a), (b), and (c) to show how makespan and robustness are related based on communication time.

on increases in the computation time.

A factor that was not important to the performance of the heuristics was the placement of satellite data. This was due to the ability of the heuristics to compensate for different initial satellite data placements. The MC Genitor was unable to improve the performance of the MC DL and DATCP heuristics. To observe if the performance of the MC Genitor could be improved, we removed the stopping criteria and allowed the MC Genitor to run for 10,000 iterations for 5 of the 50 simulation runs. The performance, in these trials, was only improved by 1 to 2% when allowed to run for all 10,000 iterations.

Future work could include using more realistic data instead of using the coefficient of variation based method to generate the ETC values for the simulations. Also, we could refine the hardware model to do a more detailed simulation of multicore chips using the framework we have designed here (e.g., a more sophisticated communication topology).

CHAPTER VI

CONCLUSIONS

In the various environments studied in this thesis, we have shown that robustness is fundamentally different than other performance criteria (e.g., makespan, slack). It is possible, in some test scenarios, for them to be correlated. The advantage of using robustness is being able to give a guarantee on performance given variations in system parameters. This is where traditional performance criteria and custom tailored robustness metrics are different. Because we know that there is uncertainty in a system parameter, we are able to produce resource allocations that are resilient against variations in this parameter.

For the iterative approach, we initially thought that using a resource allocation heuristic on a subset of tasks and machines would improve the performance of the system. However, we saw that the resource allocation generated by some heuristics had variation, while those generated by other heuristics did not have any variation. When we investigated why this occurred we discovered that some heuristics are iteration invariant. For a subset of these heuristics (generalized completion time iteration invariant heuristics), we derived a theorem to prove why these heuristics will always produce the same resource allocation when used with the iterative approach.

REFERENCES

[1] I. Al-Azzoni and D. G. Down, "Linear programming based affinity scheduling for heterogeneous computing systems," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '07)*, Jun. 2007.

[2] 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, and B. Yao, "Characterizing resource allocation heuristics for heterogeneous computing systems," in *Advances in Computers Volume 63: Parallel, Distributed, and Pervasive Computing*, 2005, pp. 91–128.

[3] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang Journal of Science and Engineering, Special 50th Anniversary Issue*, vol. 3, no. 3, pp. 195–207, Nov. 2000.

[4] S. Ali, A. A. Maciejewski, and H. J. Siegel, "Perspectives on robust resource allocation for heterogeneous parallel systems," in *Handbook of Parallel Computing: Models, Algorithms, and Applications*, S. Rajasekaran and J. Reif, Eds. Boca Raton, FL: Chapman & Hall/CRC Press, 2008, pp. 41–1–41–30.

[5] S. Ali, A. A. Maciejewski, H. J. Siegel, and J.-K. Kim, "Measuring the robustness of a resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 7, pp. 630–641, Jul. 2004.

[6] ——, "Measuring the robustness of a resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 7, pp. 630–641, Jul. 2004.

[7] G. J. Armitage, "An experimental estimation of latency sensitivity in multiplayer quake 3," in *11th IEEE International Conference on Networks (ICON '03)*, Sep. 2003, pp. 137–141.

[8] C. Artigues, J. Billaut, and C. Esswein, "Maximization of solution flexibility for robust shop scheduling," *European Journal of Operational Research*, vol. 165, no. 2, pp. 314–328, 2005.

[9] H. Barada, S. M. Sait, and N. Baig, "Task matching and scheduling in heterogeneous systems using simulated evolution," in *10th IEEE Heterogeneous Computing Workshop (HCW '01)*, Apr. 2001, pp. 875–882.

[10] L. Barbulescu, A. E. Howe, L. D. Whitley, and M. Roberts, "Trading places: How to schedule more in a multi-resource oversubscribed scheduling problem system," in *International Conference on Automated Planning and Scheduling (ICAPS-04)*, Jun. 2004.

[11] L. Barbulescu, L. D. Whitley, and A. E. Howe, "Leap before you look: An effective strategy in an oversubscribed scheduling problem," in *19th National Conference on Artificial Intelligence*, Jul. 2004, pp. 143–148.

[12] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games," in *IEEE Conference on Computer Communications (INFOCOM '01)*, Mar. 2001, pp. 104–113.

[13] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan, "New grid scheduling and rescheduling methods in the GrADS project," *International Journal of Parallel Programming*, vol. 33, no. 2–3, Jun. 2005, uses Min-Min, Max-Min, and sufferage.

[14] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using apples," *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[15] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," in *3rd Symposium on Networked Systems Design and Implementation*, 2006, pp. 155–168.

[16] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2, may 2005, pp. 759 – 767 Vol. 2, uses min-min in a grid environment.

[17] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[18] L. Briceno, J. Smith, H. J. Siegel, A. A. Maciejewski, P. Maxwell, R. Wakefield, A. M. Al-Qawasmeh, R. C. Chiang, and J. Li, "Robust resource allocation of dags in a heterogeneous multicore system," in *19th International Heterogeneity in Computing Workshop (HCW'10),*, 2010.

[19] L. D. Briceno, M. Oltikar, H. J. Siegel, and A. A. Maciejewski, "Study of an iterative technique to minimize completion times of non-makespan machines," in *16th Heterogeneous Computing Workshop (HCW 2007), in the proceedings of the 21st International Parallel and Distributed Processing Symposium*, Mar 2007.

[20] L. D. Briceño, M. Oltikar, H. J. Siegel, and A. A. Maciejewski, "Study of an iterative technique to minimize completion times on non-makespan machines," in *International Heterogeneity in Computing Workshop (HCW '07)*, Mar. 2007, p. 138.

[21] L. D. Briceno, H. J. Siegel, A. A. Maciejewski, Y. Hong, B. Lock, M. N. Teli, F. Wedyan, C. Panaccione, and C. Zhang, "Resource allocation in a client/server hybrid network for virtual world environments," in *International Heterogeneity in Computing Workshop (HCW '08)*, 2008.

[22] L. D. Briceño, H. J. Siegel, A. A. Maciejewski, Y. Hong, B. Lock, M. N. Teli, F. Wedyan, C. Panaccione, and C. Zhang, "Robust resource allocation in a massive multiplayer online gaming environment," in *International Conference on the Foundations of Digital Games 2009 (ICFDG 2009)*, Apr. 2009.

[23] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *9th IEEE Heterogeneous Computing Workshop (HCW 2000)*, Mar. 2000, pp. 349–363.

[24] A. Chen and R. Muntz, "Peer clustering: A hybrid approach to distributed virtual environments," in *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

[25] R. Cheng, M. Gen, and Y. Tsujimura, "A tutorial survey of job-shop scheduling problems using genetic algorithms–i. representation," *Computers & Industrial Engineering*, vol. 30, no. 4, pp. 983 – 997, 1996.

[26] E. G. Coffman, *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, NY, 1976.

[27] F. D. Croce, R. Tadei, and G. Volta, "A genetic algorithm for the job shop problem," *Computers & Operations Research*, vol. 22, no. 1, pp. 15–24, 1995, genetic Algorithms.

[28] E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin, "An efficient synchronization mechanism for mirrored game architectures," *Multimedia Tools Applications*, vol. 23, no. 1, pp. 7–30, 2004.

[29] G. Deen, M. Hammer, J. Bethencourt, I. Eiron, J. Thomas, and J. H. Kaufman, "Running quake ii on a grid," *IBM Systems Journal*, vol. 45, no. 1, pp. 21–44, 2006.

[30] M. K. Dhodhi, I. Ahmad, and A. Yatama, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338–1361, Sep. 2002.

[31] Q. Ding and G. Chen, "A benefit function mapping heuristic for a class of meta-tasks in grid environments," in *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, May 2001, p. 654.

[32] A. Dogan and F. Ozguner, "Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systems," *Cluster Computing*, vol. 7, no. 2, pp. 177–190, Apr. 2004.

[33] M. M. Eshaghian, *Heterogeneous Computing.* Artech House, Norwood, MA, 1996.

[34] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transaction on Software Engineering*, vol. SE-15, no. 11, pp. 1427–1436, Nov. 1989.

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

[36] R. F. Freund and H. J. Siegel, "Heterogeneous processing," *IEEE Computer*, vol. 26, no. 6, pp. 13–17, Jun. 1993.

[37] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, Jun. 1993.

[38] S. Ghanbari and M. R. Meybodi, "Learning automata based algorithms for mapping of a class of independent tasks over highly heterogeneous grids," in *European Grid Conference (EGC '05)*, Feb. 2005, pp. 681–690.

[39] ——, "On-line mapping algorithms in highly heterogeneous computational grids: A learning automata approach," in *International Conference on Information and Knowledge Technology (IKT '05)*, May 2005.

[40] T. Hagras and J.Janecek, "A high performance, low complexity algorithm for compile time job scheduling in homogeneous computing environments," *Parallel Computing*, vol. 31, no. 7, pp. 653–670, Jul. 2005.

[41] A. Hertz and D. de Werra, "The tabu search metaheurstic: How we used it," *Annals of Mathematics and Artificial Intelligence*, vol. 1, no. 1–4, pp. 111–121, Sep. 1990.

[42] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

[43] T. Iimura, H. Hazeyama, and Y. Kadobayashi, "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games," in *3rd ACM SIGCOMM workshop on Network and System Support for Games*, Aug. 2004, pp. 116–120.

[44] Z. Jinquan, N. Lina, and J. Changjun, "A heuristic scheduling strategy for independent tasks on grid," in *Eighth International Conference on High-Performance Computing in Asia-Pacific Region 2005*, Nov. 2005.

[45] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann, "Addressing cheating in distributed mmogs," in *4th ACM SIGCOMM Workshop on Network and System Support for Games*, 2005.

[46] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, Jul. 1998.

[47] K. Kaya, B. Ucar, and C. Aykanat, "Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, pp. 271–285, Mar. 2007.

[48] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*, Nov. 1995, pp. 1942–1948.

[49] S. U. Khan and I. Ahmad, "A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids," *IEEE*

TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, vol. 20, no. 3, Mar. 2009.

[50] A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, vol. 26, no. 6, pp. 18–27, Jun. 1993.

[51] J.-K. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann, "Dynamic mapping in energy constrained heterogeneous computing systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, Apr. 2005.

[52] ——, "Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling," *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Power-Aware Parallel and Distributed Systems*, vol. 19, no. 11, pp. 1445–1457, Nov. 2008.

[53] S. Kim, S. Lee, and J. Hahm, "Push-pull: Deterministic search-based DAG scheduling for heterogeneous cluster systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1489–1502, Nov. 2007.

[54] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in *IEEE Conference on Computer Communications (INFOCOM '04)*, Mar. 2004, pp. 96–107.

[55] L. A. Kramer and S. L. Smith, "Maximizing flexibility: A retraction heuristic for oversubscribed scheduling problems," in *Eighteenth International Joint Conference on Artificial Intelligence*, Aug. 2003.

[56] A. Kumar and R. Shorey, "Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 10, pp. 1147–1164, Oct. 1993.

[57] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, May 1996.

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

[59] J. Lee, "Considerations for movement and physics in mmp games," in *Massively Multiplayer Game Development.* Hingham, MA: Charles River Media, Inc., 2003, pp. 275–289.

[60] K.-W. Lee, B.-J. Ko, and S. Calo, "Adaptive server selection for large scale interactive online games," *Computer Networks*, vol. 49, no. 1, pp. 84–102, Sep. 2005.

[61] G. Liu, K. Poh, and M. Xie, "Iterative list scheduling for heterogeneous computing," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 654–665, May 2005.

[62] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," in *Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster, Ed. New York, NY: John Wiley, 1999, vol. 8, pp. 679–690.

[63] E. Machtans, L. Sato, and A. Deppman, "Improvement on scheduling dependent tasks for grid applications," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, aug. 2009, pp. 95 –102.

[64] F. Magouls, J. Pan, K.-A. Tan, and A. Kumar, *Introduction to Grid Computing.* Chapman & Hall/CRC Numerical Analy & Scient Comp. Series, 2009.

[65] 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," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–121, Nov. 1999.

[66] A. M. Mehta, J. Smith, H. J. Siegel, A. A. Maciejewski, A. Jayaseelan, and B. Ye, "Dynamic resource allocation heuristics that manage tradeoff between makespan and

robustness," *Journal of Supercomputing, Special Issue on Grid Technology*, vol. 42, no. 1, pp. 33–58, Jan. 2007.

[67] NESDIS, "National environmental satellite data information service (nesdis)," http://www.nesdis.noaa.gov/About/about.html, Mar. 2006.

[68] M. Oltikar, J. Brateman, J. White, J. Martin, K. Knapp, A. A. Maciejewski, and H. J. Siegel, "Robust resource allocation in weather data processing systems," in *8th Workshop on High Performance Scientific and Engineering Computing (HPSEC 2006)*, Aug. 2006, pp. 445–454.

[69] D. Paranhos, W. Cirne, and F. Brasileiro, "Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids," in *International Conference on Parallel and Distributed Computing*, Aug. 2003.

[70] J. Pugh and A. Martinoli, "Discrete multi-valued particle swarm optimization," in *IEEE Swarm Intelligence Symposium '06*, May 2006, pp. 103–110.

[71] S. Russell and P. Norvig, *Artificial Intelligence a Modern Approach, Second Edition.* Prentice Hall, 2005.

[72] P. SaiRanga and S. Baskiyar, "A low complexity algorithm for dynamic scheduling of independent tasks onto heterogeneous computing systems," in *43rd annual Southeast regional conference - Volume 1*, Mar. 2005, pp. 63–68.

[73] A. Shaikh, S. Sahu, M.-C. Rosu, M. Shea, and D. Saha, "On demand platform for online games," *IBM Systems Journal*, vol. 45, no. 1, pp. 7–20, 2006.

[74] V. Shestak, J. Smith, H. J. Siegel, and A. Maciejewski, "Stochastic robustness metric and its use for static resource allocations," *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1157–1173, Aug. 2008.

[75] V. Shestak, E. K. P. Chong, H. J. Siegel, A. A. Maciejewski, L. Benmohamed, I.-J. Wang, and R. Daley, "A hybrid branch-and-bound and evolutionary approach for

allocating strings of applications to heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 410–426, Apr. 2008.

[76] S. Shivle, 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, and J. Velazco, "Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment," *Journal of Parallel and Distributed Computing, Special Issue on Algorithms for Wireless and Ad-hoc Networks*, vol. 66, no. 4, pp. 600–611, Apr. 2006.

[77] G. Sih and E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, Feb. 1993.

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

[79] J. Smith, V. Shestak, H. J. Siegel, S. Price, L. Teklits, and P. Sugavanam, "Robust resource allocation in a cluster based imaging system," *Parallel Computing*, vol. 35, no. 7, pp. 389–400, Jul. 2009.

[80] P. Sugavanam, H. J. Siegel, A. A. Maciejewski, M. Oltikar, A. Mehta, R. Pichel, A. Horiuchi, V. Shestak, M. Al-Otaibi, Y. Krishnamurthy, S. Ali, J. Zhang, M. Aydin, P. Lee, K. Guru, M. Raskey, and A. Pippin, "Robust static allocation of resources for independent tasks under makespan and dollar cost constraints," *Journal of Parallel and Distributed Computing*, vol. 67, no. 4, pp. 400–416, April 2007.

[81] X. Tanga, K. Li, G. Liao, and R. Li, "List scheduling with duplication for heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 4, pp. 323–329, Apr. 2010.

[82] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[83] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing, Special Issue on Parallel Evolutionary Computing*, vol. 47, no. 1, pp. 8–22, Nov. 1997.

[84] B. Wei, G. Fedak, and F. Cappello, "Scheduling independent tasks sharing large data distributed with bittorrent," in *The 6th IEEE/ACM International Workshop on Grid Computing*, Nov. 2005.

[85] D. Whitley, "The genitor algorithm and selective pressure: Why rank based allocation of reproductive trials is best," in *3rd International Conference on Genetic Algorithms*, Jun. 1989, pp. 116–121.

[86] M. Wu and W. Shu, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," in *9th Heterogeneous Computing Workshop (HCW '00)*, Mar. 2000, pp. 375–385.

[87] M. Wu and D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, Jul. 1990.

[88] D. Xu, K. Nahrstedt, and D. Wichadakul, "Qos and contention-aware multi-resource reservation," *Cluster Computing*, vol. 4, no. 2, pp. 95–107, Apr. 2001.

[89] J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of execution times on heterogeneous supercomputer architectures," in *International Conference on Parallel Processing*, vol. I, Aug. 1993, pp. 219–225.