Dissertation

Highly Scalable Algorithms For Scheduling Tasks and Provisioning Machines on Heterogeneous Computing Systems

Submitted by

Kyle M. Tarplee

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 2015

Abstract

Highly Scalable Algorithms For Scheduling Tasks and Provisioning Machines on Heterogeneous Computing Systems

As high performance computing systems increase in size, new and more efficient algorithms are needed to schedule work on the machines, understand the performance trade-offs inherent in the system, and determine which machines to provision. The extreme scale of these newer systems requires unique task scheduling algorithms that are capable of handling millions of tasks and thousands of machines. A highly scalable scheduling algorithm is developed that computes high quality schedules, especially for large problem sizes. Large-scale computing systems also consume vast amounts of electricity, leading to high operating costs. Through the use of novel resource allocation techniques, system administrators can examine this trade-off space to quantify how much a given performance level will cost in electricity, or see what kind of performance can be expected when given an energy budget. Trading-off energy and makespan is often difficult for companies because it is unclear how each affects the profit. A monetary-based model of high performance computing is presented and a highly scalable algorithm is developed to quickly find the schedule that maximizes the profit per unit time. As more high performance computing needs are being met with cloud computing, algorithms are needed to determine the types of machines that are best suited to a particular workload. An algorithm is designed to find the best set of computing resources to allocate to the workload that takes into account the uncertainty in the task arrival rates, task execution times, and power consumption. Reward rate, cost, failure rate, and power consumption can be optimized, as desired, to optimally trade-off these conflicting objectives.

Acknowledgements

The following research was made possible by numerous people. Thanks to my wife for motivating me to keep making progress on the degree, giving me deadlines, and most of all praying for me. A special thanks to Professor Anthony A. Maciejewski and Professor Howard Jay Siegel for their extremely valuable guidance and insight on a weekly basis. I would also like to thank Ryan Friese, Bhavesh Khemka, Mark Oxley, Tim Hansen, Greg Pfister, Jerry Potter and the other members of the robust computing research group for their suggestions and feedback.

Thanks to my parents, Gary and Debi Tarplee, for always being there for me. I would like to thank my sister-in-law Dr. Allison Tarplee M.D. for her timely encouragement in pursuing this degree. Last but not least I would like to thank my brother Todd for his friendship and helpful discussions about my research.

This work was supported by the Sjostrom Family Scholarship, Numerica Corporation, the National Science Foundation (NSF) under grants CNS-0905399 and CCF-1302693, the NSF Graduate Research Fellowship, and by the Colorado State University George T. Abell Endowment. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

This dissertation is typset in LaTeX using a document class designed by Leif Anderson.

Dedication

To my loving wife and children.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction and Overview

High performance computing systems have continued to grow in computational capacity. This computational growth is often obtained by increasingly larger quantities of machines fitted with fast special purpose coprocessors. There are often tens to hundreds of thousands of machines that compose today's computing systems. The need for these extremely large high-performance computing (HPC) systems is driven by increasingly large HPC workloads composed of potentially millions of tasks. Thus, efficiently scheduling tasks onto machines at such a large scale is becoming more important.

For a variety of reasons, HPC systems are often composed of different types of machines. Machine heterogeneity can be caused by building the HPC system in multiple phases, where each expansion phase involves purchasing a newer/different server model. Heterogeneity might also be introduced into a system from the start to decrease the run time of relatively slow tasks. For example, GPUs and specialized co-processors have been used to greatly accelerate the computation of data parallel tasks [1]. Systems composed of a non-uniform set of compute resources are called heterogeneous computing (HC)) systems. The focus of this research is on HC systems that are heterogeneous in both performance and power consumption. For example, some tasks may execute faster on machines that support a particular CPU instruction set while another set of tasks may execute faster on machines with higher memory IO bandwidth. The energy consumed by a task running on a GPU-enabled machine may be different than when running solely on the CPU. The nature of a task dictates how efficient, in run time and energy, it will perform on any given machine. This task and machine heterogeneity provides additional degrees of freedom that can be leveraged by the scheduling algorithms to create resource management schedules that improve the workload's run time performance and reduce the energy consumption of the overall system.

A novel algorithm for minimum makespan scheduling for very large scale systems is addressed in Chapter 2. Solution quality and scalability results of the proposed algorithms are shown to outperform many best-of-breed scheduling algorithms.

The race for increased performance in HPC systems has resulted in a large increase in the power consumption of these systems [2]. This increase in power consumption can cause degradation in the electrical infrastructure that supports these facilities, as well as increase electricity costs for the operators [3]. The goals of HPC users conflict with the HPC operators in that the users' goal is to finish their workload as quickly as possible. That is, the small energy consumption desired by the system operator and the high system performance desired by the users are conflicting objectives that require the sacrifice of one to improve the other. Balancing the performance needs of the users with energy costs proves difficult without tools designed to help a system administrator choose from among a set of solutions.

A set of efficient and scalable algorithms are proposed in Chapter 3 that build on Chapter 2 that can help system administrators quickly gain insight into the energy and performance trade-off of their HPC systems through the use of intelligent resource allocation. The algorithms proposed have very fast run times, good asymptotic algorithm complexity, and produce schedules that are closer to optimal as the problem size increases. As such, this approach is very well suited to large scale HPC systems.

While minimizing energy consumption and increasing performance is desirable, it is often not the driving factor for decision making within organizations. Often decision makers are driven to directly maximize profit. Chapter 4 builds on Chapter 3 to describe a novel algorithm to efficiently compute a near-optimal maximum profit schedule.

Some HPC users are turning to cloud providers to complete their work due to the potential cost effectiveness and/or ease of use of cloud computing. The ability to provision hardware on-demand from a pre-defined set of different machine types is very powerful. The hardware can be provisioned in such a way as to increase the performance of the particular workload at that time while minimizing costs. When making a hardware provisioning decision not all the information is necessarily available nor is the information perfectly accurate. For example, arrival rates of tasks or the performance of the machines might not be known perfectly. In fact, studies have shown that machines of the same type can vary significantly in performance as discussed in [4–6].

An algorithm to find the number of each type of machine to provision to maximize the performance of the system for processing a user-defined workload is presented in Chapter 5. This algorithm simultaneously optimizes the schedule for the tasks and the number of machines to purchase using multiple conflicting objectives while accounting for the uncertainty in the task arrival rates, performance of the machines, and power consumption.

CHAPTER 2

Makespan Scheduling [1]

2.1. Introduction

Today's HPC systems often have hundreds of thousands of machines. The need for these extremely large HPC systems is driven by increasingly larger HPC workloads comprising potentially millions of tasks. The increase in computational capability of HPC environments can only be maintained if the tasks can be intelligently assigned to machines quickly. Therefore, there is a growing need for efficiently scheduling tasks to machines in such large-scale environments.

Our work considers a common scheduling model where users submit a set of independent tasks known as a *bag-of-tasks* [10]. We assume that the full bag-of-tasks is known a priori [10] (i.e., *static scheduling*), a task can be scheduled to execute on only one machine, and machines may only process one task at a time. The HPC environments of primary interest have highly heterogeneous tasks and machines and are known as HC systems [11].

HC systems often have some special-purpose machines that can perform specific tasks quickly, while other tasks might not be able to run on them. Another cause of heterogeneity is differing computational requirements, input/output bottlenecks, or memory limitations. For instance, a task that runs on a GPU might execute much faster than the same task run on a general-purpose machine. The heterogeneity in execution time of the tasks provides the scheduler with degrees of freedom to greatly decrease the maximum of all the task finishing times, known as the *makespan*, compared to a naïve scheduling algorithm. The makespan is a very common offline scheduling objective [12, 13]. The algorithms in this work can be

[1]This work is under review with co-authors Ryan Friese, Anthony A. Maciejewski, and Howard Jay Siegel [7]. A preliminary version of this work appeared in [8, 9] with the same co-authors.

adapted to online batch mode scheduling algorithms where the makespan is minimized for each batch of tasks. When a new task arrives or a task is removed from the batch because it is now running on a machine, the schedule for the batch of tasks can be recomputed.

Finding the optimal schedule for this static scheduling problem is NP-Hard in general [14]. Therefore we seek to design algorithms that find near-optimal solutions relatively quickly.

In this study, a set of efficient and scalable algorithms are proposed that schedule heterogeneous tasks to a set of heterogeneous machines with the goal of minimizing makespan. These algorithms compute a lower bound using linear programming (LP) and then quickly compute the fully feasible schedule. The algorithms have very small run times, find schedules that have solutions closer to optimal as the problem size increases, and good asymptotic algorithmic complexity. This approach is therefore very well suited to large-scale HPC environments. Often large computing systems are composed of heterogeneous clusters of homogeneous machines. The proposed algorithms decompose naturally into a high level scheduler that determines which cluster should process the task followed by a lower level scheduler per cluster that assigns the task to a particular machine.

In summary the contributions of this chapter are:

(1) the formulation and evaluation of an algorithm that efficiently computes a tight lower bound on the makespan,
(2) the design and evaluation of a recovery algorithm to take the lower bound solution and compute a near-optimal feasible schedule,
(3) a comparison to other heuristic scheduling algorithms, and
(4) an evaluation and analysis of the scaling properties of the proposed algorithms and algorithms from the literature.

The rest of this chapter is organized as follows. First an algorithm for minimum makespan scheduling is presented in Section 2.2. Section 2.3 describes the nominal HC system and workload used for simulations and evaluation. Bounds on the solution quality are provided by the algorithm and are discussed in Section 2.4. In Section 2.5, we compare this algorithm to other heuristic algorithms. The applicability of the algorithm to very large-scale problems is shown in Section 2.6 along with simulation results for very large system configurations. We discuss related work in Section 2.7, and Section 2.8 concludes this study and presents some ideas for future work.

2.2. Algorithm Design

2.2.1. Approach.

The fundamental approach of this work is to apply divisible load theory (DLT) [15, 16] to ease the computational requirements of calculating a solution to the makespan scheduling problem. The technique operates in two steps to calculate the lower and upper bounds on makespan. The first step uses DLT, where we assume a single task is allowed to be divided and scheduled onto any number of machines, to calculate the lower-bound solution. After the lower-bound solution is computed, a two-phase algorithm is used to recover a feasible solution from the infeasible lower-bound solution. The feasible solution will be shown empirically to be a tight upper bound on the optimal makespan.

HC systems often have groups of machines, typically purchased at the same time, that have identical or very similar performance characteristics. This allows one to group these similar machines (for the purposes of analysis) into a unique machine type. Machines belonging to a *machine type* have virtually indistinguishable performance properties with respect to the workload. Machines of the same type may differ vastly in feature sets so long as the performance of the tasks under consideration are not affected. Tasks often exhibit natural groupings as well. Tasks of the same *task type* are often submitted many times to perform statistical simulations and other repetitive jobs. Having groupings for tasks and for machines permits less profiling effort to estimate the run time for each task on each machine.

Traditionally the static scheduling problem is posed as assigning all tasks to all machines. The classic formulation is not well suited for recovering a high quality feasible solution from a relaxation of the problem. The decision variables in the classic formulation are binary valued (a task is assigned or not assigned to a machine), and rounding a real value from the lower bound to a binary value can change the objective significantly. Complicated rounding schemes are necessary to iteratively compute a suitable solution. Rather than addressing the problem of assigning all tasks to all machines, we pose the problem as determining the number of tasks of each type to assign to machines of each type. With this modification, decision variables will be large integers $\gg 1$, resulting in only a small error to the objective function when rounding to the nearest integer. This approximation is most accurate when the number of tasks assigned to each machine type is large. In addition to easing the recovery of the integer solution, another benefit of this formulation is that it is significantly less computationally intensive due to solving the higher level assignment of tasks types to machine types with DLT, before solving the fine-grain assignment of individual tasks to machines. As such, this approach can be thought of as a hierarchical solution to the static scheduling problem.

2.2.2. Lower Bound. The lower bound on the makespan is given by the solution to an LP problem and is formulated as follows. Let there be T task types and M machine types. Let T_i be the number of tasks of type i and M_j be the number of machines of type j. Let μ_{ij} be the number of tasks of type i assigned to machine type j, where $\mu_{ij} \in \mathbb{R}$ is the primary decision variable in the optimization problem. Let **ETC** be a $T \times M$ matrix where ETC_{ij}

is the *estimated time to compute* a task of type i on a machine of type j. The **ETC** matrix is frequently used in scheduling algorithms (e.g., [10, 17–20]). **ETC** is generally obtained from historical data in real environments.

The lower bound on the finishing time of the machines of a given type is found by allowing tasks assigned to a machine type to be divided among all machines to ensure the minimal finishing time. With this conservative approximation, all machines of type j finish at the same time. The finishing time of all machines of type j for divisible tasks, denoted by F_j, is given by

$$F_j = \frac{1}{M_j} \sum_i \mu_{ij} ETC_{ij} \quad . \tag{1}$$

Throughout this chapter, sums over i always go from 1 to T and sums over j always go from 1 to M, thus the ranges are omitted. Given that F_j is a lower bound on the finishing time for a machine type, the tightest lower bound on the makespan is

$$MS_{LB} = \max_j F_j \quad . \tag{2}$$

The resulting optimization problem for the lower bound is:

$$\begin{aligned} &\underset{\boldsymbol{\mu},\, MS_{LB}}{\text{minimize}} && MS_{LB} \\ &\text{subject to:} \quad \forall i && \textstyle\sum_j \mu_{ij} = T_i \\ &\qquad\qquad\quad\ \forall j && F_j \le MS_{LB} \\ &\qquad\qquad\quad\ \forall i,j && \mu_{ij} \ge 0 \quad . \end{aligned} \tag{3}$$

The objective of Equation (3) is to minimize MS_{LB}, where $\boldsymbol{\mu}$ is the primary decision variable. MS_{LB} is an auxiliary decision variable necessary to model the objective function in Equation (2). The first constraint ensures that all tasks in the bag-of-tasks are assigned to machine types. The second constraint is the makespan constraint. Because the objective is to minimize makespan, the MS_{LB} variable will be equal to the maximum finishing time of all the machine types. The third constraint ensures that there are no negative assignments in the solutions.

Ideally, this LP problem would be solved optimally with $\mu_{ij} \in \mathbb{Z}_{\geq 0}$. However, for practical scheduling problems, finding the optimal integer solution is often not possible due to the high computational cost. Fortunately, efficient algorithms exist that produce high quality sub-optimal feasible solutions. The next few sections describe how we take an infeasible real-valued solution from the linear program and build a complete feasible allocation.

2.2.3. Recovery Algorithm.

2.2.3.1. *Overview.* An algorithm is necessary to recover a feasible solution (or full resource allocation) from the infeasible solution obtained from the lower bound in Equation (3). Numerous approaches have been proposed in the literature for solving integer LP problems by first relaxing them to real-valued LP problems [21]. Our approach here follows this common technique except using computationally inexpensive algorithms tailored to this particular optimization problem. The recovery algorithm is decomposed into two phases. The first phase rounds the solution while taking care to maintain feasibility of Equation (3). The second phase assigns tasks to actual machines to build the full resource allocation. The next two sections detail the two phases of this recovery algorithm.

2.2.3.2. *Rounding.* Let the optimal real-valued solution from Equation (3) be $\boldsymbol{\mu}^*$. Due to the nature of the problem, $\boldsymbol{\mu}^*$ often has few non-zero elements per row, thus requiring the rounding of only a few elements. Usually all the tasks of one type will be assigned to a small number of machine types. In the original scheduling problem, tasks are not divisible, therefore this solution needs to be converted to a solution with an integer number of tasks to assign to each machine type. The following algorithm finds $\hat{\mu}_{ij} \in \mathbb{Z}_{\geq 0}$ such that it is near μ^*_{ij} while maintaining the task assignment constraint. Recall that the task assignment constraint requires the sum of the elements in a row of $\boldsymbol{\mu}^*$ be equal to T_i, an integer. Finding an integer solution near the original solution is important because it will make for a tighter bound on the objective. Algorithm 1 finds $\hat{\boldsymbol{\mu}}$ that minimizes $\sum_j |\hat{\mu}_{ij} - \mu^*_{ij}|$ for a given i.

Algorithm 1 Round to the nearest integer solution while maintaining the constraints

```
1: for $i = 1$ to $T$ do
2:     $n \leftarrow T_i - \sum_j \lfloor \mu^*_{ij} \rfloor$
3:     $\forall j \quad f_j \leftarrow \mu^*_{ij} - \lfloor \mu^*_{ij} \rfloor$
4:     Let set $K$ be the indices of the $n$ largest $f_j$
5:     $\forall j \quad \hat{\mu}_{ij} \leftarrow \begin{cases} \lceil \mu^*_{ij} \rceil, & j \in K \\ \lfloor \mu^*_{ij} \rfloor, & \text{otherwise} \end{cases}$
6: end for
```

Algorithm 1 operates on each row (i.e. task type) of $\boldsymbol{\mu}^*$ independently. The variable n is the number of assignments in a row that must be rounded up to satisfy the task assignment constraint. Let f_j be the fractional part of the number of tasks (of type i) that are assigned to machine type j. The algorithm rounds up (ceiling operator) those n assignments that have the largest fractional parts, and all other fractional assignments are rounded down (floor operator). The result is an integer solution $\hat{\boldsymbol{\mu}}$ that still assigns all tasks properly and is close to the lower-bound solution. Algorithm 1 minimizes the L_1 norm between the integer solution and the real-valued solution. This algorithm chooses n entries to round up that will introduce the least error per entry and thus the least overall error in the L_1 norm sense because the L_1 norm is separable.

To illustrate the behavior of the algorithm, let the input $\boldsymbol{\mu}^*$ be given by Equation (4). The values in bold indicate assignments that are to be rounded up. The output $\hat{\boldsymbol{\mu}}$ of the algorithm is given in Equation (5). The first row has $n = 0$, thus does not need to be rounded. The second row has $n = 1$, thus rounds up 9.6 because $0.6 \geq 0.4$ and rounds every other component down. The third row also has $n = 1$ but shows that the algorithm does not perform traditional rounding because it rounds up 11.4 due to $0.4 \geq 0.3$. The last row shows how the algorithm would round up two values when $n = 2$.

$$\boldsymbol{\mu}^* = \begin{pmatrix} 3 & 0 & 9 & 11 & 0 & 0 \\ 3 & 0 & \mathbf{9.6} & 11.4 & 0 & 0 \\ 3 & 15.3 & 9.3 & \mathbf{11.4} & 0 & 0 \\ 3 & 15.2 & \mathbf{9.9} & \mathbf{11.4} & 2.3 & 4.2 \end{pmatrix} \tag{4}$$

$$\hat{\boldsymbol{\mu}} = \begin{pmatrix} 3 & 0 & 9 & 11 & 0 & 0 \\ 3 & 0 & \mathbf{10} & 11 & 0 & 0 \\ 3 & 15 & 9 & \mathbf{12} & 0 & 0 \\ 3 & 15 & \mathbf{10} & \mathbf{12} & 2 & 4 \end{pmatrix} \tag{5}$$

The makespan computed from the integer solutions produced by Algorithm 1 may still not be realizable, even though an integer number of tasks are assigned to each machine type. To obtain the makespan of the integer solution, computed similarly to Equation (2) as $\max_j \frac{1}{M_j} \sum_i \hat{\mu}_{ij} ETC_{ij}$, one might still be forced to split tasks among machines of a given machine type to force the finishing times of all the machines to be the same. Having a schedule with a fraction of a task assigned to a machine is not a feasible allocation. Figure 2.1

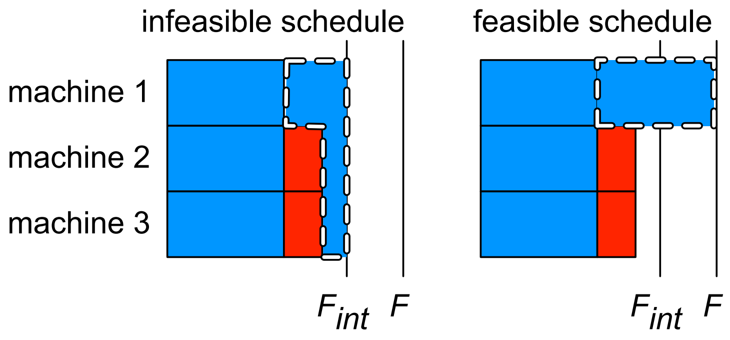


FIGURE 2.1. For any given machine type, even though there are an integer number of tasks of each type (blue and red task types) the lower-bound finishing time of the integer solution, F_{int}, may not be equal to the true finishing time, because the last blue (dashed outline) task on machine 1 would be divided.

shows an example where four blue tasks and two red tasks are assigned to three machines of the same machine type. Even with an integer number of tasks assigned to the machines, the makespan is still larger than the lower-bound on the finishing time of the integer solution, F_{int}, shown in the figure, because the last blue task (dashed outline) would be divided. In the next subsection we explain our local assignment algorithm that will remedy this by forcing each task to be wholly assigned to a single machine.

2.2.3.3. *Local Assignment.* The last phase in recovering a feasible assignment solution schedules the tasks, already assigned to each machine type, to specific machines within that group of machines. This scheduling problem is much easier than the general, heterogeneous, case because the execution characteristics of all machines in a group are the same. This problem is formally known as the multiprocessor scheduling problem [22]. One must schedule a set of heterogeneous tasks onto a set of identical machines. The longest processing time (LPT) algorithm is commonly used for solving the multiprocessor scheduling problem [22]. Algorithm 2 uses the LPT algorithm to independently schedule each machine type.

Algorithm 2 Assign tasks to machines using LPT algorithm for each machine type

```
1: for $j = 1$ to $M$ do
2:     Let $z$ be an empty list
3:     for $i = 1$ to $T$ do
4:         $z \leftarrow$ join($z$, (task type $i$ replicated $\hat{\mu}_{ij}$ times))
5:     end for
6:     $y \leftarrow$ sort descending by ETC(z)
7:     for $k = 1$ to $\| y \|$ do
8:         assign task $y_k$ to the earliest ready time machine of type $j$
9:         update ready time
10:    end for
11: end for
```

Each column (i.e., machine type) of $\hat{\boldsymbol{\mu}}$ is processed independently. List z contains $\hat{\mu}_{ij}$ tasks for each task type i. The tasks are then sorted in descending order by execution time. Next the algorithm loops over this sorted list one task at a time and assigns the task to the machine that has the earliest ready time. The *ready time* of a machine is the time at which all tasks assigned to it will complete. This heuristic packs the largest tasks first in a greedy manner. The body of the outer loop of Algorithm 2 can be thought of as scheduling heterogeneous tasks onto a homogeneous cluster of machines. For environments where the identical machines are arranged in distinct clusters of homogeneous machines, this scheduling would likely be performed by the lower level cluster schedulers.

Algorithms exist that will produce better solutions, but it will be shown that the effect of the sub-optimality of this algorithm on the overall performance diminishes as the problem size becomes large. The makespan of this feasible solution is an upper bound on the optimal makespan. The quality of these solutions is evaluated in Section 2.4.

2.3. Simulation Setup

An **ETC** matrix is needed to evaluate the algorithms. To generate this matrix a set of five benchmarks executed over nine machine types were used to construct the initial matrices [23]. Then the method found in [24] was used to construct a larger **ETC** matrix. Nominally there are 1,100 tasks composed of 30 task types. The number of tasks per task type varies from 11 to 75 and was generated by the method used in [24]. There are nine machine types with four machines of each type for a total of 36 machines. This environment will be referred to as the nine machine type environment. For a complete description of this environments see the supplementary material. This environment was chosen to highlight key aspects of the algorithms. The simulations where executed for 200 Monte Carlo trials unless otherwise noted. In Section 2.6 the size of the environment will be scaled up considerably to show the efficiency of the proposed algorithm.

The simulations were performed on an Apple MacBook Pro Mid 2014, 2.2 GHz Intel Core i7. The software is single threaded so timing results are for one core. All the algorithms were implemented in C++ and optimized using our best effort. The COIN-OR CLP solver was used to solve the LP problems. The third party CLP library is open source and written in C++.

2.4. Minimum Makespan Quality Bounds

2.4.1. Introduction. In this section, we empirically evaluate the tightness of the bounds computed by the minimum makespan scheduling algorithm described in Section 2.2 (henceforth referred to as LP-makespan). The lower bound is compared to an alternative lower bound based on minimum execution time (MET) for each task. The lower and upper bounds are compared to each other to show how small the margin for improvement is in the solution quality of LP-makespan. Lastly, we compare the run times of the three phases of the algorithm. The nine machine type environment is used for this set of simulations. These simulations vary the number of tasks to show the scaling trends. The bag-of-tasks is generated by sampling with replacement from the original task type distribution. All other parameters remain unchanged.

2.4.2. MET Lower Bound Comparison. One lower bound on makespan used in the literature is found by assigning each task to its MET machine and assuming all machines are equal to that task's MET machine. The bound can be thought of as processing each task sequentially by distributing a task over all machines assuming all the machines are identical to the MET machine for that task. This is a lower bound because not all machines will be the MET machine for a given task type. This lower bound is feasible when machines are homogeneous and the number of tasks is a multiple of the number of machines. The MET lower bound is given by

$$MS_{LB}^{MET} = \frac{\sum_i T_i \min_j ETC_{ij}}{\sum_j M_j} \quad . \tag{6}$$

Figure 2.2 shows the MET-based lower bound alongside the LP-makespan lower bound. The width of the glyphs represent the normalized sample probability density of the makespan. In statistics, these are referred to as relative frequency distributions [25]. The wider the glyph the more probability density that exists at that value for makespan. The glyphs are offset in the x-axis; however, they correspond to the same number of tasks for each lower bound shown. The LP-makespan lower bound is much tighter (i.e., larger makespan).

2.4.3. Upper and Lower Bound Tightness. LP-makespan produces upper and lower bounds that can be used to determine how much improvement in makespan is theoretically possible. The feasible schedule's makespan cannot be smaller than the LP-based lower bound. This lower bound is only achievable when the optimal schedule has no machine idle

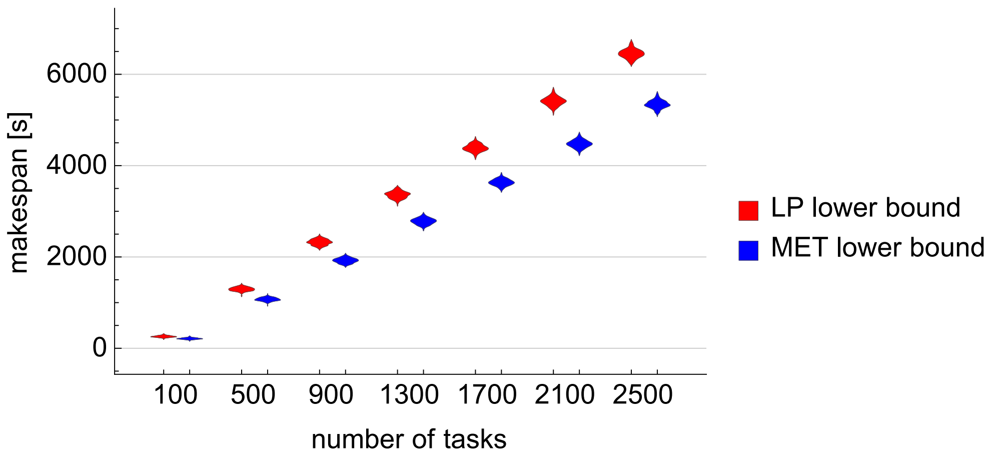


Figure 2.2. Distributions of lower bounds from the LP-makespan and MET algorithms: The shape of the glyphs in this figure show the probability density of different y-axis values for a given x-axis value. The broader the shape, the higher the probability at that y-axis value. LP-makespan lower bound is much tighter than the MET-based lower bound.

for any length of time. Figure 2.3 shows the probability distributions of the percent increase in the upper bound's makespan compared to the lower bound as the number of tasks to be executed increases. The gap between the upper and lower bound decreases as the number of tasks increase because the lower bound becomes tighter as the constraint of task indivisibility has less of an effect. The variance in the gap also decreases as the number of tasks increase. On average, only a 1.8 % improvement might be possible in the LP-makespan algorithm at 2,500 tasks. It is hard to determine where the optimal makespan lies within the lower and upper bounds because it is extremely computationally expensive to compute.

2.4.4. Run Times for the Algorithm Phases.

Figure 2.4 shows the probability distributions of the run times of the three phases of the LP-makespan algorithm. The number of tasks is varied to show the dependence on that parameter. The plot is logarithmic in the time axis because the run times of the rounding and local assignment are much shorter than the time required to find the lower bound for these small problem sizes. Only the local assignment has a strong dependence on the number of tasks to be scheduled. The run time

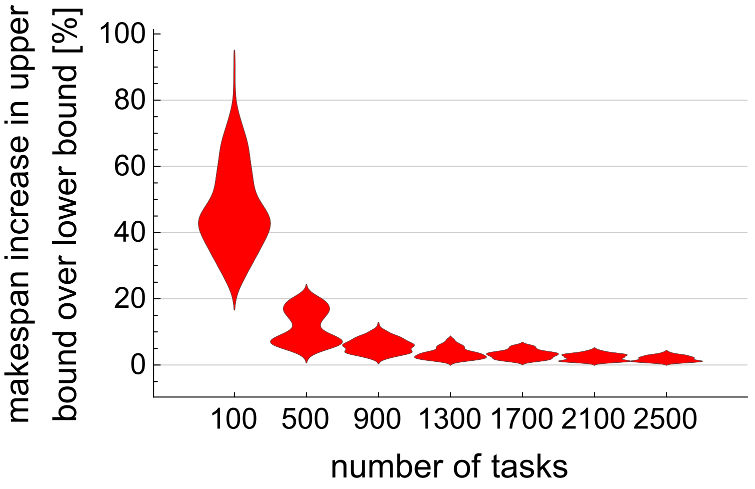


FIGURE 2.3. Distribution of the percent change in the LP-makespan upper bound relative to the LP-makespan lower bound: The room for improvement in the LP-makespan algorithm is less than a few percent as the number of tasks grows large for this particular environment.

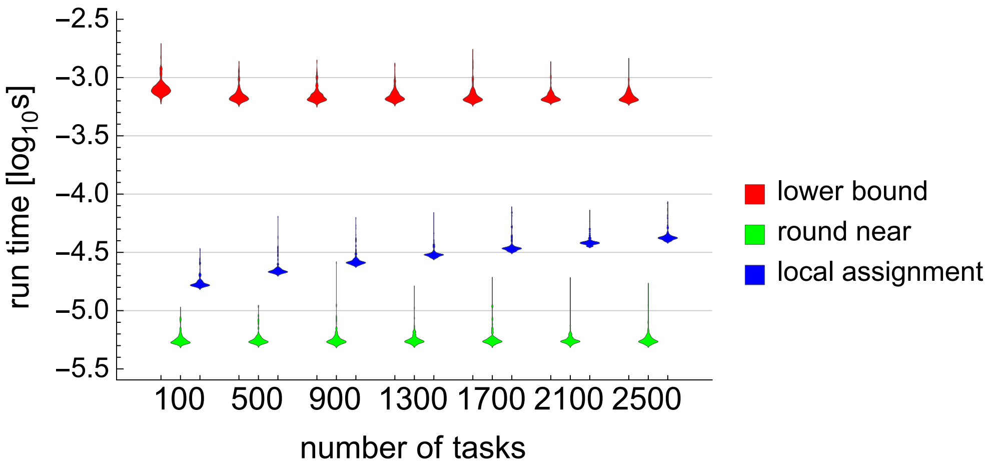


FIGURE 2.4. Distributions of the logarithm of the run time for the three phases of the LP-based algorithm when varying the number of tasks: Lower-bound algorithm and rounding algorithm are not strongly dependent on the number of tasks. The local assignment algorithm run time is linear in the number of tasks. The lower-bound algorithm dominates the run times for the size of problems considered and takes a few milliseconds to complete.

of all the phases of the algorithm is reasonably small, taking only a few milliseconds to complete for this HPC environment.

2.5. Heuristic Algorithms Comparison

2.5.1. Overview. It has been shown that the min-min and max-min algorithms are effective heuristics for minimizing makespan within a reasonable amount of computation time for heterogeneous computing systems [10, 26]. The solution quality, run time, and scalability of both these heuristic algorithms and the LP-makespan algorithm will be analyzed in this section.

2.5.2. Classical Algorithm. The min-min and max-min algorithms are described in [10]. The max-min algorithm, to be described later, is a variant of the min-min algorithm. In the classical min-min algorithm, there is no assumption that one has groups of task types and machine types [27]. Algorithm 3 is the min-min algorithm designed without any regard to task and machine groups. Let t_{type} and m_{type} be the types of task t and machine m, respectively. Let the ready time of machine m be given by rt_m. The min-min algorithm iteratively assigns the task with the minimum completion time to that task's minimum completion time machine.

Algorithm 3 Classic min-min algorithm

```
1: $U$ = set of all tasks from all task types
2: $\forall m \quad rt_m = 0$
3: while $U \neq \emptyset$ do
4:     for $t$ in $U$ do
5:         $mct_t \leftarrow \min_m \left(rt_m + ETC_{t_{\text{type}}\, m_{\text{type}}}\right)$
6:         $m_t \leftarrow \arg\min_m \left(rt_m + ETC_{t_{\text{type}}\, m_{\text{type}}}\right)$
7:     end for
8:     $t^* \leftarrow \arg\min_t mct_t$
9:     $m^* \leftarrow m_{t^*}$
10:    assign task $t^*$ to machine $m^*$
11:    $rt_{m^*} \leftarrow mct_{t^*}$
12:    $U \leftarrow U \setminus t^*$
13: end while
```

Algorithm 3 starts with a set of tasks U and sets the the ready times of all machines to zero. This algorithm then loops over all the tasks. Each iteration computes the minimum completion time, mct_t for each task t and records the minimum completion time (MCT) machine as m_t. The overall minimal MCT pair (t^*, m^*) is chosen for assignment. Lastly, the ready time of the assigned machines and the set of unassigned tasks U are updated.

2.5.3. Optimized Algorithm. The classic min-min algorithm described in Algorithm 3 is not optimized with respect to run time and scalability for our problem formulation. To provide fairer run time and scalability comparisons to the LP-based algorithm, some implementation improvements to the min-min algorithm are desirable. Most of the improvements to the classic min-min algorithm are algorithmic and are used to reduce the computational complexity of the optimized min-min algorithm. Some of the improvements are implementation improvements that are known best practices and have been empirically shown to improve the performance of the algorithm. The classic and the optimized algorithms produce identical output thus only the optimized min-min algorithm will be used for comparison. The outline of the improvements to Algorithm 3 is:

(1) The outer minimization step is computed on the fly keeping track of the current best overall MCT task-machine pair.
(2) Groups of tasks and groups of machines are used to reduce the complexity where possible.
(3) A data structure containing the best machine for each task type is maintained to avoid recomputing the best match.
(4) The task type entry is purged from the list when there are no tasks of that type left to be assigned.
(5) Parameters and return values are counts of tasks instead of lists of tasks or task types.

Computing the new best minimum MCT task and machine pair at each iteration of the outer loop of the algorithm is a minor optimization. Each task of the same type has the same execution time properties, thus, when computing a task's best match the algorithm only needs to consider each task type and not each individual task. This computation to find the best candidate match for each task type need not be recomputed if that task type's MCT machine was not assigned the task on the last iteration. Thus, the optimized algorithm stores each task types's best match and removes the match if that machine was assigned in the last iteration. The algorithm also stores the task type list in such a way that the task type entries that have no more tasks to be assigned can be quickly and safely removed from the list to reduce overhead of subsequent iterations. An important improvement in the algorithm was to remove a large amount of dynamic memory allocation in terms of both number of allocations and size of the allocations, for the function parameters and returned schedule. The parameter that described the bag-of-tasks could easily be implemented as a list of tasks to be assigned. This has the downside of requiring a huge amount of storage when scheduling a large number of tasks. Instead an array of length T that contains the number of tasks of each type is used to describe the bag-of-tasks. The mapping of a particular task to a particular machine is irrelevant when that task has the same run time characteristics as all other tasks of the same type. All that is relevant is the number of tasks of type i that are assigned to a machine. As such, no more information than necessary is computed, which further improves the performance. The resultant task assignments are also stored in a single dense ragged (i.e., irregular) [28] array of integers where the first dimension is of size T, the second dimension is of size M, and the last is of size M_j. The entries of this array, denoted y_{ijk}, are the number of tasks of type i assigned to machine type j, machine k. Algorithm 4 incorporates all of these improvements into the min-min algorithm.

Algorithm 4 Optimized min-min algorithm

```
Require: G: set of all task types
 1: $\forall i \quad n_i$ = number of tasks of type $i$ in G
 2: $\forall j,k \quad rt_{jk} = 0$
 3: $prior = \emptyset$
 4: $\forall i,j,k \quad y_{i,j,k} = 0$
 5: while $G \neq \emptyset$ do
 6:     for $i$ in $G$ do
 7:         if $best_i = prior$ then
 8:             $best_i \leftarrow \arg\min_{j,k} \; (rt_{jk} + ETC_{ij})$
 9:         end if
10:         $j, k \leftarrow best_i$
11:         if $rt_{jk} + ETC_{ij} < rt_{j^*k^*} + ETC_{i^*j^*}$ then
12:             $i^*, j^*, k^* \leftarrow i, j, k$
13:         end if
14:     end for
15:     $y_{i^*j^*k^*} \leftarrow y_{i^*j^*k^*} + 1$
16:     $rt_{j^*k^*} \leftarrow rt_{j^*k^*} + ETC_{i^*j^*}$
17:     $n_{i^*} \leftarrow n_{i^*} - 1$
18:     if $n_{i^*} = 0$ then
19:         $G \leftarrow G \setminus i^*$
20:     end if
21:     $prior \leftarrow (j^*, k^*)$
22: end while
23: return $\mathbf{y}$
```

The set G in Algorithm 4 is an array of task type entries. The outer loop of Algorithm 4 iterates exactly as many times as there are tasks (line 5), similar to Algorithm 3. The inner loop processes one task type i per iteration and recomputes the best match only if the last task assignment iteration assigned a task to the best machine for task type i (lines 7-9). As the loop iterates it also maintains the overall minimum completion time task machine tuple as (i^*, j^*, k^*) (lines 10-13). Once the loop completes, the best task-machine tuple is used to update the result y_{ijk}, ready times rt_{jk}, and the remaining number of tasks for the currently considered task type n_i (lines 14-17). If the remaining number of tasks for that task type is zero then the task type is removed from the list G (lines 18-20). Lastly, *prior* is set to the machine type and machine pair to which the most recent assignment was made (line 21) to be used for invalidating the saved best task assignments. The assignment stored in $\mathbf{y}$, is returned as a ragged array.

The max-min algorithm is very closely related to the min-min algorithm. To convert Algorithm 3 from the min-min algorithm into the max-min algorithm, the min operator on line 8 is changed to the max operator. Algorithm 4 can be converted to the max-min algorithm by reversing the inequality on line 11.

Algorithm 4 is significantly faster than Algorithm 3, especially as the number of tasks becomes large. The complexity of Algorithm 3 is quadratic in the total number of tasks because both the inner and outer loop effectively iterate over all tasks. Algorithm 4 is only linear in the total number of tasks because the inner loop only iterates over task types.

2.5.4. Results. Figure 2.5 shows the makespan of the min-min and max-min compared to the makespan of the LP-makespan algorithm for the nine machine type environment. For all but small numbers of tasks the LP-makespan algorithm produces a shorter makespan. For large numbers of tasks, the min-min algorithm produces on average a 13 % longer makespan than LP-makespan for this particular HPC environment. Max-min performed even worse as the number of tasks become large, producing schedules that are on average 26 % longer than LP-makespan. The LP-makespan algorithm outperforms both heuristics for large problem sizes because it solves a global optimization problem for the relaxation allowing it to make very complex decisions about the allocation to directly minimize makespan. The heuristics only indirectly minimize makespan. When the problem size is small, the task divisibility modeling assumption breaks down leading to poor performance from the LP-makespan algorithm. The variance of the relative makespan distribution is very large for small numbers of tasks, however, the variance decreases rapidly as the number of tasks become larger.

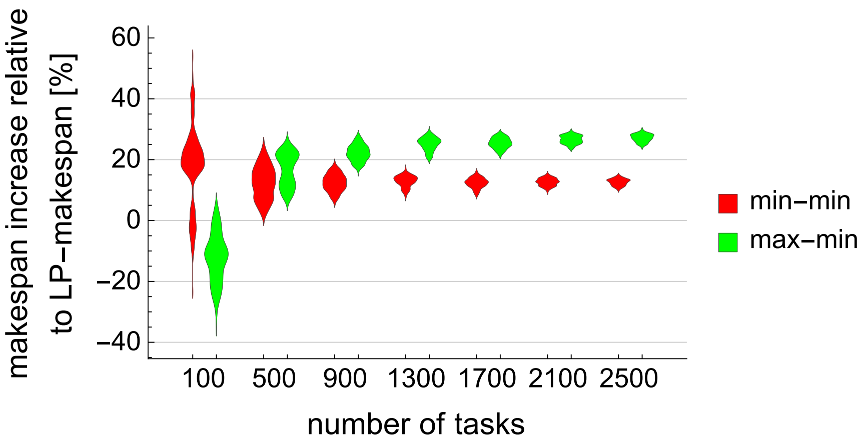


FIGURE 2.5. Distributions of the makespan from min-min and max-min relative to LP-makespan as the number of tasks varies: The LP-makespan algorithm produces better schedules for sufficiently large number of tasks.

The run time comparison between min-min using Algorithm 4 and LP-makespan is shown in Figure 2.6 for the nine machine type environment. Min-min is linearly dependent on the number of tasks, while the LP-makespan algorithm has a fixed run time cost to solve the LP problem but nearly no increase thereafter. LP-makespan is slightly slower than the heuristic algorithms for less than 1,300 tasks, but faster for larger numbers of tasks. The max-min algorithm differs from the min-min algorithm in the orientation of a single inequality operator yet its run time is measurably worse. The difference lies in the effectiveness of storing the MCT machine for each task. This storage is invalidate when the machine is this task's MCT machine that was assigned a task in the previous iteration. For min-min 70 % were valid whereas for max-min only 60 % of the reads were valid. This means that the expensive operation of computing the MCT machine for a task type (iterating over all machines of all types) occurs more often for max-min then it does for min-min for this particular environment. When the MCT machine storage is disabled (i.e., the MCT machine is found every iteration), the algorithms have identical run times. There are some environments where max-min will have a higher percentage of valid reads from the MCT

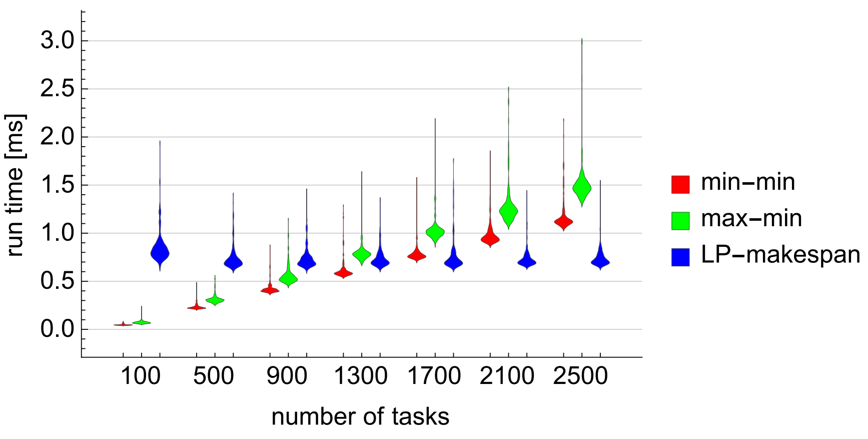
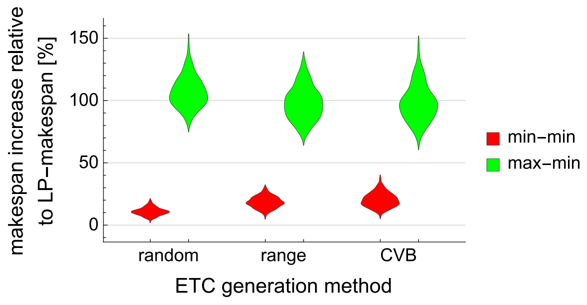


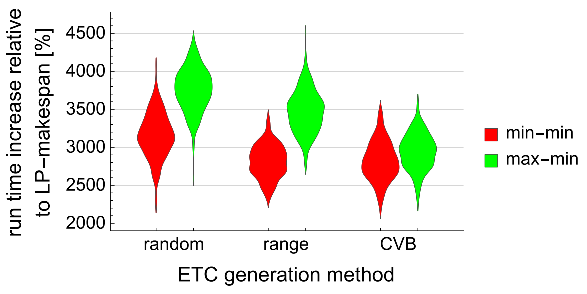
FIGURE 2.6. Distributions of the algorithm run time for min-min, max-min, and LP-makespan as the number of tasks is increased: The LP-makespan algorithm is faster for large numbers of tasks.

machine storage, so this property is not intrinsic to the algorithms but rather a property of the environment.

A set of randomly generated simulation environments are used to compare the min-min and max-min algorithms with the LP-makespan algorithm. There are 15 task types and ten machine types in these systems. One million tasks were used with each task type being equally likely. One thousand machines were used with each machine type being equally likely. Three different methods are used to generate the **ETC** matrix. The "random" method has independent elements that are uniformly distributed from 1 s to 10 s. The "range" method is the range-based method described in [26, 29] with parameters 100 and 10 for tasks and machines respectively. The coefficient of variation (CoV) based method, denoted CVB, is defined in [29] and is based on the gamma distribution. The CoV used for the tasks and machines is 0.6 with a mean of 10 s. Figure 2.7 shows the makespan and run time of the min-min and max-min relative to LP-makespan for 200 different systems for each **ETC** generation method.



(A) makespan



(B) run time

FIGURE 2.7. Distributions of the (a) makespan and (b) run time of min-min and max-min relative to LP-makespan: For each **ETC** generation method, 200 different environments were used. LP-makespan has a smaller makespan in every case and is over 20 times faster.

The LP-makespan algorithm took only 64 ms to schedule one million tasks to one thousand machines in Figure 2.7. For ten million tasks and ten thousand machines the LP-makespan algorithm takes only 0.87 s while the min-min takes over 476 s to produce a schedule who's makespan is longer than LP-makespan.

From Figures 2.5 to 2.7 it can be seen that for large problems the LP-makespan algorithm should be preferred. For the HPC environments under consideration, the LP-makespan algorithm has smaller run times and shorter schedules compared to both the min-min and max-min algorithms.

2.6. Computational Complexity

2.6.1. Analysis.

A complexity analysis of each phase of the LP-makespan algorithm reveals desirable properties. A real-valued LP problem must be solved to compute the lower bound on the makespan. Using the simplex algorithm to solve the LP problem yields exponential complexity (i.e., traversing all the vertices of the polytope) in the worst case; however the average case complexity for a very large class of problems is polynomial time [21]. Recall that there are T task types and M machine types. The lower bound LP problem has $T + M$ nontrivial constraints and $TM + 1$ variables. The average case complexity of computing the lower bound is $(T + M)^2(TM + 1)$. Next is the rounding algorithm. The outer loop iterates T times, and the rounding is dominated by the sorting of M items. Thus the complexity of rounding algorithm defined by Algorithm 1 is $\mathcal{O}\left(T(M \log M)\right)$. The local assignment algorithm defined by Algorithm 2 has an outer loop that is run M times. Inside this loop there are two steps. The first step is sorting at most T items which takes $\mathcal{O}\left(T \log T\right)$ time. The second step is a loop that iterates $n_j = \sum_i \mu_{ij}$ times and finds the machine with the earliest ready time each iteration, a procedure with $\mathcal{O}\left(\log M_j\right)$ complexity. The worst case complexity of local assignment is thus $\mathcal{O}\left(M \max_j \left(T \log T + n_j \log M_j\right)\right)$.

Let $T_{\text{total}} = \sum_i T_i$ be the total number of tasks and $M_{\text{total}} = \sum_j M_j$ be the total number of machines. Assume for the sake of analysis that tasks and machines are evenly distributed across machine types so $n_j \approx \frac{T_{\text{total}}}{M}$ and $M_j \approx \frac{M_{\text{total}}}{M}$. The computational complexity of local assignment can then be written as

$$\begin{aligned} & M \max_j \left(T \log T + n_j \log M_j\right) \\ =& M \max_j \left(T \log T + \frac{T_{\text{total}}}{M} \log \frac{M_{\text{total}}}{M}\right) \\ =& MT \log T + T_{\text{total}} \log \frac{M_{\text{total}}}{M} \\ =& MT \log T + T_{\text{total}} \log M_{\text{total}} - T_{\text{total}} \log M \quad . \end{aligned} \tag{7}$$

The local assignment scales linearly in the number of tasks, T_{total}. The complexity in the number of machine types follows the negative logarithm. The complexity in the number of machines is actually sub-linear.

The complexity of the overall algorithm to find both the lower bound and upper bound (full allocation) is driven by either the lower-bound algorithm or the local assignment algorithm. Complexity of the lower bound and rounding algorithms are independent of the number of tasks and machines. Those algorithms depend only on the number of task types and machine types. This is a very important property for large-scale HPC environments. Very large numbers of tasks and machines can be handled easily if the machines can be reasonably placed in a small number of homogeneous machine types and, likewise, tasks can be grouped by a small number of task types. Only the local assignment algorithm's complexity has a dependence on the number of tasks and machines. This phase is only necessary if a full allocation or schedule is required. The lower bound can be used to analyze much of the behavior of the system at less computational cost. Furthermore, local assignment can be trivially parallelized because each machine type is scheduled independently.

2.6.2. Results. An important property of a scheduling algorithm is its ability to scale well as the size of the problem grows. Simulations were carried out to quantify how the relative error and the computational cost of the algorithm scales. These simulations are used to validate the complexity analysis results from Section 2.6.1. The environment used for this set of simulations is a scaled up version of our typical nine machine type environment. The number of machines was increased to 36,000 and the number of tasks was increased to 1,100,000, still with nine machine types and 30 task types, respectively. The distributions of the task types and machines types remain the same as the nine machine type environment.

The number of tasks, machines, task types, and machine types are varied independently to show the scalability of the LP-makespan algorithm w.r.t. each parameter. For environments this large, it is intractable to solve for the optimal makespan. It is even too expensive to solve the LP relaxation of the assignment of individual tasks to individual machines for this environment. This highlights the need for much more scalable algorithms such as LP-makespan. Even though the optimal solution is not known it is still possible to compare bounds on the makespan to gain insight into the algorithm's solution quality. Each of the parameter sweeps is computed by taking random subsets with replacement to handle the sweep variable. These results are averaged over 50 Monte Carlo trials.

Figure 2.8 shows the relative change in makespan as the number of tasks increase. The number of task types, machines, and machine types are held constant and are the same as the nominal environment. The relative increase in makespan is shown from the makespan lower bound, MS_{LB}, to the makespan after rounding. Also shown is the increase in makespan from the integer solution to the full allocation. The relative increase in makespan from the lower bound to the upper bound or full allocation is also shown. The loss in quality of the makespan from the rounding algorithm is relatively low. Most of the increase in makespan is caused by local assignment. However, Figure 2.8 also shows that the relative increase in makespan diminishes as the number of tasks increase. This is because the approximation that tasks are divisible has less of an impact on the solution as the number of tasks per machine increases. Figure 2.8 shows a cyclical or periodic pattern in the quality of the local assignment algorithm. This pattern is not present in the lower bound or the integer solutions. This pattern is caused by the discrete nature of the problem of assigning tasks to machines. The makespan can increase significantly when just one task is added to the bag-of-tasks that does not pack well onto the machines. Recall that local assignment, by design, only assigns tasks to within a single type of machine so the degrees of freedom are limited in how the algorithm can distribute the load and mitigate the peaks in the relative makespan.

To quantify the computational efficiency of our algorithms, we show the run time of the techniques as a function of the number of tasks in Figure 2.9. Figure 2.9a is the time taken to compute the lower bound (i.e., solve the LP problem). Figure 2.9b shows the time required to round the solution. Both of the computations required to compute the lower bound and the integer solution do not depend on the number of tasks. This corresponds to the results derived for the complexity of the algorithm. Figure 2.9c shows that the local assignment algorithm scales linearly with the number of tasks. This also corresponds to the analysis in Section 2.6.1. Notice that the magnitude of the run times are rather small. Even for 10^8 tasks (not shown in the figure) the total run time is only 8.4 s running on a single core. The LP-makespan algorithm is highly parallelizable so further improvements in runtime could be made if necessary.

The relative increase in makespan when varying the total number of machines is shown in Figure 2.10. The figure shows the same three curves as Figure 2.8, however in this case, varying the total number of machines. The number of tasks, machine types, and task types are held constant. As the number of machines grow, the increase in makespan due to the local assignment step grows rapidly. This is caused by assigning fewer tasks to each machine as the number of machines increases. The approximation that tasks are divisible becomes a worse approximation as the number of machines increases relative to the number of tasks.

Figure 2.11 shows the run time of the three parts of the scheduling algorithm as the total number of machines is varied. Both the lower bound and the rounding are independent of the number of machines. The local assignment step is approximately logarithmic in the number of machines. This corresponds to the analysis in Section 2.6.1.

Figure 2.12 shows the same three curves as Figure 2.8, however in this case varying the number of task types. The number of tasks, machines, and machine types are held constant for this simulation. Figure 2.12 shows that the local assignment algorithm (integer to full allocation) is again causing most of the degradation in makespan. The relative increase

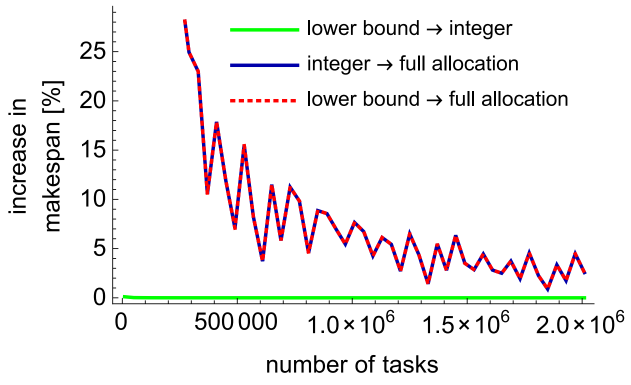
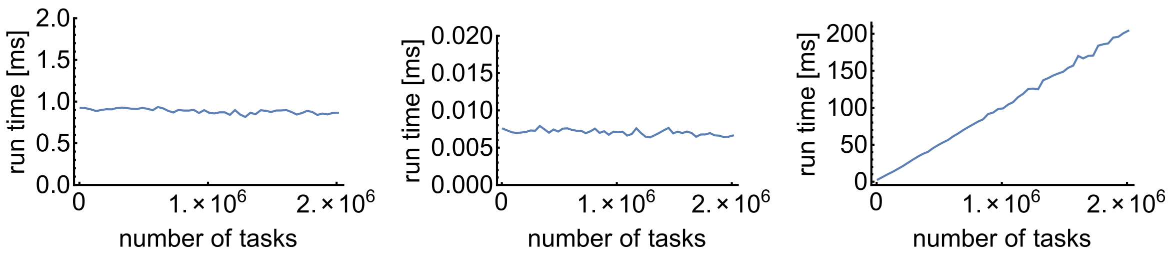


Figure 2.8. Relative percent increase in makespan as a function of the *total number of tasks*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. The relative increase in makespan decreases, thus the quality of the solution improves, as more tasks are used.



(a) lower bound (b) rounding (c) local assignment

Figure 2.9. Algorithm run time versus *total number of tasks*: Both the lower bound and the rounding algorithms run time, (a) and (b) respectively, are independent of the number of tasks. The local assignment complexity (c), used to obtain the full allocation, is linearly dependent on the number of tasks.

in makespan does not tend to zero because increasing the number of task types does not improve the quality of the approximation. LP-makespan still finds a solution that is within just 6 % of optimal.

Figure 2.13 shows the run time of the three phases when varying the number of task types. Here the lower bound has small super linear dependence on the number of task types. According to the complexity analysis, this relationship should be cubic. However, Figure 2.13a does not exhibit such poor scaling behavior. This is likely due to the increase in the sparsity of the constraint matrix as the number of tasks types increase making the LP

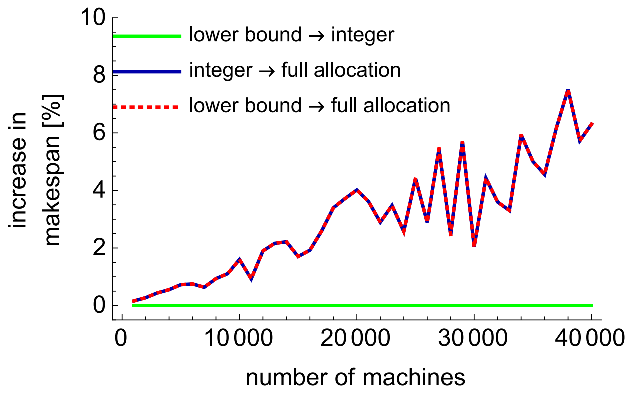


FIGURE 2.10. Relative percent increase in makespan as a function of the *total number of machines*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. The quality of the solution decreases as more machines are used.

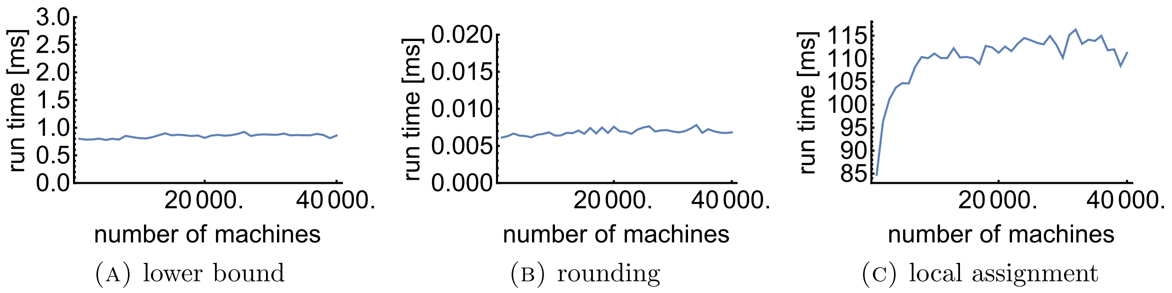


FIGURE 2.11. Algorithm run time versus *total number of machines*: Both the lower bound and the rounding algorithm run times, (a) and (b) respectively, are independent of the number of machines. The local assignment complexity (c), used to obtain the full allocation, is logarithmically dependent on the number of machines.

problem more efficient to solve. The rounding algorithm increases linearly, which matches our complexity analysis. The local assignment phase seems to be linearly dependent on the number of task types. This is close to the analysis that expected a log-linear dependence on the number of task types.

Figure 2.14 shows the relative increase in makespan as the number of machine types varies. In the previous parameter sweeps, the number of tasks of a particular type may be zero if the random sampling selected that configuration. Allowing the number of machines within a machine type to be zero is problematic because some constraint coefficients will

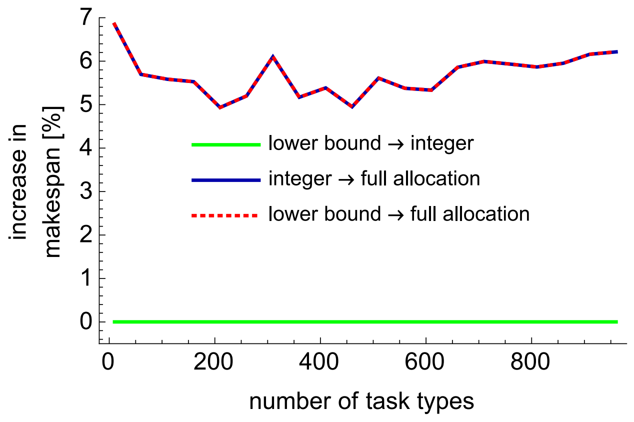
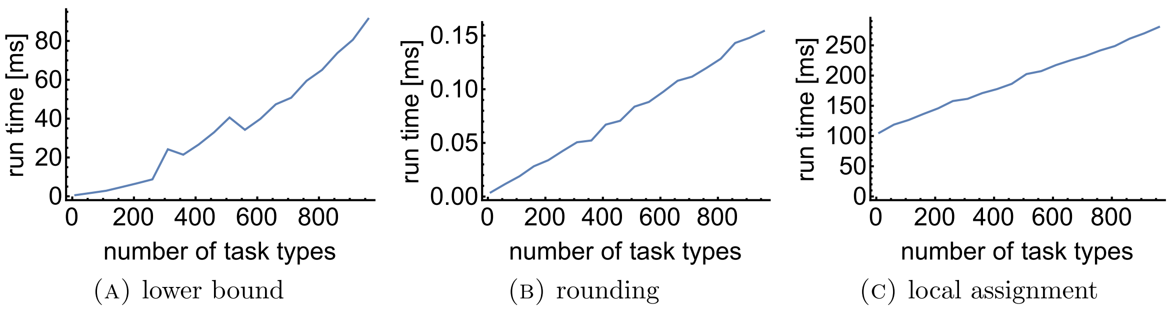


Figure 2.12. Relative percent increase in makespan as a function of the *number of task types*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. Quality of the solutions is tightly bounded and is approximately independent of the number of task types.



(a) lower bound (b) rounding (c) local assignment

Figure 2.13. Algorithm run time as a function of *number of task types*: The complexity of the lower bound algorithm (a) grows super linearly with the number of task types. The rounding and local assignment algorithm run times, (b) and (c) respectively, are linearly dependent on the number of task types.

be ∞ (due to dividing by zero in Equation (1)). Practically, $M_j = 0$ means that the j^{th} column of **ETC** should be removed and the solution should never assign a task to that machine type because it has no machines. To avoid this case, each machine type is forced to have at least one machine to avoid degeneracy. Figure 2.14 also shows that the quality of the rounding algorithm decreases as the number of machine types increase. This is expected because there are less tasks to assign to each machine's type, making the approximation weaker.

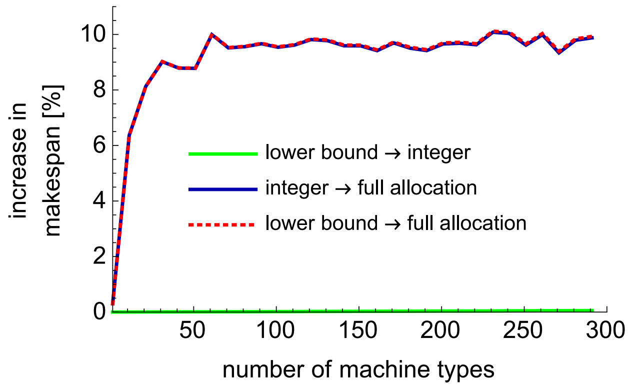


FIGURE 2.14. Relative percent increase in makespan as a function of the *number of machine types*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. Overall performance is approximately independent of the number of machine types.

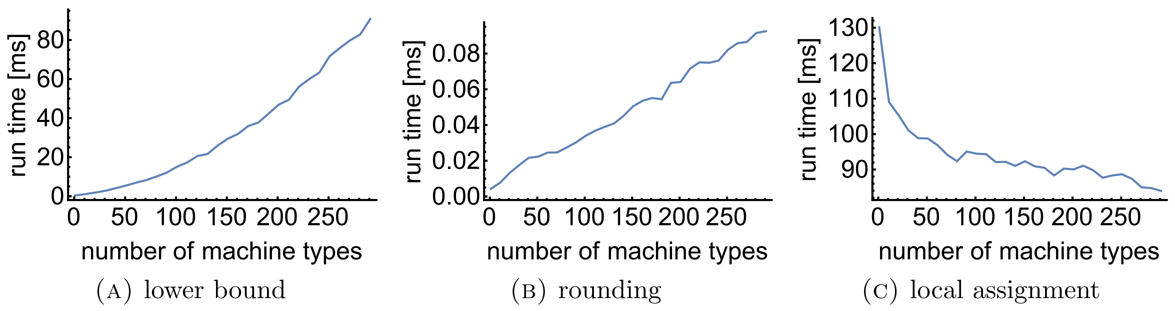


FIGURE 2.15. Algorithm run time versus the *number of machine types*: Lower-bound algorithm complexity (a) is super linear in the number of machines types. The rounding algorithm run time (b) is approximately linear in the number of machine types. Local assignment algorithm run time (c) goes like the negative of the logarithm in the number of machine types.

Figure 2.15 shows the run time as the number of machine types is increased. As expected, the lower bound calculation has an approximately cubic relationship to the number of machine types. The rounding algorithm grows roughly linearly in the number of machine types. As the number of machine types increases, the time spent performing local assignment for each machine type decreases because fewer tasks are scheduled to fewer machines. This matches our analysis in Section 2.6.1.

Even though the run time and solution quality of the polynomial time LP-makespan algorithm is desirable, there is some prior work on theoretical bounds that should be noted. In

[30], it is proven that there exists no polynomial algorithm that can provably find a schedule that is less than 3/2 the optimal makespan, unless $P = NP$. Even though Figures 2.8 to 2.15 suggest that one can do better then 3/2, this is only the case on average.

In summary all three phases of the LP-makespan algorithm have reasonable run times for large problems. The solution quality bounds also show that the solutions are very close to the optimal makespan for sufficiently large problems.

2.7. Related Work

The LP-based approach in this chapter achieves significant decrease in run time and increases in solution quality over prior methods by exploiting properties that are common to static scheduling problems. Our approach takes advantage of the common property that each machine in an HPC system is not unique but belongs to one of a few types of machines. Our work also is focused on very large-scale environments and finding high quality solutions on average, whereas [14, 31] are concerned with worst-case performance of the scheduling algorithms.

Static scheduling for minimum makespan is surveyed in [10]. Min-min and max-min or a hybrid of both algorithms are found to generally be the best algorithms for this problem domain [32]. Our results in Section 2.5 show that min-min almost always performs better than max-min. The max-min tends to perform better then min-min when there are many more short running tasks than long running tasks [26]. The min-min algorithm will schedule the shorter tasks to run on all the machines leaving the fewer long tasks to the end, increasing the makespan. In our simulations there are similar numbers of short and long tasks so the key conceptual benefit of max-min cannot be achieved.

While this chapter deals with scheduling tasks to entire machines, the algorithms could also be applied to scheduling tasks to cores within a machine or across cores on many machines. The full allocation recovery algorithm we use is conceptually similar to the algorithms presented in [33]; however, those algorithms are designed for scheduling tasks on a single machine with deadlines to determine the best dynamic voltage and frequency scaling (DVFS) parameters to use to minimize energy as a secondary objective. Another related algorithm is presented in [34] that approximates makespan to provide computationally efficient schedules while considering reliability for DVFS scheduling on identical processors.

In [20], the A* search algorithm is used to assign tasks to machines considering task dependencies and communication constraints. This algorithm is very expensive for large numbers of tasks because the algorithm's branching factor is on the order of the number of machines and the depth is on the order of the number of tasks.

Allocating services running within virtual machines to physical machines is addressed in [35]. The services being considered are CPU bound processes that are allocated fractions of machines. Multiple smaller services can be allocated to one machine. Their approach is similar to ours in that they formulate a linear program, solve the relaxation, and then recover a feasible solution. The authors note that using binary variables degrades the quality of the solution from the rounding methods used after solving the linear program. We try to address this issue by formulating the linear program to have decision variables that are large values that round easily to large integers, resulting in little degradation in the quality of the solution. They also propose a genetic algorithm (GA) and heuristic algorithms to solve the problem faster and with a higher quality than rounding the result of their linear program. The work in [35] is extended from a single homogeneous set of machines to a heterogeneous collection of machines in [36]. Our work focuses on highly scalable algorithms whereas [35, 36] focus on algorithms that work on relatively small problem sizes and have non-negligible run times for the schedulers.

2.8. Conclusions

A highly scalable scheduling algorithm for computing a near-optimal minimum makespan schedule was presented. The three-phase LP-makespan algorithm was shown to outperform the min-min and max-min heuristics with respect to makespan for larger problem sizes. The LP-makespan has a further benefit in that it produces tight lower and upper bounds on the optimal makespan. Furthermore, the scalability of the LP-makespan algorithm was evaluated to show that a very large number of tasks can be scheduled in a very short amount of time. The complexity of the first two phases of the LP-makespan algorithm are independent of the number of tasks and machines. Only the last, computationally inexpensive and trivially parallelizable, phase is dependent on the number of tasks and machines. The last phase of the algorithm is computed on a per machine type basis, therefore, for very large systems this work can be distributed among lower level schedulers (e.g., each responsible for a cluster of homogeneous machines). The quality of the solution also improves as the size of the problem increases. These scaling properties make this algorithm perfectly suited for very large scheduling problems.

The LP-makespan scheduling algorithm only takes a fraction of a second to compute a single schedule for a given bag-of-tasks so it is possible to use this scheduler for online batch-mode scheduling. Specifically, this algorithm can be used to schedule tasks as they arrive at the system by computing a schedule for all tasks waiting in the queue (as a batch) and recomputing the schedule when a task completes or a new task arrives.

CHAPTER 3

Energy-Aware Scheduling [1]

3.1. Introduction

Today's HPC systems often have hundreds of thousands of cores, processors, and/or machines. The need for these extremely large HPC systems is driven by increasingly large HPC workloads comprising potentially millions of tasks. The increase in computational capability of HPC systems also results in a significant increase in its energy consumption. Therefore, there is a growing need for computationally efficient algorithms for energy-aware scheduling of tasks to machines in such large-scale environments.

HPC systems have seen dramatic increases in their power consumption [2, 38]. This increase in power consumption can increase electricity costs for the operators, cause degradation in the electronic components, and create additional stress on the electrical infrastructure that supports these facilities [3]. Additionally, the goals of HPC users often conflict with the goals of HPC operators. The user's goal is to finish their workload as quickly as possible. Often, this is in conflict with the goal of the system operator to consume less energy, and typically such a situation requires the sacrifice of one of the goals to satisfy the other. To balance the performance and energy costs of the system it is important to provide the system administrator with a tool that provides a set of solutions that trade-off these objectives.

In this study, a set of efficient and scalable algorithms are proposed that can help system administrators quickly gain insight into the energy and performance trade-offs of their HPC system through the use of intelligent resource allocation. The algorithms proposed have very fast run times, good asymptotic algorithm complexity, and produce schedules that are

[1]This work is under review with co-authors Ryan Friese, Anthony A. Maciejewski, and Howard Jay Siegel [37]. A preliminary version of this work appeared in [8, 9] with the same co-authors.

closer to optimal as the problem size increases. This approach is therefore very well suited to large-scale HPC environments.

Our work considers a common scheduling model where users submit a set of independent tasks known as a *bag-of-tasks* [39]. We assume that the full bag-of-tasks is known a priori [10] (i.e., *static scheduling*), a task executes on only one machine, and a machine may only process one task at a time. We study HPC environments that have highly heterogeneous tasks and machines, known as HC systems [11].

HC systems often have some special-purpose machines that can perform specific tasks quickly, while other tasks might not be able to run on them. Another cause of heterogeneity is differing computational capability, input/output bottlenecks, or memory limitations. The machines may further differ in the average power consumed for each task type. Machines can have different architectures, leading to vastly different power consumption characteristics. For instance, a task that runs on a GPU might consume more power but execute much faster, therefore consuming less energy to execute than the same task run on a general-purpose machine. The heterogeneity in execution time of the tasks provides the scheduler degrees of freedom to greatly improve the performance as compared to a naïve scheduling algorithm. Similarly the heterogeneity in the power consumption allows the scheduler to decrease the energy consumption.

In this study, we consider optimizing two conflicting objectives. The first is to minimize the *makespan*, that is, the maximum finishing time of all tasks. The second is to minimize the total energy consumption of all machines in the HPC system. We design a novel technique that utilizes a unique relaxation of this scheduling problem then solves it using, in part, linear programming for generating a set of high-quality solutions that represent the tradeoff space between makespan and energy consumption (i.e., Pareto front).

In summary the contributions of this chapter are:

(1) the formulation and evaluation of algorithms that:

  (a) efficiently compute tight lower bounds on the energy and makespan using LP,

  (b) generate a set of high quality bi-objective solutions (i.e., Pareto front), and

  (c) improve upon the Pareto front approximation via convex filling,

(2) the addition of idle power consumption to the formulation of the energy/makespan problem in [39],

(3) a comparison to other Pareto front generation algorithms,

(4) the design and evaluation of a quantitative measure for comparing the quality of bounds on the Pareto front.

The rest of this chapter is organized as follows: first an algorithm for minimum makespan and energy scheduling is presented in Section 2.2. Vector optimization background is given as a tool to solve the bi-objective energy and makespan scheduling problem in Section 3.2. Section 3.3 describes an algorithm to generate Pareto fronts and the convex fill algorithm to further improve the Pareto fronts. Section 3.4 presents the results by comparing the Pareto fronts to an implementation of the non-dominated sorting genetic algorithm II (NSGA-II) for various HPC environments. The algorithm's complexity is given in Section 3.5 along with experimental execution time results. We discuss related work in Section 3.6 and Section 3.8 concludes this study and presents some ideas for future work.

3.1.1. Approach. The fundamental approach of this chapter is to apply DLT [15] to ease the computational requirements of calculating solutions for the makespan and energy scheduling problem. The technique has two major steps. The first step uses DLT, where we assume a single task is allowed to be divided and scheduled onto any number of machines, to calculate the lower-bound solution. After the lower-bound solution is computed, a two-phase algorithm is used to recover a feasible solution from the infeasible lower-bound solution. The feasible solution will be shown empirically to be a tight upper bound on the optimal solution.

HC systems often have groups of machines, typically purchased at the same time, that have identical or very similar performance and power characteristics. This allows one to view these similar machines (only for the purposes of analysis) as a unique machine type. Machines belonging to a *machine type* have virtually indistinguishable performance and power properties with respect to the workload. Machines of the same type may differ vastly in feature sets so long as the performance and power consumption of the tasks under consideration are not affected. Tasks often exhibit natural groupings as well. Tasks of the same *task type* are often submitted many times to perform statistical simulations and other repetitive jobs. Having groupings for tasks and groupings for machines permits less profiling effort to estimate the run time and power consumption for each task on each machine.

Traditionally the static scheduling problem is posed as assigning all tasks to all machines. This formulation is not well suited for recovering a high quality feasible solution from a relaxation of the problem. The decision variables in the classic formulation are binary valued (a task is assigned or not assigned to a machine), and rounding a real value from the lower bound to a binary value can change the objective significantly. Complicated rounding schemes are necessary to iteratively compute a suitable solution. Rather than addressing the problem of assigning all tasks to all machines, we pose the problem as determining the number of tasks of each type to assign to machines of each type. With this modification, decision variables will be large integers $\gg 1$, resulting in only a small error to the objective function when rounding to the nearest integer. This approximation is most accurate when the number of tasks assigned to each machine type is large. In addition to easing the recovery of the integer solution, another benefit of this formulation is that it is significantly less computationally intensive due to solving the higher level assignment of tasks types to machine types with DLT, before solving the fine-grain assignment of individual tasks to machines. As such, this approach can be thought of as a hierarchical solution to the static scheduling problem. Furthermore, for the size of problems considered in this work, the classical relaxation is not solvable in reasonable run time with current computing capabilities.

3.1.2. Lower Bound. The lower bound on the makespan and energy is given by the solution to an LP problem and is formulated as follows. Let there be T task types and M machine types. Let T_i be the number of tasks of type i and M_j be the number of machines of type j. Let μ_{ij} be the number of tasks of type i assigned to machine type j, where $\mu_{ij} \in \mathbb{R}$ is the primary decision variable in the optimization problem. Let **ETC** be a $T \times M$ matrix where ETC_{ij} is the *estimated time to compute* a task of type i on a machine of type j. Similarly, let **APC** be a $T \times M$ matrix where APC_{ij} is the *average power consumption* for executing a task of type i on a machine of type j. These matrices are frequently used in scheduling algorithms (e.g., [10, 19, 20, 24, 40, 41]). **ETC** and **APC** are generally obtained from historical data in real environments.

The lower bound on the finishing time of the machines of a machine type is found by allowing tasks assigned to a machine type to be divided among all machines to ensure the minimal finishing time. With this conservative approximation, all tasks in machine type j finish at the same time. The finishing time of any machine of type j, denoted by F_j, is given by

$$F_j = \frac{1}{M_j} \sum_i \mu_{ij} ETC_{ij} \quad . \tag{8}$$

Throughout this work, sums over i always go from 1 to T and sums over j always go from 1 to M, thus the ranges are omitted. Given that F_j is a lower bound on the finishing time for a machine type, the tightest lower bound on the makespan, denoted by MS_{LB}, is

$$MS_{LB} = \max_j F_j \quad . \tag{9}$$

Without idle power, the energy consumed by a bag-of-tasks is given by $\sum_i \sum_j \mu_{ij} APC_{ij} ETC_{ij}$. To incorporate idle power consumption, one must consider the time duration for which the machines are powered on. In this model, the time duration is the makespan. Not all machines will finish executing tasks at the same time. All but the last machine(s) to finish will accumulate idle power. The idle power consumption $APC_{\emptyset j}$ is that part of APC_{ij} that occurs when no task is executing on a machine of type j. The equation for the lower bound on the energy consumed while incorporating idle power, denoted by E_{LB}, is given by

$$\begin{aligned} E_{LB} &= \sum_i \sum_j \mu_{ij} APC_{ij} ETC_{ij} \\ &\quad + \sum_j M_j APC_{\emptyset j} (MS_{LB} - F_j) \\ &= \sum_i \sum_j \mu_{ij} ETC_{ij} \left(APC_{ij} - APC_{\emptyset j} \right) \\ &\quad + \sum_j M_j APC_{\emptyset j} MS_{LB} \end{aligned} \tag{10}$$

where the second term in the first equation accounts for the idle power. The second equation in Equation (10) breaks the energy into dynamic power and idle power consumption terms. Due to the idle power model, the energy consumption depends directly on the makespan.

The bi-objective optimization problem for the lower bound is:

$$
\begin{aligned}
&\underset{\boldsymbol{\mu},\, MS_{LB}}{\text{minimize}} \quad \begin{pmatrix} E_{LB} \\ \\ MS_{LB} \end{pmatrix} \\
&\text{subject to:} \quad \begin{array}{ll} \forall i & \sum_j \mu_{ij} = T_i \\ \forall j & F_j \leq MS_{LB} \\ \forall i, j & \mu_{ij} \geq 0 \end{array} \quad .
\end{aligned} \tag{11}
$$

The objective of Equation (11) is to minimize E_{LB} and MS_{LB}, where $\boldsymbol{\mu}$ is the primary decision variable. MS_{LB} is an auxiliary decision variable necessary to model the objective function in Equation (9). The first constraint ensures that all tasks in the bag are assigned to some machine type(s). The second constraint is the makespan constraint. Because the objective is to minimize makespan, the MS_{LB} variable will be equal to the maximum finishing time of all the machine types. The third constraint ensures that there are no negative assignments in the solutions.

This vector optimization problem can be solved to find a collection of optimal solutions. It is often solved by weighting the objective functions to form a *linear programming* (LP) problem. Methods to find a collection of solutions are presented in Section 3.3.

Ideally, this LP problem would be solved optimally with $\mu_{ij} \in \mathbb{Z}_{\geq 0}$. However, for practical scheduling problems, finding the optimal integer solution is often not possible due to the high computational cost. Fortunately, efficient algorithms exist that produce high quality sub-optimal feasible solutions. The next few sections describe how we take an infeasible real-valued solution from the linear program and build a complete feasible allocation.

3.1.3. Recovery Algorithm.

3.1.3.1. *Overview.* An algorithm is necessary to recover a feasible solution (i.e., full resource allocation) from the infeasible solution obtained from the lower bound in Equation (11). Numerous approaches have been proposed in the literature for solving integer LP problems by first relaxing them to real-valued LP problems [21]. Our approach here follows this common technique except using computationally inexpensive algorithms tailored to this particular optimization problem. The recovery algorithm is decomposed into two phases. The first phase rounds the solution to the nearest solution while taking care to maintain feasibility of Equation (11). The second phase, called local assignment, assigns tasks to actual machines to build the full resource allocation. The details of the two phases of the recovery algorithm are detailed in Section 2.2.3.

3.2. Linear Vector Optimization

3.2.1. Introduction.

Multi-objective optimization is challenging because there is usually no single solution that is superior to all others. Instead, there is a set of superior feasible solutions that are referred to as the *non-dominated* solutions [42]. When all objectives are to be minimized, a feasible solution x dominates a feasible solution y when

$$\begin{aligned} &\forall i \quad f_i(x) \le f_i(y) \\ &\exists i \quad f_i(x) < f_i(y) \end{aligned} \tag{12}$$

where $f_i(\cdot)$ is the i^{th} objective function. Feasible solutions that are dominated are generally of little interest because one can always find a better solution in some or all objectives by selecting a solution from the non-dominated set. The non-dominated solutions, also known as *outcomes* and *efficient points*, compose the Pareto front.

The optimization problem in Equation (11) is used to compute the lower bound to a bi-objective linear convex optimization problem with convex constraints. The results to follow in this section apply only to this lower-bound scheduling algorithm. These results do not apply after the solution has been rounded or locally assigned because those are non-linear operations. In this section, the term Pareto front will be used to denote the Pareto front of the linear vector optimization problem (lower bound).

Let $C \in \mathbb{R}^{m \times n}$ be the linear mapping from the schedule to the objective space. For our scheduling problem this is a two-dimensional space consisting of energy and makespan; however, these results apply to larger dimensional objective spaces as well. Let $\mathcal{X} \subset \mathbb{R}^n$ be the convex set of constraints, thus it has the property

$$\forall x_a, x_b \in \mathcal{X} \implies \forall \lambda \in [0,1] : \lambda x_a + (1-\lambda) x_b \in \mathcal{X} \quad . \tag{13}$$

The decision variable, x, is contained within $\mathcal{X}$. For the lower-bound optimization problem x is a vector that contains the schedule, $\boldsymbol{\mu}$, and the auxiliary decision variable, makespan.

Using the above notation, the linear convex vector optimization problem is

$$\underset{x \in \mathcal{X}}{\text{minimize}} \quad y = Cx \quad . \tag{14}$$

The lower-bound optimization problem in Equation (11) can be easily converted to this form.

Let the objective space, spanned by y, be given by $\mathcal{Y} \subset \mathbb{R}^m$ and its non-dominated subspace given by $\mathcal{Y}_{ND} \subset \mathcal{Y}$. The Pareto front is given by all the $y \in \mathcal{Y}_{ND}$. This Pareto front is convex and will be proven below. Figure 3.1 is an illustration of the proof. It shows the decision space $\mathcal{X}$ and the objective space $\mathcal{Y}$. Given two points y_a and y_b along the Pareto front,

$$
\begin{aligned}
y_a =& C x_a \in \mathcal{Y}_{ND} \\
y_b =& C x_b \in \mathcal{Y}_{ND} \quad ,
\end{aligned} \tag{15}
$$

a point in-between can be found. For any $\lambda \in [0,1]$ let y_c be on the line between y_a and y_b, such that

$$
\begin{aligned}
y_c =& \lambda y_a + (1-\lambda) y_b \\
y_c =& \lambda C x_a + (1-\lambda) C x_b \\
y_c =& C\left(\lambda x_a + (1-\lambda) x_b\right) \\
y_c =& C x_c \\
y_c \in& \mathcal{Y} \quad .
\end{aligned} \tag{16}
$$

Therefore y_c is feasible and it is on the line between y_a and y_b so the Pareto front cannot have any concave regions. If there were any concave regions of the Pareto front then for some λ the point y_c would not be in the feasible region. It is important that x_c is a convex combination of x_a and x_b. This fact will be used to help fill gaps in the Pareto front in the convex fill algorithm described in Section 3.3.4. A more general version of this proof is available in [43].

The Pareto front for a linear objective function and convex constraint set is also connected [43]. This means that given one point in the Pareto front $\mathcal{Y}_{ND}$ all other points in the Pareto front can be found by taking infinitesimal steps along the Pareto front while never leaving the Pareto front. This is important because if one can find points along the Pareto front then it is possible to connect those points to form an approximation to the Pareto front.

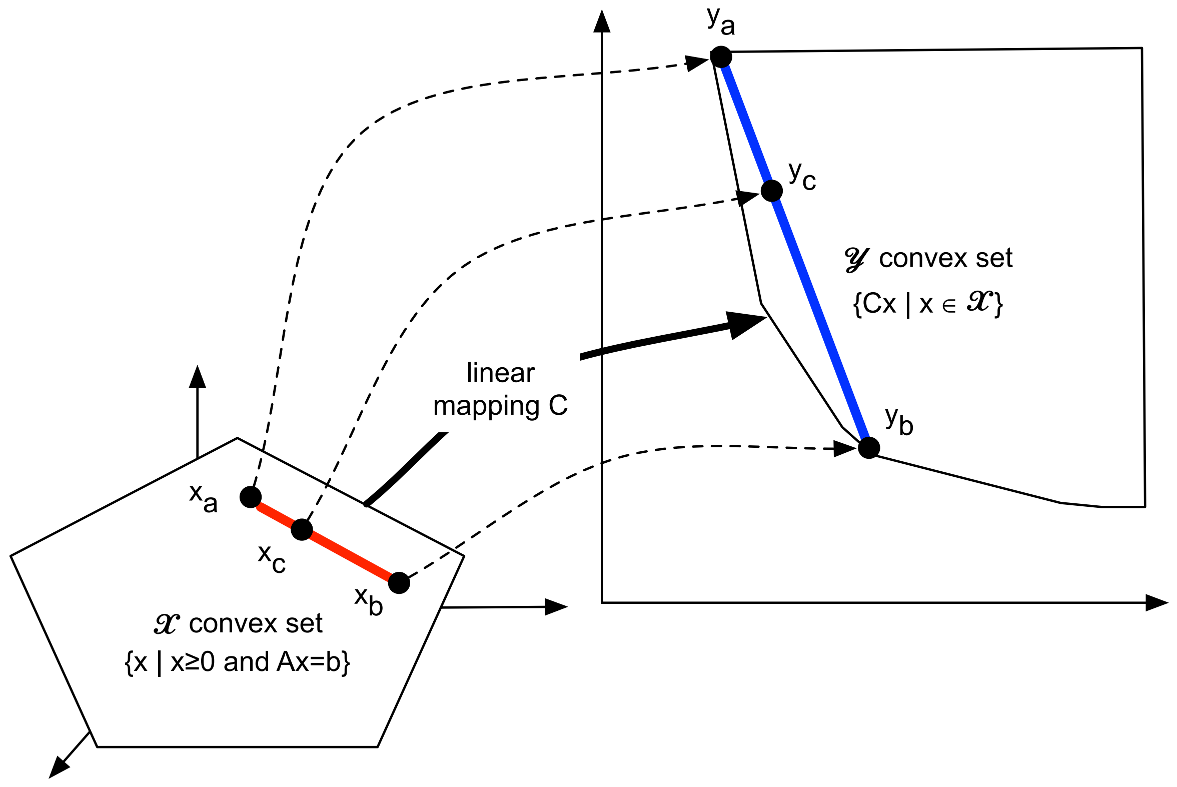


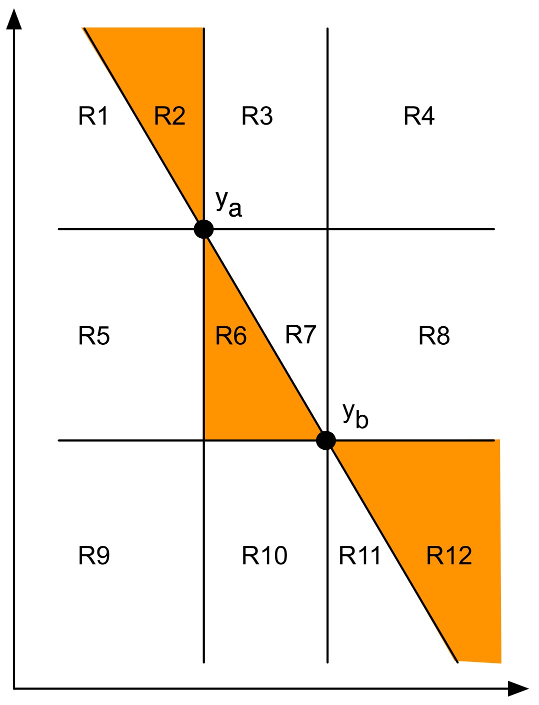
FIGURE 3.1. Illustration of the proof of convexity: Showing the linear mapping from the convex set in the decision space to the convex set in the objective space.

3.2.2. MULTIPLE NON-DOMINATED SOLUTIONS. It is desirable to tightly bound the Pareto front using algorithms that are computationally efficient and scale well as the problem size increases. Non-dominated solutions help to restrict the size of the regions where the remaining Pareto front may exist. Given any optimal non-dominated solution, the Pareto front does not exist to the region to the top right nor to the bottom left of the non-dominated solution. When given any two non-dominated solutions there is more information about the Pareto front that can be extracted when considering them jointly than when considering each individually. Figure 3.2a shows an example of two non-dominated solutions y_a and y_b. The orange regions in Figure 3.2a show where the Pareto front can reside. The Pareto front cannot be in any of the unshaded areas. Regions 3, 4, and 8 are dominated by y_a and/or y_b

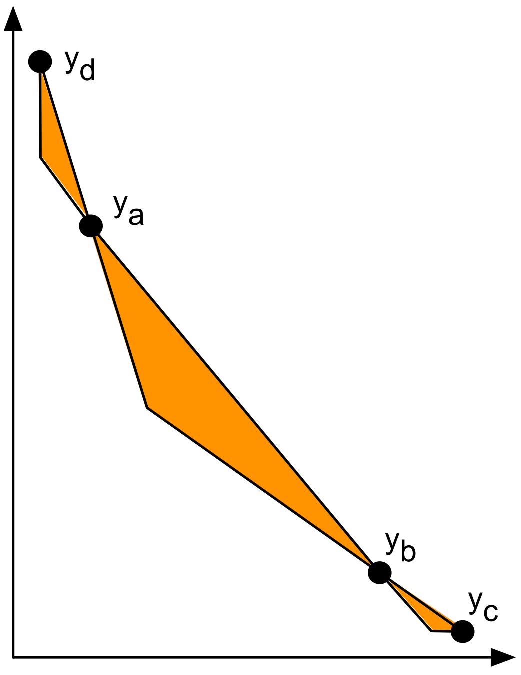
so they cannot be in the Pareto front. Regions 5, 9, and 10 would dominate y_a and/or y_b but y_a and y_b are in the Pareto front so these regions also cannot contain the Pareto front. If the Pareto front were in regions 1, 7, or 11 then the Pareto front would not be convex thus they are excluded as well. The orange regions 2, 6, and 12 are the only regions where the Pareto front can reside.

With four non-dominated solutions, the region where the Pareto front can reside is reduced even further. Figure 3.2b shows four non-dominated solutions. The orange regions show where the Pareto front can reside. For instance, the region between y_a and y_b is reduced due to the convexity requirement imposed by y_d and y_c. It can be shown that adding a fifth non-dominated solution outside of y_d and y_c would not reduce the region between y_a and y_b any further due to convexity of the Pareto front.

3.2.3. Inner and Outer Approximations. In Section 3.3, multiple Pareto front approximation schemes will be discussed. Some of these approximations form an inner approximation while others form an outer approximation. Figure 3.3 shows an example of an optimal Pareto front along with inner and outer approximations for the linear vector optimization problem. The outer approximation is a polytope that encloses $\mathcal{Y}$. Some solutions in an outer approximation may not be feasible but it will encapsulate all the solutions. An inner approximation is a polytope that is fully enclosed by $\mathcal{Y}$. All solutions in an inner approximation are feasible solutions. The Pareto solutions, $\mathcal{Y}_{ND}$, only exist between the inner and outer approximations. Also shown in Figure 3.3 are the nadir and utopia points that form the bounds on the objective space region of interest. To find the nadir and utopia points one must first solve the optimization problem for each objective individually. The nadir point is then found by selecting the maximum value of each objective. Likewise, the



(A) two points



(B) four points

FIGURE 3.2. Given y_a, y_b, y_c, and y_d in the Pareto front only the orange shaded regions may contain the Pareto front. Considering two points together provide much more information than considering them independently. Four points provide much more information than considering only two points due to the convexity of the Pareto front.

utopia point is found by selecting the minimum value of each objective. The nadir and utopia points will be used in the weighted sum and convex fill algorithms.

3.3. Pareto Front Generation

3.3.1. Introduction.

Finding the Pareto front can be computationally expensive because it involves solving numerous variations of the optimization problem to find many optimal solutions. Most algorithms use scalarization techniques to convert the multi-objective problem into a set of scalar optimization problems. Major approaches of scalarization include the hybrid method [43], elastic constraint method [43], Benson's algorithm [44, 45], and Pascoletti-Serafini scalarization [46]. Pascoletti-Serafini scalarization is a generalization of

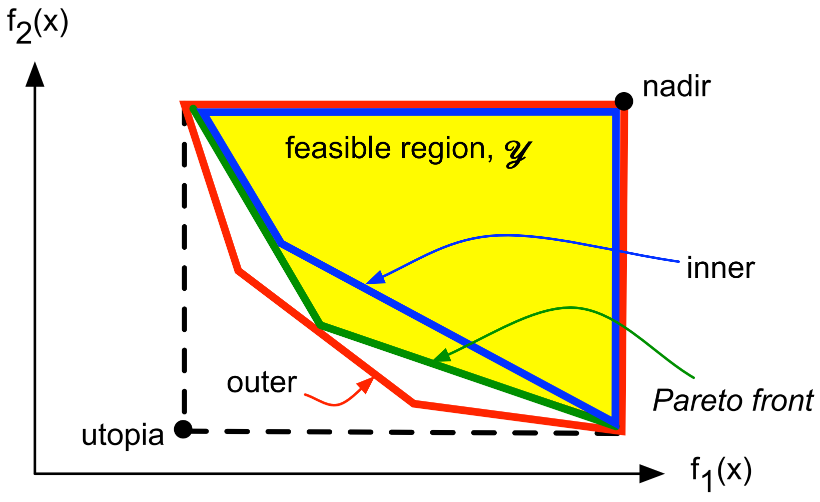


FIGURE 3.3. Inner and outer approximation of the Pareto front: the feasible region is shown in yellow. The Pareto front is in the region between the inner and outer approximation polygons.

many common approaches such as normal boundary intersection, ϵ-constraint, and weighted sum. We will use the weighted sum algorithm in this work. The weighted sum algorithm can find all the non-dominated solutions for problems with a convex constraint set and convex objective functions, when enough weights are chosen [46]. Weighted sum is used for the linear convex problem in Equation (11) to find all non-dominated solutions. A known issue with the weighted sum algorithm is that it does not uniformly distribute the solutions along the Pareto front. The clustering of solutions from weighted sum is mostly overcome by using the algorithm in Section 3.3.4.

Finding the optimal schedule for makespan alone is NP-Hard in general [14], thus finding the optimal (true) Pareto front is also NP-Hard. However, computing tight upper and lower bounds on the Pareto front is still possible. Specifically, a lower bound on a Pareto front is a set of solutions for which no feasible solution dominates any of the solutions in this set. An upper bound on the Pareto front is a set of feasible solutions that do not dominate any

Pareto optimal solutions. The true Pareto front only exists between the lower-bound curve, an outer approximation, and the upper-bound curve, an inner approximation.

3.3.2. Weighted Sum. The weighted sum algorithm forms the convex combination of the objectives and sweeps the weights to generate the Pareto front. The first step is to compute the lower-bound solution for energy and makespan independently of each other. This is used to find the nadir and utopia points, y^{nadir} and y^{utopia} respectively. This is illustrated in Figure 3.3. The next step is to compute the maximum change in each dimension as:

$$y^{\text{nadir}} - y^{\text{utopia}} = \begin{pmatrix} \Delta E_{LB} \\ \Delta MS_{LB} \end{pmatrix} \quad . \tag{17}$$

The scalarized objective for the energy and makespan scheduling problem is then given by:

$$\underset{\boldsymbol{\mu},\, MS_{LB}}{\text{minimize}} \left(\frac{\alpha}{\Delta E_{LB}} E_{LB} + \frac{1-\alpha}{\Delta MS_{LB}} MS_{LB} \right) \quad . \tag{18}$$

A lower bound on the Pareto front can be generated by using several values of $\alpha \in [0, 1]$. As the weights are changed, the objective function changes but the constraints all remain the same. This means that the optimal solution to the LP in the prior step is still feasible in the new problem however possibly sub-optimal. To decrease the run-time the prior solution and the corresponding basis can be used to warm start the primal simplex algorithm [21]. In practice, this leads to significant savings in algorithm run time. Weighted sums will produce duplicate solutions (i.e., $\boldsymbol{\mu}$ is identical for neighboring values of α). Duplicate solutions are removed to increase the efficiency of the subsequent algorithms. Each solution is rounded to generate an intermediate Pareto front. Rounding often introduces many duplicates that can be safely removed. Each integer solution is converted to a full allocation with the local assignment algorithm to create the upper bound on the Pareto front.

3.3.3. Non-dominated Sorting Genetic Algorithm II. The non-dominated sorting genetic algorithm II (NSGA-II) [47] is an adaptation of the GA optimized to find the Pareto front of a multi-objective optimization problem. Similar to all GAs, the NSGA-II uses mutation and crossover operations to evolve a population of chromosomes (solutions). Ideally, this population improves from one generation to the next. Chromosomes with a low fitness are removed from the population. The NSGA-II algorithm modifies the fitness function to work well for discovering the Pareto front. In prior work [39], the mutation and crossover operations were defined for this problem. The NSGA-II algorithm will be seeded in two ways in the following results. The first seeding method uses the minimum energy solution (only minimal energy when there is no idle energy), sub-optimal minimum makespan solution (from the min-min [10] algorithm), and a random population as the initial population. This is the original seeding method used in [39]. The second seeding method uses the full allocations from the local assignment algorithm as the initial population for the NSGA-II.

3.3.4. Convex Fill Algorithm. The weighted sum algorithm finds lower-bound solutions that are on the vertices of the objective space convex set $\mathcal{Y}$. As such, the weighted sum algorithm's solutions tend to be clustered because vertices of the polytope $\mathcal{Y}$ tend to be non-uniformly distributed in the objective space. This leaves large gaps between solutions in the Pareto front. Recall that Figure 3.2 shows that as the distance between the known points along the Pareto front increase so does the size of the allowable region for the Pareto front. To better contain or bound the Pareto front, solutions are needed to help fill the gaps

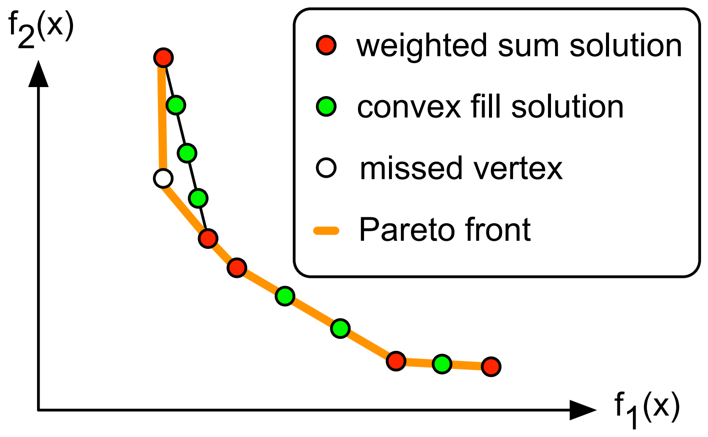


FIGURE 3.4. Example solutions from the weighted sum and convex fill algorithms: Weighted sum's solutions are red and convex fill's additional solutions are green. The Pareto front is the thick orange line. The white solution is a non-dominated solution that the weighted sum algorithm did not discover due to a limited number of weights that causes the neighboring convex fill solutions to not be a lower bound. The convex fill algorithm accurately approximates the solutions within the regions that the weighted sum algorithm missed.

between the weighted sum solutions. The convex fill algorithm developed next is a very fast way to find these desired missing solutions.

Figure 3.4 shows an example of the lower-bound curve. Overlaid on the figure are the weighted sum algorithm's solutions in red. The white solution was not found by sweeping the weights for the weighted sum algorithm due to a fixed number of weights. Convex fill's solutions are shown in green. These solutions fill in gaps between weighted sum solutions.

Recall from Section 3.2 that the convex combination of solutions is also a solution. The convex fill algorithm populates the gaps in the objective space by using this convexity property on the decision space. The convex fill algorithm uses all the unique lower-bound solutions from the weighted sum algorithm.

Define the normalized objective value to be

$$\bar{y} = \frac{y - y^{\text{utopia}}}{y^{\text{nadir}} - y^{\text{utopia}}} \quad . \tag{19}$$

For the energy and makespan problem, this becomes

$$\bar{y} = \begin{pmatrix} \frac{E_{LB} - E_{LB}^{\text{utopia}}}{\Delta E_{LB}} \\ \frac{MS_{LB} - MS_{LB}^{\text{utopia}}}{\Delta MS_{LB}} \end{pmatrix} . \tag{20}$$

The convex fill algorithm uses the L_1 norm as the measure of distance between outcomes in the normalized objective space. For the two-dimensional objective space, when N solutions are provided by weighted sum, the total distance is

$$\sum_{t=1}^{N-1} \parallel \bar{y}_t - \bar{y}_{t+1} \parallel_1 = 2 \ . \tag{21}$$

Let s be the desired maximum L_1 norm distance between two adjacent points in the normalized objective space.

Algorithm 5 Convex fill algorithm

```
Require: $X$ be the list of lower-bound solutions from the weighted sum algorithm
Require: $s$ be the maximum desired spacing between solutions
 1: $Z \leftarrow X$
 2: for all adjacent pairs $(x_a, x_b)$ in $X$ do
 3:     $d \leftarrow \parallel \bar{y}_a - \bar{y}_b \parallel_1$
 4:     $n \leftarrow \lceil d/s \rceil - 1$
 5:     for $t = 1$ to $n$ do
 6:         $\lambda \leftarrow \frac{t}{n+1}$
 7:         $x \leftarrow (1-\lambda)x_a + \lambda x_b$
 8:         $y \leftarrow (1-\lambda)y_a + \lambda y_b = Cx$
 9:         $Z \leftarrow Z \cup \{x\}$
10:     end for
11: end for
12: return $Z$
```

Algorithm 5 gives our convex fill algorithm. It takes the list of lower-bound solutions X and a maximum desired spacing s and produces a list of solutions Z that has no gaps larger than s. This algorithm only works on vector optimization problems with a two-dimensional objective space. Our convex fill algorithm iterates over *adjacent solutions* in

X. Two solutions are said to be adjacent if they are nearest to each other in the objective space. Practically, these adjacent solutions are found by first lexicographically sorting the solutions by their objective vectors. For a sorted set of solutions, the adjacent solutions are those that are consecutive in the list. The solutions from the weighted sum algorithm are already lexicographically sorted if α is swept from 0 to 1. Let the distance between any two solutions be d, and let n be the number of solutions to be added between y_a and y_b to ensure maximum spacing of s. The convex combination of the pair of solutions and the objective values of the solutions are computed. Lastly, the new solution x is appended to the list Z.

Unlike solutions from the weighted sum lower bound, the solutions from the convex fill algorithm are not guaranteed to be a lower bound. This is because there is no guarantee that all solutions were found when performing the weighted sum sweep. If all vertices or solutions of $\mathcal{Y}$ are found, then convex fill will produce lower-bound solutions. Figure 3.4 illustrates how a vertex that is not found by weighted sum, causes the convex fill algorithm's solutions to no longer be on the lower-bound curve. To use the convex fill algorithm for producing lower-bound solutions, an optimal algorithm such as Benson's algorithm is required [44]. Benson's algorithm for Equation (11) is much slower than weighted sum.

To construct the full allocation from the lower bound, the recovery procedure described in the Section 3.3.2 is used. Results for the convex fill algorithm are presented in Section 3.4.5.

3.3.5. Pareto Front Solution Quality. Many approaches to quantitatively and qualitatively measure the quality of a Pareto front have been used in the literature. One approach uses a measure of how well-spaced the solutions are in the objective space by computing the sample variance of the distance between solutions [48]. While this is useful in some cases it is not a good measure of the overall quality of an approximation to the Pareto front. A byproduct of the weighted sum algorithm described in Section 3.3.2 is that it produces many lower and upper-bound solutions that can be used to constrain the Pareto front to a small region. To quantitatively measure the performance of algorithms, we can compute the area of this region. The true Pareto front becomes more tightly bounded when the area of this region becomes smaller.

Computing the area of this region in a consistent manner is nontrivial. Careful definitions of the outer approximation (lower bound) and the inner approximation (upper bound) are necessary.

After the lower and upper-bound solutions are obtained, we begin the calculation for the area of this region by computing the overall nadir point from the lower and upper-bound solutions. We then add three more points to the Pareto front to form a closed polygon, namely $(E_{LB}^{\text{utopia}}, MS_{LB}^{\text{nadir}})$, $(E_{LB}^{\text{nadir}}, MS_{LB}^{\text{utopia}})$, and $(E_{LB}^{\text{nadir}}, MS_{LB}^{\text{nadir}})$. Next, the outer approximation polygon is found using the convexity properties of the lower bound outlined in Figure 3.2. The inner approximation polygon is computed by using the fact that only the region to the top-right of each point should be included in the polygon. Both the inner and outer approximation polygons are not convex polygons. The area where the Pareto front can reside is found by taking the difference between the areas of the outer and inner approximation polygons. Section 3.4 shows example inner and outer approximation polygons in Figure 3.10.

3.4. Results

3.4.1. Simulation Setup.

ETC and **APC** matrices are needed to evaluate the algorithms. To generate these matrices, a set of five benchmarks executed over nine machine types was used to construct the initial matrices [23]. Then the method found in [24] was used to construct larger **ETC** and **APC** matrices. Nominally there are 1,100 tasks comprised of

30 task types. The number of tasks per task type varies from 11 to 75 and was generated by the method used in [24]. There are nine machine types with four machines of each type for a total of 36 machines. A complete description of the environment are available in the supplementary material. This environment will be referred to as the nine machine type environment.

Unless noted otherwise, the simulations were performed on a mid-2009 MacBook Pro with a 2.5 GHz Intel Core 2 Duo processor. All the algorithms were implemented in C++ and optimized using our best effort. The COIN-OR CLP solver was used to solve the LP problems. The third party CLP library is also written in C++ [49]. The hardware being used for running the NSGA-II simulations is a 2013 Dell XPS'15 with an Intel i7-4702HQ 2.2 GHz CPU. The NSGA-II code is implemented in C++.

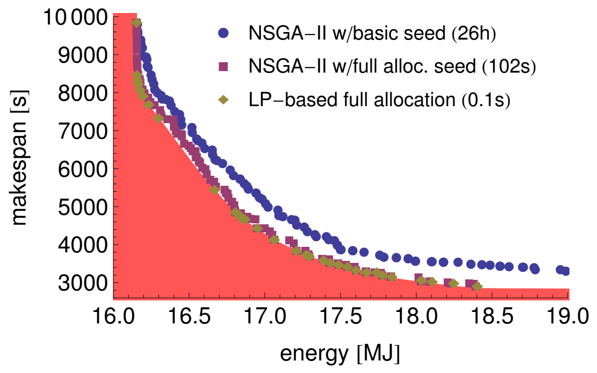
The LP-based Pareto fronts are all generated with 1,000 evenly distributed weights. The weights are used in the weighted sum algorithm to parametrically sweep the Pareto front. Generally this leads to fewer than 100 full allocations depending on the particular problem.

3.4.2. Pareto Fronts. Figure 3.5 shows the lower bound and approximate Pareto fronts for four different environments. The LP-based lower bound is shown by the red shaded region. The figure shows the actual solutions as markers that are connected by lines for the NSGA-II algorithm and the LP-based algorithm. The legend shows the techniques associated with the markers in addition to the total algorithm execution time. All the systems have zero idle power consumption. The NSGA-II algorithm was allowed to run for one million generations when seeded with the basic seed. One thousand generations were used when seeded with the full allocation seed.

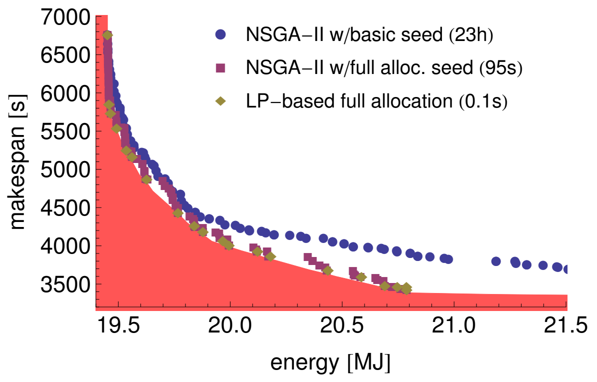
Figure 3.5a shows the results for the nine machine type environment. The lower bound and and LP-based full allocation are nearly indistinguishable along the entire Pareto front.

This means that the true Pareto front is tightly bounded even though it is unknown. The curve that is dominated (i.e., higher values in both makespan and energy) by all other curves is the set of solutions generated by the NSGA-II using the first seeding method. This means that it took NSGA-II over a day to find a set of solutions that are of poor quality in comparison to our technique that took 0.1 s. The set of red solutions are those obtained from seeding NSGA-II with the set of solutions produced by our local assignment algorithm. Seeding with the full allocation allows the NSGA-II to both converge to an improved front as well as decrease the run time. The NSGA-II attempts to evenly distribute the solutions along the Pareto front as can be seen in Figure 3.5a. All the algorithms seem to perform well at minimizing energy, presumably because computing the optimal minimum energy solution is relatively easy compared to finding the optimal minimum makespan solution. To obtain the minimum energy solution, each task is assigned to the machine that requires the lowest energy to execute that task. Figure 3.5a shows that all the algorithms produce good minimum energy solutions; however, for makespan there are significant differences in solution quality. The new LP-based algorithms produce better quality solutions in significantly less time.

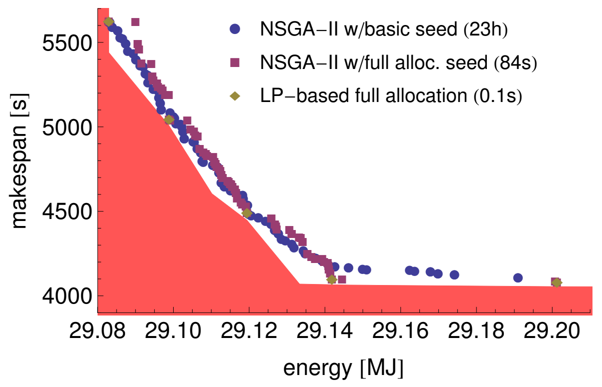
A few different systems are used to further demonstrate the applicability of the LP-based Pareto front generation technique. Figure 3.5b shows a system composed of just the first six machine types from the previous system, with six machines per type. Figure 3.5c shows an even smaller system by taking only the first two machine types, with 18 machines per type. The total number of tasks, task types, and machines is unchanged. These figures show how the lower bound and upper bound still outperform the NSGA-II algorithm even when the number of machines types become small.



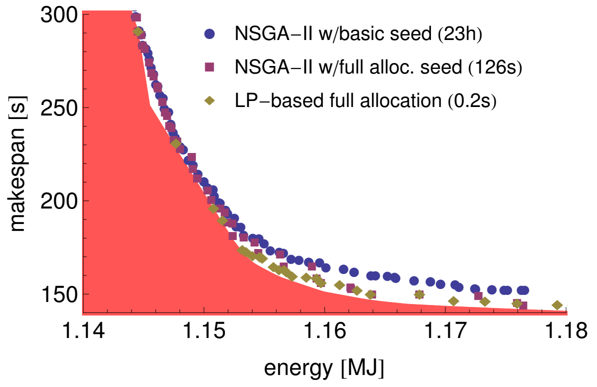
(A) nine machine type environment



(B) six machine type environment



(C) two machine type environment



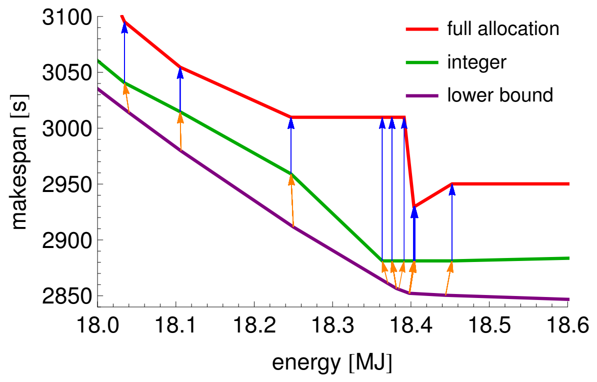
(D) synthetic ten machine type environment

FIGURE 3.5. Lower bound and approximate Pareto fronts: The region excluded by the lower bound from the LP is shaded in orange and truly bounds the approximate Pareto fronts. The full allocation or upper bound is very near the lower bound so the Pareto front is tightly bounded. The times shown in the parenthesis in the legend indicate the total time to compute the solution. Solution quality is rather poor with the NSGA-II using the original seed and expensive to compute, however the NSGA-II seeded with the full allocations produces a reasonable result, close to the full allocation, in much less time, but still is not as good as the full allocation in places.

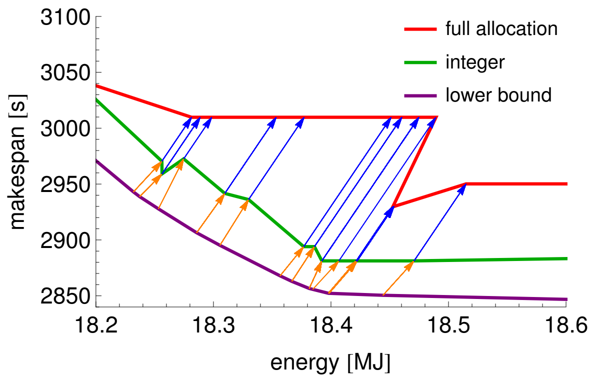
The results in Figure 3.5d are based on an entirely different environment that was previously used in [39]. The HPC system has 50 machines selected from ten machine types. There are 1,000 tasks made from 50 task types. The **ETC** and **APC** matrices were generated randomly with the CoV method described in [29]. Even though this environment is very different from the previous environments, the LP-based algorithm produces a superior Pareto front in significantly less time.

3.4.3. Solution Progression. To further understand the effect of the three phases of the proposed algorithm we can follow a set of solutions as they progress from the lower bound to the upper bound. Figure 3.6 illustrates how the solutions progress through the three phases of the algorithm. Figure 3.6a shows the progression without considering idle power consumption. This figure is a zoomed in version of a portion of Figure 3.5a that details the progression of individual solutions for the nine machine type environment. The lowest line represents the lower bound on the Pareto front. Each orange arrow represents a solution as it is rounded. In every case, the makespan increases while the energy may increase or decrease. The energy consumption can change during the rounding phase because tasks may become assigned to different machine types that may be more or less efficient compared to the original fractional assignment. As a given solution, $\boldsymbol{\mu}$, is rounded, machines will finish at different times, thus increasing the makespan. Each blue arrow represents a solution that is being fully allocated via the local assignment algorithm. The energy in this case does not change because the local assignment algorithm does not move tasks across machine types, thus the power consumption cannot change. The makespan increases are highly varying and depend on how well tasks in a machine type pack onto individual machines. The full allocation solution second from the right dominates the one on the far right. In this case the solution on the far right would be removed from the estimate of the Pareto front.

Figure 3.6b shows the progression of the solutions when considering idle power. The idle power consumption is set to 10% of the mean power for each machine type, specifically $APC_{\emptyset j} = \frac{0.1}{T} \sum_i APC_{ij}$. As the makespan increases, more machines will be idle for longer, so the idle energy increases. The local assignment phase now negatively affects the energy consumption because it will typically have machines idle for some amount of time.



(A) no idle power



(B) 10% idle power

FIGURE 3.6. Progression of solutions from lower bound to integer to upper bound without idle power (a) and with idle power (b).

3.4.4. IDLE POWER CONSUMPTION. Figure 3.7 shows the effect of idle power on the Pareto front for the nine machine type environment. The curves show the lower bound on the optimal Pareto front with different percentages of idle power. The penalty for having a large makespan increases as the idle power increases because a large fraction of machines are idle for longer. The optimal energy solutions must now have a shorter makespan to reduce energy usage. This causes the Pareto front to contract in the makespan dimension and shift to the right slightly. As idle power usage approaches 100%, the problem degenerates to the single objective minimum makespan scheduling problem.

3.4.5. CONVEX FILL. Figure 3.8 shows the solution front after convex filling while Figure 3.5a is shown without the convex filling. Convex filling increases the run time only slightly, yet produces a much more complete Pareto front compared to using the weighted sum algorithm alone.

Figure 3.9 shows how the solutions from the lower bound progress to the full allocation when using the convex fill algorithm with $s = 0.01$. Comparing this figure to Figure 3.6a shows that the solutions added by the convex fill algorithm to the lower bound generate many unique integer and full allocation solutions. This allows the upper bound formed by

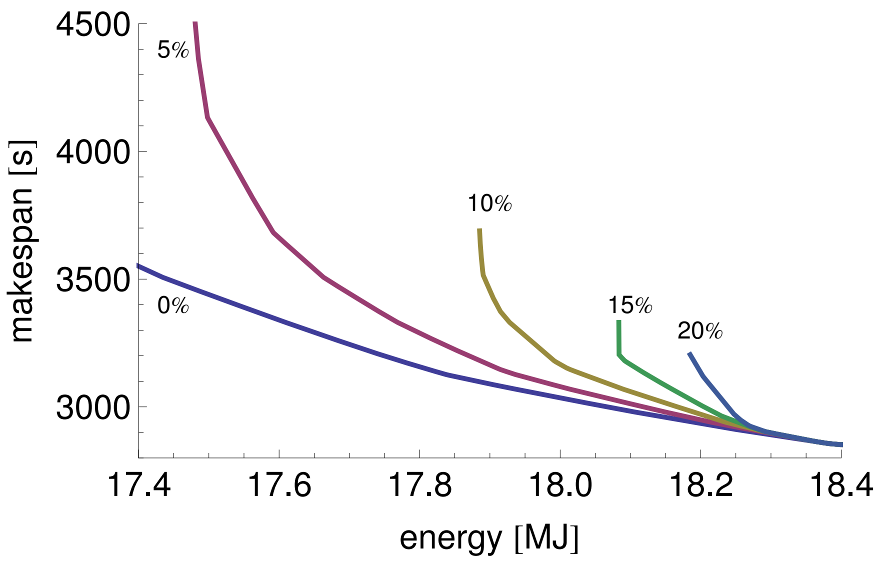


FIGURE 3.7. Pareto front lower bounds when varying idle power: Idle power is increased in 5% increments as labelled on the figure. As idle power increases, the reward for minimizing makespan also increases. The curve without idle power is only partially shown.

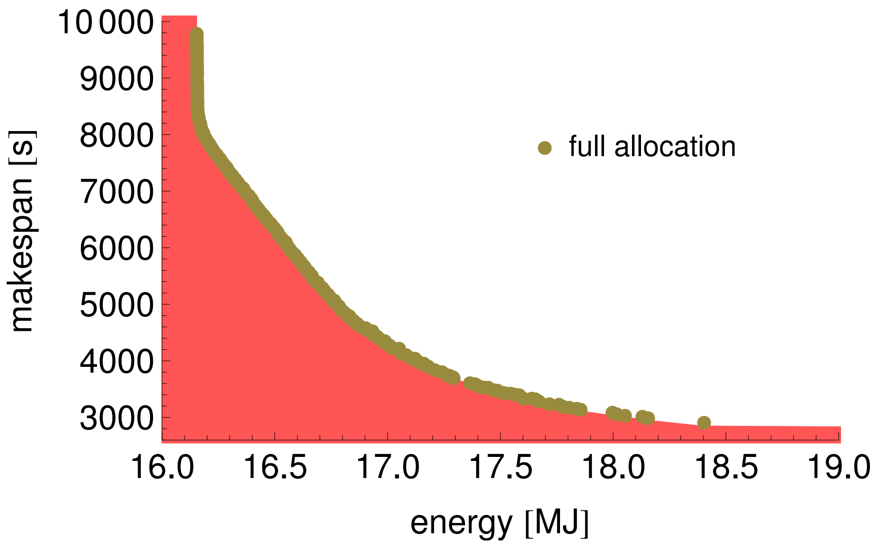


FIGURE 3.8. Solutions after applying the convex fill algorithm: there are no more large spaces between full allocation solutions as compared to Figure 3.5a.

the full allocations to be much tighter, as will be measured quantitatively in Section 3.4.6. A decision maker also would benefit from having fewer gaps in the Pareto front solutions when selecting an appropriate schedule. The additional run time of generating these extra solutions is negligible compared to the run time of the weighted sum algorithm.

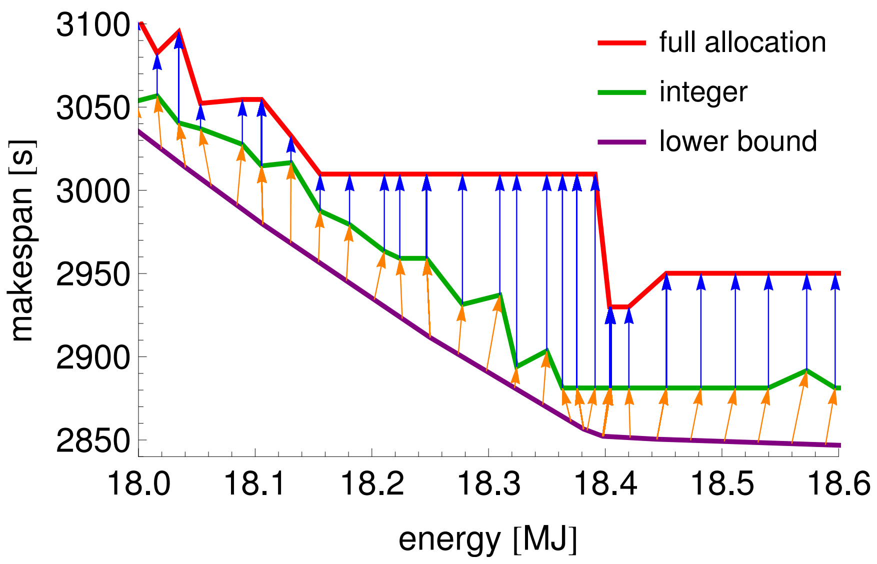
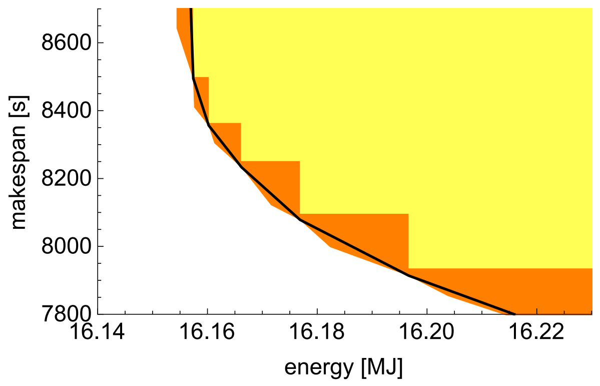
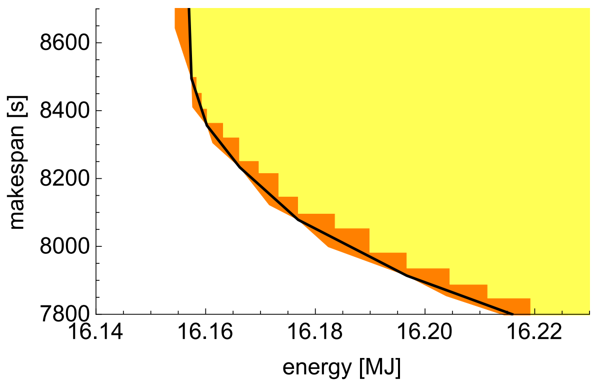


FIGURE 3.9. Progression of solutions from lower bound to upper bound when using the convex fill algorithm: Convex filling produces unique integer and full allocations that tighten the Pareto front bounds compared to without convex filling in Figure 3.6a.

3.4.6. AREA BETWEEN PARETO FRONT BOUNDS. Using the algorithm detailed in Section 3.3.5 to compute the area between the inner and outer approximations, the quality of the different algorithms that generate bounds on the Pareto front can be quantified. Figure 3.10 shows examples of the inner and outer approximations of the Pareto front for the nine machine type environment. The orange area is the region where the true Pareto front can exist. The yellow region in the upper right is forbidden because full allocations have been found that dominate every solution in that region. The white part of the graph to the bottom left is also forbidden because there are no feasible solutions in that region. This white region is bounded by the outer approximation found from the lower-bound solution. The LP-based algorithm using just weighted sum is shown in Figure 3.10a. The same region along the Pareto front after applying the convex fill algorithm is in Figure 3.10b. The convex filling does not change the outer approximation but it does add more unique full allocations



(A) LP-based



(B) LP-based with convex fill

FIGURE 3.10. Inner and outer approximation polygons with and without convex filling: The orange region is where the Pareto front can exist. The convex fill algorithm greatly reduces the allowable area where the Pareto front can exist.

that greatly increases the area of the inner approximation making the bound on the Pareto front tighter.

Table 3.1 lists the area (in megajoule-seconds) that is between the inner and outer approximation polygons. When the area is small, the Pareto front is tightly bounded. The area is computed using the method in Section 3.3.5. Of the Pareto front generation algorithms discussed, only the LP-based algorithm produces an outer approximation or lower bound. The LP-based outer approximation is used for all the results shown in Table 3.1. The table shows four different algorithms for computing the inner approximation. The results are shown for the nine, six, two, and ten machine type environments whose Pareto fronts are shown in Figure 3.5. The NSGA-II with the basic seed can only very loosely bound the Pareto front. The LP-based algorithm bounds the Pareto front much more tightly than NSGA-II. However, running the NSGA-II algorithm as a post process to the LP-based algorithm does improve the quality of the bounds. This is because the NSGA-II will find solutions that are between the seeded full allocations thus filling in the gaps and reducing the area. The convex fill algorithm is an alternative post process to the LP-based algorithm

TABLE 3.1. Area between bounds

algorithm	nine	six	two	ten
NSGA-II	2149	1351	115	2.655
LP-based	684	339	63	1.011
NSGA-II seeded	436	306	53	0.851
LP with convex fill	231	238	38	0.762

that executes extremely fast. The convex fill algorithm bounds the Pareto front the tightest for all environments considered here.

The lower bound can be tightened even further by using the technique described in Section 3.7 at the cost of greater computation.

3.5. Computational Complexity

3.5.1. Analysis. A complete analysis of the scaling properties of the single objective minimum makespan scheduling problem are in Chapter 2. Those results are summarized below and then extended for the full Pareto front generation problem.

Recall that T and M are the number of task and machine types respectively. The average case complexity of solving a single LP problem with the simplex algorithm is $(T+M)^2(TM+1)$. The complexity of the rounding algorithm is $\mathcal{O}(T(M\log M))$. Let $T_{\text{total}} = \sum_i T_i$ be the total number of tasks and $M_{\text{total}} = \sum_j M_j$ be the total number of machines. Assuming for the sake of analysis that tasks and machines are evenly distributed so $\forall j \; \sum_i \mu_{ij} \approx \frac{T_{\text{total}}}{M}$ and $M_j \approx \frac{M_{\text{total}}}{M}$. The local assignment algorithm has complexity $\mathcal{O}(MT\log T + T_{\text{total}}\log M_{\text{total}} - T_{\text{total}}\log M)$.

The complexity of the overall algorithm to find both the lower bound and upper bound (full allocation) is driven by either the lower-bound algorithm or the local assignment algorithm. The complexity of the lower bound and rounding algorithms are independent of the number of tasks and machines. Those algorithms depend only on the number of task types and machine types. This is a very important property for large-scale environments. Millions of tasks and machines can be handled easily if the machines can be reasonably placed in a small number of homogeneous machine types and, likewise, tasks can be grouped by a small number of task types. Only the local assignment algorithm's complexity has a dependence on the number of tasks and machines. This phase is only necessary if a full allocation or schedule is required. The lower bound can be used to analyze much of the behavior of the HPC environment at a lower computational cost. Furthermore, the local assignment algorithm can be trivially parallelized because each machine type is scheduled independently.

When generating a Pareto front the lower-bound solutions are generated by re-solving a similar LP many times. The objective space of vector optimization problems are polytopes so they have a finite number of vertices. This means that there is a maximum number of solutions that can be found by the weighted sum algorithm because it is restricted to vertices. Usually there are a large number of duplicate solutions from weighted sum that can safely be removed thus reducing the computational cost of subsequent algorithms such as rounding and local assignment.

3.5.2. Results. To demonstrate the scaling properties of our Pareto front generation algorithm, a scaled up version of the nine machine type environment was used to generate the larger environments used in this simulation. The number of machines per type was changed from 4 to 400 so there are now 3,600 machines. Tasks for each trial were generated by sampling the task type distribution with replacement. The mean of 50 trials is shown. For this set of simulations the convex filling algorithm was used to improve the quality of the Pareto front that was computed.

Figure 3.11 shows the relative area between the inner and outer approximation polygons as a function of the number of tasks. The quality of the bound improves (i.e., relative area

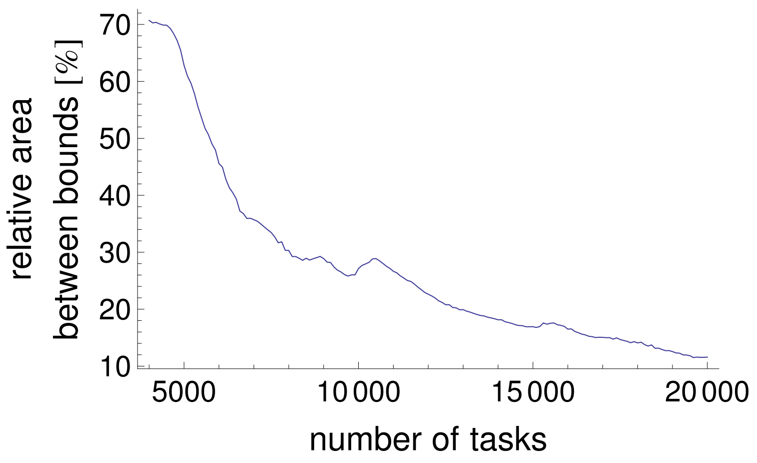


FIGURE 3.11. Relative percent increase in area as a function of the *total number of tasks*: The quality of the solution improves as more tasks are used.

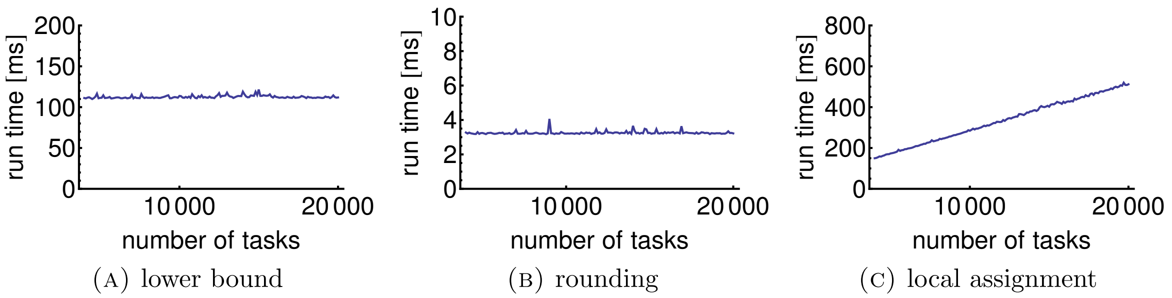


FIGURE 3.12. Algorithm run time versus *total number of tasks*: Both the lower bound and the rounding algorithms are independent of the number of tasks. The local assignment, used to obtain the full allocation, is linearly dependent on the number of tasks.

decreases) as the number of tasks to schedule increases. Figure 3.12 shows the run time of the three phases of the algorithm as a function of the number of tasks. The Figure 3.12a shows the time to run the weighted sum algorithm and solve all the resultant LPs. Corresponding to the analysis in Section 3.5.1, the weighted sum algorithm is independent of the number of tasks. The rounding algorithm is shown in Figure 3.12b. Its runtime is also approximately independent of the number of tasks. Figure 3.12c shows the local assignment and is the only phase of the algorithm that depends on the number of tasks. The dependency is linear which matches the analysis in Section 3.5.1.

The time required to solve the initial LP problem is on average 12.6 times more expensive than doing a single re-solve of the problem after perturbing the weights. This is because the LP problem changes only slightly in the objective function so only a few primal simplex steps are required to restore optimality.

3.6. Related Work

Techniques for generating Pareto fronts have been well studied (e.g., [24, 39, 41, 42, 47]). Our LP-based approach achieves huge gains in run time and solution quality over prior methods by exploiting properties that are common to static scheduling problems. Our approach takes advantage of the common property that each machine in an HPC system is not unique but belongs to one of a few machine types. Our work also is focused on very large-scale systems and finding high quality solutions on average, whereas [14, 31] are concerned with worst-case performance of the scheduling algorithms. The energy and makespan problem is a specialization of the classic optimization problem of minimizing makespan and cost [14, 31].

While our work deals with scheduling tasks to entire machines, the algorithms could also be applied to scheduling tasks to cores within a machine or across cores on many machines. The full allocation recovery algorithm we use is similar in nature to the algorithms presented in [33] that deal with scheduling on a single machine with deadlines to determine the best DVFS parameters to use to minimize energy as a secondary objective. An algorithm is presented in [34] that minimizes energy while constraining makespan and reliability to provide computationally efficient schedules for DVFS scheduling on identical processors.

In [20], the A* search algorithm is used to assign tasks to machines considering task dependencies and communication constraints. This algorithm is very expensive for large numbers of tasks because the algorithm's branching factor is on the order of the number of machines and the depth is on the order of the number of tasks.

NSGA-II based approaches to find the energy and makespan Pareto front are in [39, 41] without the use of task and machine types. Other algorithms exist in the literature that may perform differently than NSGA-II such as the improved strength pareto evolutionary algorithm (SPEA2) algorithm [50].

Makespan and energy bi-objective optimization is also proposed in [51] via a mixed integer linear programming (MILP) formulation using the weighted sum algorithm to find solutions along the Pareto front. They present an adaptive algorithm that fills in the weighted sum solutions by solving additional MILP problems. They assign individual tasks to individual machines so scalability will suffer. An extension of their work that uses vector ordinal optimization to approximate the Pareto front is presented in [52].

3.7. Tighter Lower Bound on the Pareto Front

3.7.1. Motivation. The vector optimization in Equation (11) solved via weighted sums in Section 3.3.2 provides a set of solutions that are on the outer approximation to the Pareto front. In Section 3.3.5 these lower bound solutions are used to form a non-convex polygon that is the outer approximation. This lower bound is not as tight as it could be because the actual outer approximation must be convex.

Consider Figure 3.13 where y_1, y_2, and y_3 are Pareto optimal for the linear vector optimization problem in Equation (11). Points x_1 and x_2 in addition to solutions y_1, y_2 and y_3 are used for the outer approximation polygon in Section 3.4.6. We introduce two new points z_1 and z_2 that are unknown. The points y_1, z_1, y_2, z_2, and y_3 form part of a convex polygon. The polygon with maximum area can be used as the area for the lower bound polygon. Note

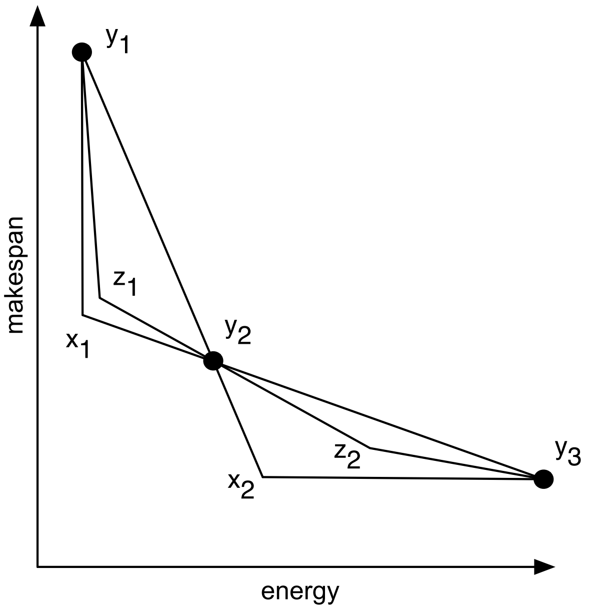


FIGURE 3.13. Simple Pareto front: y_1, y_2, and y_3 are from weighted sum however z_1 and z_2 can be found to create a tighter bound then when using x_1 and x_2.

that z_1 and z_2 are not necessarily solutions like y_1, y_2, and y_3. This convex polygon is not the outer approximation polygon but rather has area that is required to be larger than the area of the true Pareto front that is know to be convex. Next we start building the optimization problem that finds this maximum area polygon.

The signed area of a triangle is used to develop the problem to follow. The signed area of triangle ABC is given by [53]

$$\begin{aligned} \text{area}(A,B,C) =& \frac{1}{2}\begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_x & C_y & 1 \end{vmatrix} \\ =& \frac{1}{2}(-B_xA_y + C_xA_y + A_xB_y \\ & - C_xB_y - A_xC_y + B_xC_y) \ . \end{aligned} \tag{22}$$

If A, B, and C are counter clockwise around the triangle then the area is positive. Likewise if the points are clockwise the area is negative.

We desire to maximize the area(z_1, y_2, y_1) + area(z_2, y_3, y_2). From Equation (22) one can see that this objective is linear in z_1 and z_2. The point z_1 must be contained within the triangle y_1, y_2, x_1 to be within the original outer approximation and maintain a convex polygon. Likewise the point z_2 must be contained within the triangle y_2, y_3, x_2. These constraints are linear half plane constraints. To ensure that the polygon is convex, the point y_2 must be below the line defined by z_1 and z_2. At optimality z_1, y_2, and z_1 will be collinear. This is equivalent to area$(z_1, y_2, z_2) = 0$. This constraint is quadratic in z_1 and z_2. This problem is well defined and can be solved but first we generalize this to the case of an arbitrary number of solutions.

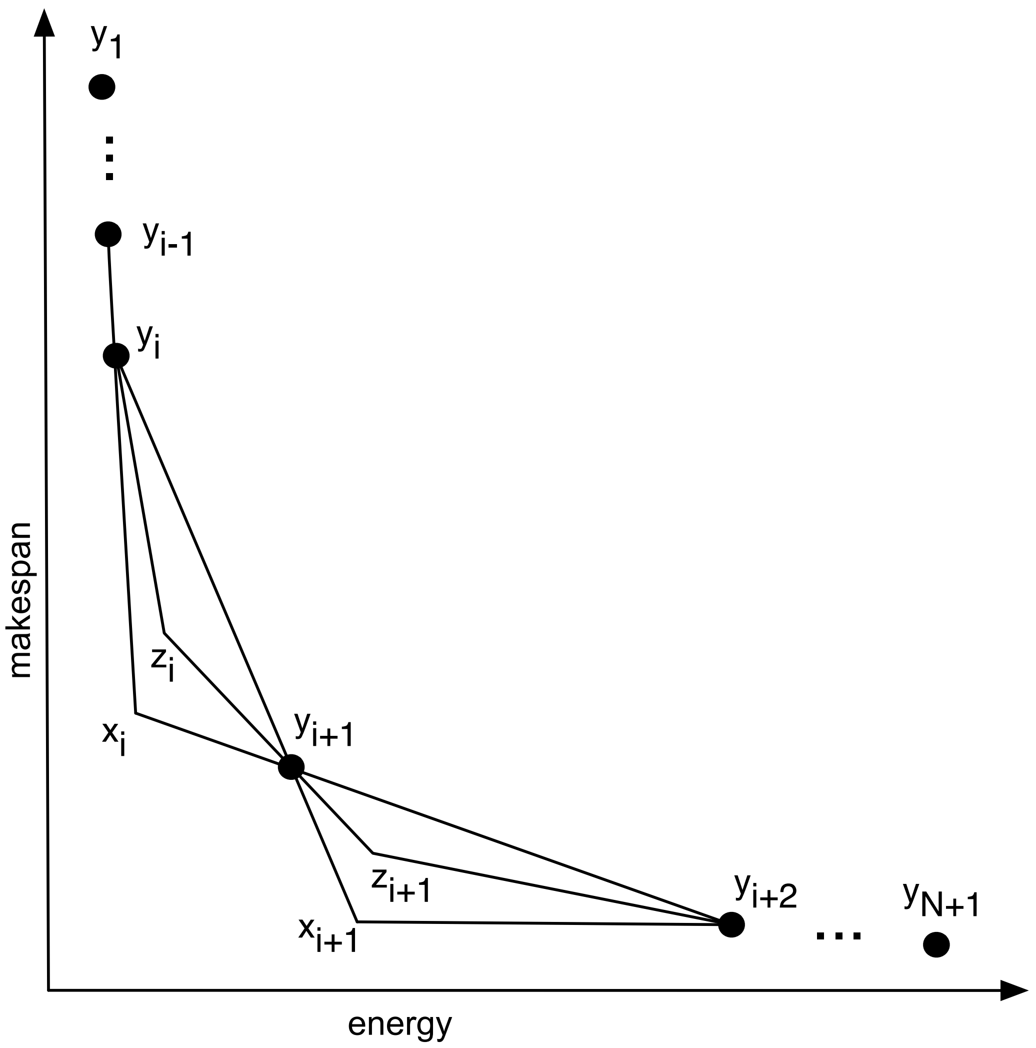


FIGURE 3.14. Generalization of the tighter, convex lower bound

3.7.2. PROBLEM FORMULATION. Figure 3.14 shows the Pareto front with an arbitrary number of Pareto efficient points. Let y_i be the i^{th} objective of the solution on the Pareto front to the linear vector optimization problem. Let z_i be the additional point in between y_i and y_{i+1} to form the convex outer approximation. Let there be $N+1$ Pareto front points y; thus there are N additional points z. To ease the problem formulation define $y_0 = y_1 + (0, 1)$ and $y_{N+2} = y_{N+1} + (1, 0)$.

The optimization problem becomes

$$\begin{aligned}
&(23a) && \underset{\mathbf{z}}{\text{maximize}} && \sum_{i=1}^{N} \text{area}(z_i, y_{i+1}, y_i) && \\
&(23b) && \text{subject to:} && && \\
&(23c) && \forall i \in \{1, \cdots, N-1\} && \text{area}(z_i, y_{i+1}, z_{i+1}) \geq 0 && \text{weak collinear} \\
&(23d) && \forall i \in \{1, \cdots, N\} && \text{area}(z_i, y_{i+1}, y_i) \geq 0 && \text{upper} \\
&(23e) && \forall i \in \{1, \cdots, N\} && \text{area}(z_i, y_{i-1}, y_i) \geq 0 && \text{left} \\
&(23f) && \forall i \in \{1, \cdots, N\} && \text{area}(z_i, y_{i+1}, y_{i+2}) \geq 0 && \text{bottom} \ .
\end{aligned}$$

The optimization problem in Equation (23) has the general form of

$$\begin{aligned}
&(24a) && \underset{\mathbf{z}}{\text{maximize}} && \mathbf{c}^T\mathbf{z} && \\
&(24b) && \text{subject to:} && && \\
&(24c) && \forall i \in \{1, \cdots, N-1\} && \mathbf{z}^T\mathbf{Q}_i\mathbf{z} + \mathbf{d}_i^T\mathbf{z} \geq 0 && \text{quadratic} \\
&(24d) && \forall i \in \{1, \cdots, 3N\} && \mathbf{A}_i\mathbf{z} \geq b_i && \text{linear}
\end{aligned}$$

where the quadratic constraint is from the first constraint in Equation (23). The quadratic coefficient matrix $\mathbf{Q}$ is not symmetric but can be made symmetric by letting

$$\mathbf{R}_i = \frac{\mathbf{Q}_i + \mathbf{Q}_i^T}{2} \tag{25}$$

and using the identity

$$\mathbf{z}^T\mathbf{Q}_i\mathbf{z} = \mathbf{z}^T\mathbf{R}_i\mathbf{z} \ . \tag{26}$$

The symmetric matrix $\mathbf{R}_i$ is $N \times N$ and very sparse with only four non-zero entries. It can be shown that the matrix $\mathbf{R}_i$ has positive and negative eigen values, and thus it is an indefinite matrix. So $\mathbf{z}^T\mathbf{R}_i\mathbf{z}+\mathbf{d}_i^T\mathbf{z}$ is not a concave function and the corresponding constraint defines a non-convex set [54]. This quadratically constrained linear programming problem is thus a non-convex optimization problem. Using the first and second derivatives from the constraints and objective, the interior point method can be used to find a local maxima relatively easily. An initial marginally feasible point is $z_i = \frac{y_{i+1}+y_i}{2}$. The interior point algorithm effectively converts the problem into an unconstrained non-linear optimization problem and then uses Newton steps to obtain local optimality. To use the solution for the lower bound, the provably optimal solution must be found which is not possible for this class of problems. Due to how tightly constrained this optimization problem is in practice, the local optimization solution seems to converge to the global maxima in practice for small problems and medium sized problems.

3.7.3. Results. The optimization problem takes about 50 s to solve[2] when $N = 53$. Table 3.2 shows areas between different bounds for the nine, six, two, and ten machine type environments. The table shows the LP-based convex filled area with the original loose lower bound and the tighter convex lower bound. Also shown is a bound, labelled "exhaustive" in Table 3.2, that is computed using Benson's optimal algorithm [44, 45] or equivalently using weighted sum algorithm with a sufficiently large (exhaustive) number of weights. In this case all the vertices are found thus the outer approximation is just the convex hull of the points.

[2]The optimization problem was implemented in Mathematica using FindMaximum[].

TABLE 3.2. Area between bounds

algorithm	nine	six	two	ten
loose	231	238	38.1	0.762
convex	226	235	36.3	0.748
exhaustive	210	226	34.1	0.689

The convex lower bound is tighter than the original lower bound. It is more expensive to compute but does provide a tighter lower bound. The exhaustive lower bound is the tightest in all cases but is not practical for extremely large problems.

3.8. Conclusions

A highly scalable scheduling algorithm for the energy and makespan bi-objective optimization problem was presented. The complexity of the algorithm to compute the lower bound on the Pareto front was shown to be independent of the number of tasks and machines. Only the algorithm to compute the full allocation, that is computationally inexpensive and trivially parallelizable, is dependent on the number of tasks and machines. The quality of the solution also improves as the size of the problem increases. The LP-based Pareto front was compared to the solution found with the NSGA-II algorithm and shown to be superior in solution quality and algorithm run time for a variety of test environments. A post-process to the LP-based algorithm was developed that fills in solutions quickly using the convexity property of the relaxed problem. This was shown to further increase the quality of the Pareto front with a negligible increase in run time. A new approach for quantifying the quality of the Pareto fronts was developed and used to compare the different algorithms. These properties make this algorithm perfectly suited for very large-scale scheduling problems. This new LP-based Pareto front generation algorithm allows decision makers to more easily trade-off energy and makespan to reduce operating costs and improve efficiency of HPC systems.

This work could be extended by considering alternative scalarization techniques to potentially reduce the time required to compute the lower bound. Many of the LP problems result in solutions that are identical, thus providing minimal information in forming the Pareto front. It is possible to avoid generating duplicate solutions by utilizing different scalarization techniques. The LP-based scheduling algorithm only takes a fraction of a second to compute a single schedule for a given bag-of-tasks so it is possible to use this scheduler for online batch-mode scheduling. Specifically, this algorithm can be used to schedule tasks as they arrive at the system by computing a schedule for all tasks waiting in the queue (as a batch) and recomputing the schedule when a task completes or a new task arrives.

CHAPTER 4

Maximum Profit Scheduling [1]

4.1. Introduction

4.1.1. Background. Scheduling tasks for high performance computing (HPC) systems has been a focus of much research in the last few decades. The primary goal has been to find algorithms that decrease the time required to process tasks [10]. Likewise, hardware manufacturers have been focusing on increasing performance (i.e., reducing execution time). As HPC systems have grown in computing capacity, they also have grown in power consumption. Both the power consumed by these massive supercomputers as well as the energy required to cool them has become increasingly significant [2]. In recent years, the high cost of operating these systems has lead to research that tries to find resource allocation schedules that reduce the required energy consumption to process tasks [9, 24, 37, 39, 41, 56]. While minimizing energy consumption and increasing performance is desirable, it is often not the driving factor for decision making within organizations. Often decision makers are driven to directly maximize profit. This chapter describes a novel algorithm to efficiently compute a near-optimal maximum profit schedule for extremely large problem sizes.

For a variety of reasons, HPC systems are often composed of different types of machines. Machine heterogeneity can be caused by building the HPC system in multiple phases, where each expansion phase involves purchasing a newer/different server model. Heterogeneity might also be introduced into a system from the start to decrease the run time of relatively slow tasks. For example, GPUs and specialized co-processors have been used to greatly accelerate the computation of data parallel tasks [1]. Systems composed of a non-uniform

[1]This work appeared in [55] with co-authors Anthony A. Maciejewski, and Howard Jay Siegel.

set of compute resources are called *heterogeneous computing* (HC) systems. The focus of this work is on HC systems that are heterogeneous in both performance and power consumption. For example, some tasks may execute faster on machines that support a particular CPU instruction set while another set of tasks may execute faster on machines with higher IO bandwidth. The energy consumed by a task running on a GPU enabled machine may be different than when running solely on the CPU. The nature of a task dictates how efficient, in run time and energy, it will perform on any given machine. This task and machine heterogeneity provides additional degrees of freedom that can be leveraged by the scheduling algorithms to create resource management schedules that improve the workload's run time performance and reduce the energy consumption of the overall system.

Trading-off the energy and performance (i.e., workload execution time) is difficult. There are many schedules that can be considered optimal in this trade-off space. The least execution time schedules typically require the most average power however they may or may not require more energy. Likewise the lowest energy schedule typically has a significantly reduced performance. A system administrator is required to choose the balance between these conflicting objectives. For typical scheduling scenarios, it is not desirable to have a human in the scheduling loop. This chapter focuses on combining the energy and performance objectives into a single profit objective. Profit is likely the driving factor behind the system administrator's decision so the scheduler should try to directly maximize profit. Profit combines energy and run time performance into a single more meaningful objective.

4.1.2. Motivation. Possibly the most obvious use case for the scheduling algorithms proposed in the chapter is software as a service (SaaS). For example, consider a specific video transcoding SaaS that processes requests to convert users' videos to many different formats. The user pays a fixed price for this service based on the length of the videos and the formats requested. The SaaS provider would like to complete the work as inexpensively as possible while recognizing that more work from other users is to follow and the user would prefer to have the work completed promptly. Imagine that for each video format there is a corresponding task type that represents converting one minute of a video. As such, video conversion workloads can be broken into a small and finite number of task types. In this example, there would be a large number of tasks but only tens of task types. Assume the SaaS provider also has special purpose machines that have GPUs installed that will transcode to particular video formats extremely quickly. It also has general purpose machines that can transcode all supported formats but do so more slowly. The SaaS provider can easily estimate the time to compute a task of a given task type on a machine of a given machine type. The service provider performs the same transcoding operations (e.g. convert 1 minute of MOV to MPEG-4) millions of times per day on all the different types of machines so they are likely to know the average time to compute and power consumption accurately. The SaaS provider is only paid for completed work, thus the scheduling algorithms should attempt to complete the work as fast as possible while balancing the cost of energy to do the transcoding. Other workloads such as scientific monte carlo simulations and computational biology (e.g., protein folding) also fit this computational model.

4.1.3. Contributions and Outline.

This chapter presents a monetary-based model for HPC where there exists a logical or financial distinction between the service provider (the one offering computing services) and the users (the ones submitting tasks). An algorithm is then developed to find the schedule that maximizes the profit for the service provider.

The contributions of this chapter are:

(1) A model for two-party monetary-based HPC systems.

(2) A scalable and efficient algorithm to find the near-optimal maximum profit schedule for an HC system.

(3) Bounds on the achievable profit for a given HC system.

The remainder of this chapter is organized as follows. The next section defines a monetary two-party model of computing. The fundamental algorithms to compute near-optimal schedules and the results of these algorithms are presented in Section 4.3. An efficient profit maximization algorithm is described in Section 4.4. In Section 4.5, the results are presented based on a example system configuration. Useful extensions to the two-party model and how to incorporate them into the algorithm are given in Section 4.6. Related work from the literature is in Section 4.7. Lastly, in Section 4.8, we conclude and list ideas for future work.

4.2. Two-Party Monetary Model

HPC systems are often *oversubscribed* because users of these systems typically want to complete more work than the systems are capable of completing in a timely manner. HPC systems within organizations typically have ad hoc rules governing how their employees share the compute resources. This makes it difficult to quantify optimality of schedules when there is a need to consider the monetary operating costs in the scheduling problem. Energy consumption and system performance must be converted into a space where they are comparable. Moreover, these objectives often conflict with each other. Typically one objective cannot be improved without compromising the other objective. The model in this chapter assumes there are two distinct parties. The first party is the set of users who pay money to submit work to the HPC system. The second party is the organization providing a service to the users by operating the HPC system and accepting workloads. The users and the organization are loosely coupled. The HPC system administrator is responsible for maximizing the profit from the HPC system.

Frequently HPC workloads consist of a bag-of-tasks [57]. Each task is executed on one machine and is independent of all the other tasks. Let there be a price p that a customer pays to have a bag-of-tasks processed that is based on that bag's composition. The cost to the organization for processing a bag-of-tasks is primarily the cost of electricity. Let c be the cost per unit of electrical energy. Additional operating costs such as purchase, replacement, and labor are discussed in Section 4.6.

Let the energy consumed by schedule or resource allocation $\mathbf{x}$ be $E(\mathbf{x})$. Let the time necessary to process the bag-of-tasks be $MS(\mathbf{x})$. Specifically, $MS(\mathbf{x})$ is known as the *makespan* and is defined as the maximum finishing time of all machines. The profit that the organization receives by executing a single bag-of-tasks is $p - cE(\mathbf{x})$. This is the profit per bag but it is not solely the quantity that the organization should maximize. The bag-of-tasks can take a considerable amount of time to compute when trying to increase the profit per bag-of-tasks by reducing electricity costs. Instead an organization should attempt to maximize the profit per unit time given by $\frac{p-cE(\mathbf{x})}{MS(\mathbf{x})}$, which is equivalent to

$$\frac{p}{MS(\mathbf{x})} - c\frac{E(\mathbf{x})}{MS(\mathbf{x})}\,. \tag{27}$$

The first term in Equation (27) is the average revenue per unit time. The second term is the average operating cost per unit time, or equivalently c times the average power consumption.

In this work the bag-of-tasks is simply the set of tasks from all users that are available to be run (i.e. all dependencies have been met) on the HPC system. The composition of the bag of tasks can change at any time. Furthermore the tasks can finish earlier or later than expected. The scheduler must attempt to maximize Equation (27) at all times by re-running the scheduler. Before addressing the maximum profit scheduling algorithm in Section 4.4 we first describe an algorithm to find high quality minimum energy and makespan schedules that will be used to construct the maximum profit algorithm.

4.3. Energy and Makespan Scheduling

4.3.1. Overview.

Classical scheduling algorithms consider the problem of assigning all tasks to all machines in a single large optimization problem. For the large problems being considered here, this approach is computationally prohibitive even when solving the linear relaxation (non-integer) optimization problem. The classical approach also leads to a more difficult procedure for recovering a feasible (integer) solution [7]. The approach used in our algorithms exploits the existence of groups of similar machines and groups of similar tasks to make the algorithm highly scalable. The scheduling problem is recast as a problem of assigning some number of tasks of each type to machines types or groups instead of directly assigning individual tasks to specific machines.

Profit per unit time is a function of both energy and makespan. In this section, algorithms are developed to trade-off energy and makespan. The profit maximization algorithm in Section 4.4 will employ all the key ideas and algorithms from this section. The minimum energy and makespan scheduling algorithm first computes a lower bound on the energy consumed by the machines for the schedule and a lower bound on the makespan of the schedule. The lower bound allows tasks to be split among any number of machines. This is a common practice in divisible load theory (DLT) [15]. In reality, divisible loads are not very common so it is not enough to simply find this lower bound solution. The solution to the lower bound is used to construct a complete resource allocation via a two-step process:

1) the real-valued solution is first rounded and 2) the integer number of tasks are assigned to actual machines within a machine type.

Finding the optimal schedule for makespan alone is NP-Hard in general [14], thus finding the optimal profit per unit time is NP-Hard as well. However, computing tight upper and lower bounds on the profit per unit time is still possible.

4.4. Profit Maximization Algorithm

Now we turn our attention back to the focus of this chapter, the profit maximization problem. Recall that given a full resource allocation the profit can be computed using Equation (27). One approach to determining the maximum profit solution is to compute the profit for all of the full allocations computed via the weighted sum algorithm and take the maximum. A more efficient approach is to find the maximum profit solution directly by solving a related scalar optimization problem. This section describes an efficient algorithm for finding the maximum profit schedule.

This algorithm combines the lower bounds on the energy and makespan objectives into a single profit per unit time objective. A scalar non-linear optimization problem is then formulated. This optimization problem is converted to an equivalent linear programming problem that can be easily solved. The full task allocation or schedule is reconstructed by using Algorithm 1 followed by Algorithm 2.

Given any optimal solution, $\mathbf{x}$, from the vector optimization problem of Equation (11) there exists no feasible solution that has both a energy less than $E_{LB}(\mathbf{x})$ and a makespan less than $MS_{LB}(\mathbf{x})$. Recall from Section 4.2 that p is the price (revenue) per bag-of-tasks and c is the cost per unit of energy. For a given solution, $\mathbf{x}$, an upper bound on the profit per bag is $p - cE_{LB}(\mathbf{x})$ and an upper bound on the profit per unit time is

$$\frac{p - cE_{LB}(\mathbf{x})}{MS_{LB}(\mathbf{x})}\,. \tag{28}$$

Note that dividing the largest value of the numerator by the smallest value of the denominator is the largest possible value that can be obtained. The largest profit per unit time over all feasible solutions is an upper bound on the optimal profit per unit time. This upper bound for a single $\mathbf{x}$, is further upper bounded by the maximum of Equation (28) over all possible solutions when relaxing the task indivisibility constraint. This is a very important property and drives the design of the algorithm to follow. Stated differently, there exists no feasible schedule that has a profit per unit time greater than the maximum value of Equation (28) over all possible solutions from the vector optimization problem in Equation (11). The algorithm below finds the solution that maximizes Equation (28) thus forming a true upper bound on the optimal profit per unit time for a given bag-of-tasks, p, c, and HPC system.

For any full allocation (feasible solution) the optimal profit per unit time must be greater than or equal to the profit per unit time of the full allocation. This means that any fully allocated solution is a lower bound on the optimal profit per unit time. Recall that reconstruction Algorithms 1 and 2 attempt to find a feasible solution that is close to the lower bound solution. This causes the bounds on profit per unit time to be very tight as the results in Section 4.5 will show.

Let $P_{\max}$ be the maximum allowed power consumption. This can be used to model the capacity of the cooling and/or power distribution system. While still allowing tasks to be divisible, the optimization problem to maximize the profit is

$$
\begin{aligned}
&\underset{\mathbf{x},\, MS_{LB}}{\text{maximize}} && \tfrac{p}{MS_{LB}} - c\tfrac{E_{LB}}{MS_{LB}} \\
&\text{subject to:} \quad \forall i && \textstyle\sum_j x_{ij} = T_i \\
&\qquad\qquad\quad\; \forall j && F_j \leq MS_{LB} \\
&\qquad\qquad\quad\; \forall i, j && x_{ij} \geq 0 \\
& && MS_{LB} \geq 0 \\
& && \tfrac{E_{LB}}{MS_{LB}} \leq P_{\max} \quad .
\end{aligned} \tag{29}
$$

The optimization problem is identical to Equation (11) except an upper bound on the profit is being maximized and a constraint has been added. The last constraint limits the average power consumption. This constrains the long running average of power, which is ideal for modeling a cooling system's capacity. Unfortunately, this optimization problem has a non-linearity in the objective function and in the last constraint. Recall that without idle energy $E_{LB} = \sum_i \sum_j x_{ij} APC_{ij} ETC_{ij}$. Thus the objective function and the constraint contain terms that are ratios of decision variables such as x_{ij}/MS_{LB} and $1/MS_{LB}$. Fortunately, a variable substitution can be used to transform the objective and all the constants into a linear optimization problem over a different set of decision variables. This is known as the Charnes-Cooper transformation [58]. The necessary variable substitution is

$$
r \leftarrow \frac{1}{MS_{LB}} \quad \text{and} \quad \forall i, j \quad z_{ij} \leftarrow \frac{x_{ij}}{MS_{LB}} \, . \tag{30}
$$

The variable z_{ij} can be interpreted as the average number of tasks of type i on a machine of type j per unit time and r is the number of bags per unit time. Matrix $\mathbf{z}$ and scalar r become the new decision variables in the linear optimization problem. The average power consumption, E_{LB}/MS_{LB} becomes

$$\bar{P} = \sum_i \sum_j z_{ij} ETC_{ij} \left(APC_{ij} - APC_{\emptyset j} \right) + \sum_j M_j APC_{\emptyset j} \tag{31}$$

and the profit per unit time becomes $pr - c\bar{P}$. Notice that both $\bar{P}$ and the profit per unit time are linear in decision variables z_{ij} and r. The linear optimization problem is given by

$$\begin{array}{lll} \underset{\mathbf{z},\, r}{\text{maximize}} & pr - c\bar{P} & \\ \text{subject to:} & \forall i & \sum_j z_{ij} = T_i r \\ & \forall j & \frac{1}{M_j} \sum_i z_{ij} ETC_{ij} \leq 1 \\ & \forall i, j & z_{ij} \geq 0 \\ & & r \geq 0 \\ & & \bar{P} \leq P_{\max} \qquad . \end{array} \tag{32}$$

The first four constraints in Equation (29) were converted to constraints in Equation (32) by dividing by MS_{LB} and performing the variable substitution in Equation (30).

Let the optimal solution to the linear program be $\mathbf{z}^*$ and r^*, then the optimal resource allocation and makespan can be computed as $x^*_{ij} = z^*_{ij}/r^*$ and $MS^*_{LB} = 1/r^*$. This optimal solution can then be used to recover the full allocation by applying Algorithm 1 followed by Algorithm 2. As such, this algorithm can find lower and upper bounds for the profit per unit time.

This algorithm is very desirable for extremely large scale problems because the run time of the algorithm is strongly dominated by computing the lower bound by solving a linear programming problem. The complexity of solving Equation (32) is, for a very large class of problems, polynomial in the number of nontrivial constraints, $T + M + 1$ and the number of variables, $TM + 1$ [21]. The complexity is not dependent on the number of tasks nor the number of machines. This allows the algorithm to scale to millions of tasks and machines easily so long as the number of task types and machines types remain reasonable. A complete analysis and experimentation of the computational scalability of this collection of algorithms is available in Chapter 2.

4.5. Results

4.5.1. Introduction.

Simulation experiments were performed to further verify the correctness of Section 4.4 and to quantify the quality of the resultant schedules. To test the algorithms, a representative HPC system and workload are necessary.

For these experiments the **ETC** and **APC** matrices are based on nine real systems from five power consumption benchmarks [23]. The number of tasks was increased by applying the method found in [24]. For all the simulations, there are nine machine types and 40 machines of each type for a total of 360 machines. The workload consists of 11,000 tasks divided among 30 task types. A complete description of the HPC system, including values for the **ETC** and **APC** matrices in addition to the values of T_i and M_j can be obtained supplementary material.

All experiments were performed on a mid-2009 MacBook Pro with a 2.5 GHz Intel Core 2 Duo processor. The software was written in C++ and the LP solver used the simplex method [21] from COIN-OR CLP[49].

To perform numerical experiments the price per bag p and the cost of electricity c must be given. Let $E_{\min}$ be the lower bound on the minimum energy consumed when ignoring makespan. Without loss of generality set $c = 1$ and $p = \gamma c E_{\min}$, where γ is a unitless parameter that will be used to affect the price per bag. That is, $\gamma = \frac{p}{cE_{\min}}$ is the ratio of the price per bag to the minimal operational expenses per bag. As such, $\gamma > 1$ implies that there exists a schedule such that positive profit is achievable, when tasks are divisible. Any $\gamma \geq 0$ is realizable. The parameter γ can be thought of as a *profit ratio* per bag that is governed by what the market can bear.

4.5.2. No Idle Power. For the results in this subsection, machines are modeled with no idle power consumption meaning they are turned off when not in use. In Section 4.5.3 the affect of non-zero idle power is considered.

Figure 4.1 shows different maximum profit solutions by sweeping the profit ratio. The profit ratio is proportional to the price per bag. The profit ratio is given by the number at the bottom left of each solution. The figure shows the energy and makespan of the maximum profit solution for a given profit ratio for the full allocation and for the upper bound solution from the LP problem. Notice that the upper bound and the corresponding full allocation are very close to each other. This means that the profit per unit time is very well bounded. The overall vertical length of the green bar above each solution is proportional to the profit per unit time corresponding to that schedule. The profit increases as the profit ratio increases. The knee of this curve is interesting because neither optimizing for energy or makespan alone will produce optimal profits. Also shown in Figure 4.1 is a power constraint given by the dashed line and the shaded region. When $P_{\max}$ is set to 55 kW the solutions within the shaded region satisfy the power constraint.

Figure 4.2 shows the relative decrease in profit between the upper bound and the lower bound. Each experiment uses 100 random bag-of-tasks where the task type for each task is sampled from the original task type distribution. The probability distribution for the relative decrease in profit per unit time is shown for each experiment. The width of the glyphs represent the normalized probability density of the relative profit decrease. The figure repeats this experiment for three different bag sizes represented as the average number of

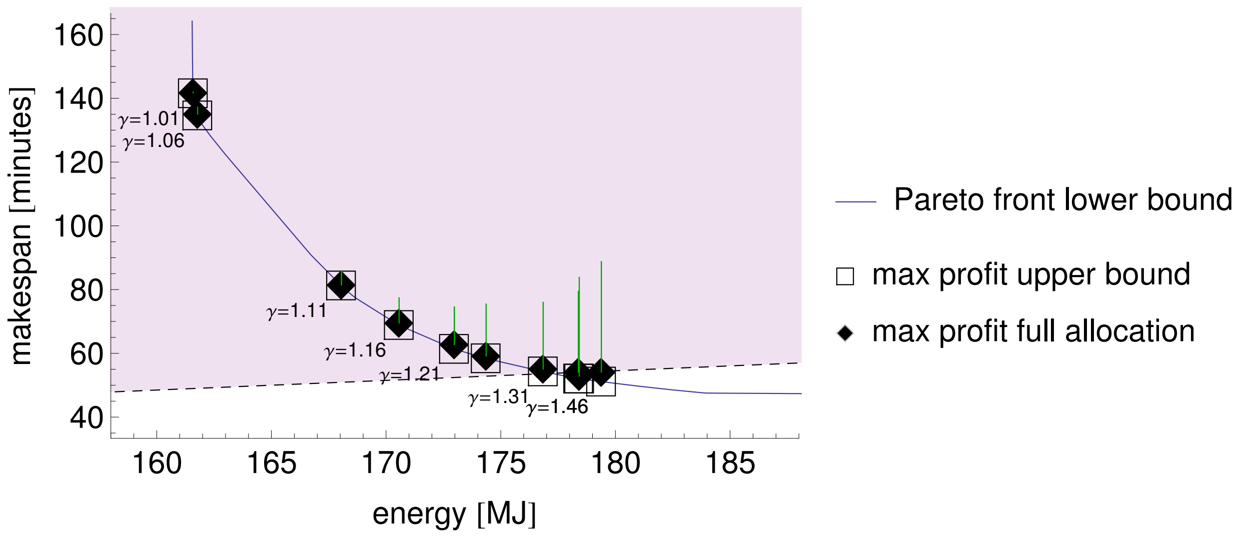


FIGURE 4.1. Parameter sweep of the profit ratio: The blue line shows the lower bound to the energy and makespan Pareto front. The shaded region shows the power constraint with $P_{\max} = 55\,\mathrm{kW}$. The height of the green bars indicates profit for the corresponding schedule which increases as the profit ratio increases. The number beside the squares is the profit ratio γ. The profit upper bound (square) and lower bound (diamond) nearly overlap indicating that there is negligible loss in energy or makespan (and thus profit) from the recovery algorithm and shows that the maximum profit is tightly bounded.

tasks per machine and profit ratios. The values of γ were chosen based on Figure 4.1. The maximum profit solution for $\gamma = 1.01$ minimizes energy alone while $\gamma = 1.5$ forces makespan to be minimized. The point where $\gamma = 1.2$ is roughly in the knee of the curve where neither minimizing only makespan or energy will find the maximum profit schedule. The average number of tasks per machine is shown on the x-axis. The y-axis shows the relative decrease in profit per unit time from the LP-based upper bound and the full allocation based lower bound. The lower the relative profit decrease, the better the approximation, and likewise, the tighter the optimal solution is bounded. As the number of tasks per machine increases, the quality of the solution improves. For $\gamma = 1.5$, minimizing makespan is the primary focus, which is more difficult than minimizing the energy. The bounds are tighter for lower profit ratios because energy is easier to minimize.

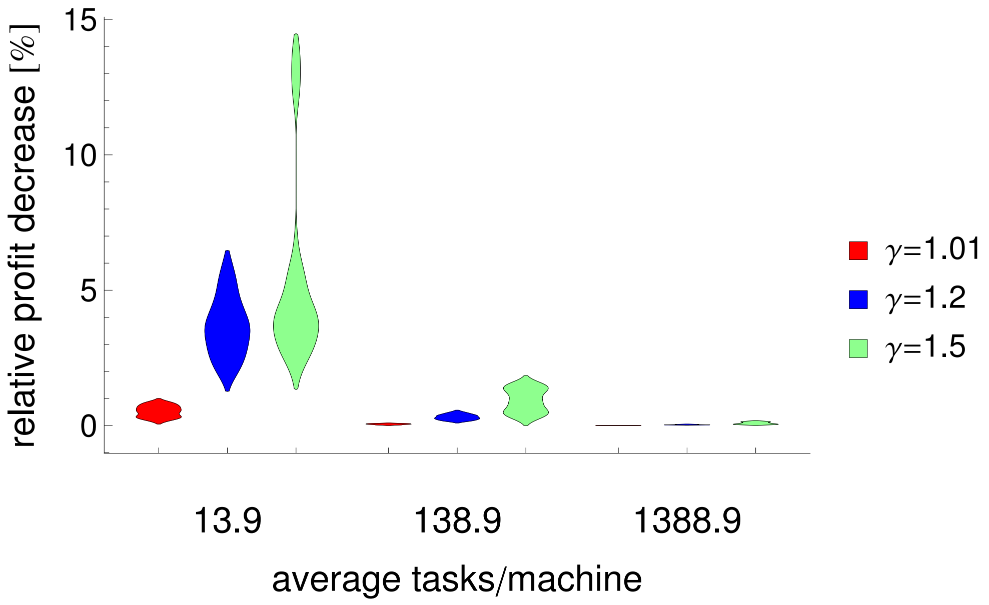


FIGURE 4.2. Probability distributions of the relative decrease in profit per unit time from the upper bound to the lower bound for various number of tasks and profit ratios: As the bag size increases the accuracy of the maximum profit solution improves. The quality of the solution is highest when the profit ratio is small.

Not only does the maximum profit algorithm produce high quality solutions but it does so extremely quickly. To find the maximum profit schedule for 10,000 tasks it takes 3.6 ms, 100,000 tasks it takes 8.9 ms, and 1,000,000 tasks it takes 74 ms. The run times are roughly linear in the number of tasks and extremely fast in all cases consider here.

4.5.3. IDLE POWER AND NEGATIVE PROFIT. To understand the effect of the idle power consumption on the algorithms, experiments were performed with the idle power set to 5% of each machine's average power consumption. Specifically, this means $APC_{\emptyset j} = \frac{0.05}{T} \sum_i APC_{ij}$. This is appropriate, for example, when modeling very-low power sleep states. The methodology used here extends to any amount of idle power from very-low power sleep states to the significantly higher power used by low frequency P-states [33]. For very high idle power consumption, the optimal schedule will always minimize makespan because in doing so energy will also be minimized, thus maximizing profit.

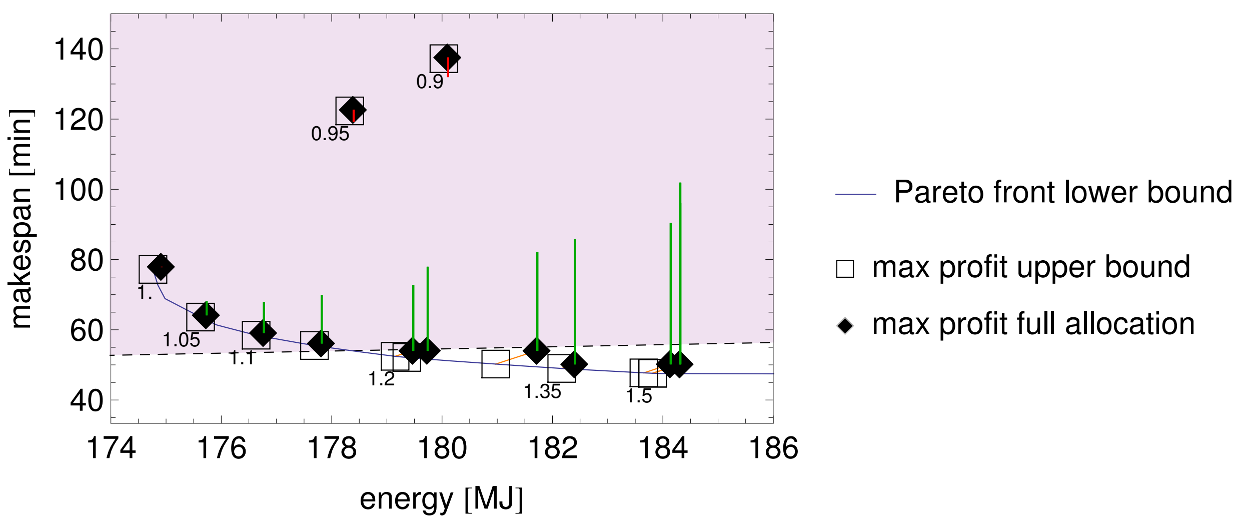


FIGURE 4.3. Parameter sweep of the profit ratio with 5% idle power: The red bars indicate a negative profit (i.e. loss) that are not along the energy and makespan Pareto front.

Similar to Figure 4.1, Figure 4.3 shows the energy, makespan, and profit for various profit ratios. The red bars indicate a negative profit or loss. The size of the downward bar indicates the magnitude of the loss.

Schedules with negative profit might need to be realized in situations where a service level agreement (SLA) is in place requiring the users' workload to be processed in spite of the loss. The loss might be caused by momentary increases in energy prices or maintenance that takes some machines offline. In any case, the schedule that minimizes the loss to the organization is highly desirable. When an SLA is not in place, the organization should choose to not process any tasks until a positive profit is once again achievable ($\gamma > 1$).

Solutions that have a loss in profit depart from the energy and makespan Pareto front as shown in Figure 4.3. This is somewhat counter-intuitive so further explanation is necessary. The key to understanding this behavior lies in the fact that Equation (32) is not optimizing profit but rather profit per unit time. This distinction is what makes the objective function more realistic but also non-linear.

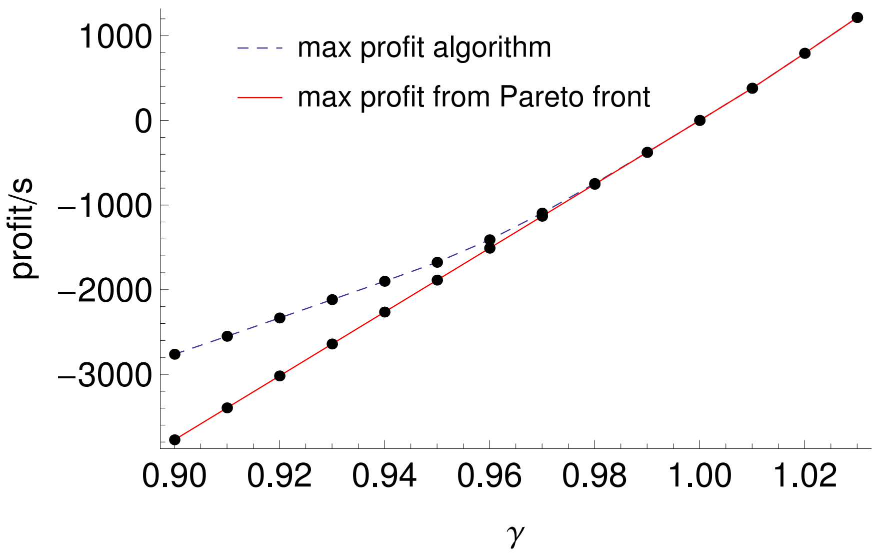
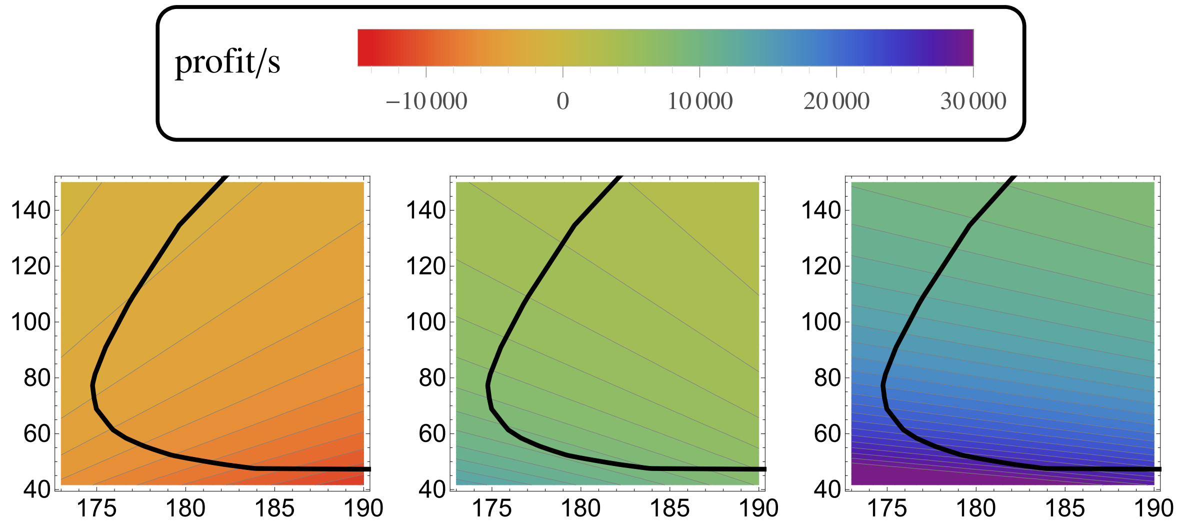


FIGURE 4.4. Profit from the maximum profit algorithm and the maximum profit from the Pareto optimal solutions: The max profit algorithm finds a higher profit solution than the maximum profit solution from the Pareto front for $\gamma < 1$.

Figure 4.4 shows the profit per unit time computed from the algorithm in Section 4.4 and the Pareto-based approach. The maximum profit solution from the Pareto front is lower than the solutions generated by the maximum profit algorithm when $\gamma < 1$. The profit is also negative for $\gamma < 1$. For $\gamma \geq 1$ the max profit solution from Section 4.4 is equal to the max profit solution along the Pareto front.

Figures 4.5a to 4.5c show the profit and the feasible solution space for profit ratios of 0.9, 1.2, and 1.5 respectively. The contours show equi-profit lines. The black line shows the boundary of the convex feasible region. Specifically, this region is the convex set defined by the constraints of Equation (29). This convex set is in the space of $\mathbf{x}$ and is projected onto the energy and makespan subspace. This projection was computed with the convex hull method described in [59]. When there is no idle power it is feasible to increase the makespan indefinitely when a minimum energy solution is sought. For this reason, without idle power the boundary of the feasible region and Pareto front have an asymptote that has infinite makespan at the minimum energy. The same affect causes the boundary of the feasible



(A) $\gamma = 0.9$ (B) $\gamma = 1.2$ (C) $\gamma = 1.5$

FIGURE 4.5. Profit per unit time for three different profit ratios: The x-axes is energy in mega joules and y-axes is makespan in minutes. The region within the curved line is the feasible region. When the price is low the optimal solution is along the top part of the feasible region which is not on the energy and makespan Pareto front.

region to have an asymptote that has infinite energy at the minimum makespan solution. However, this is not the case when idle energy is used. As the makespan increases so does the energy, thus the feasible region shown in Figures 4.5a to 4.5c is more restrictive than with no idle energy.

Figure 4.5c has a high price per bag so the maximum profit solution would minimize makespan. Positive profits are not achievable in Figure 4.5a because the profit ratio is less than unity. The non-linearity in the objective function can be seen by the lack of parallel profit contours. The minimal loss solution in this case actually tries to increase both makespan and energy. The optimal schedule slows down the processing of tasks to utilize only the most efficient machines while simultaneously decreasing the power consumption (operating expenses). This explains why a maximum profit solution is not necessarily Pareto efficient in energy and makespan.

4.6. Model Extensions

As mentioned in Section 4.2 there are other costs associated with operating an HPC system besides the cost of electricity. If conditioned power (uninterruptible power supply often with a backup generator) is used then the effective cost for electricity should be increased accordingly. Cooling of HPC systems can consume as much energy as the machines themselves depending on their geographic location and environment. *Power usage efficiency* (PUE) is a common metric used to represent the efficiency of the infrastructure within a data center. PUE is equal to the ratio of raw power to the power incident on the servers. PUE must be above one and typically is below two. To account for these inefficiencies, **APC** can be scaled by PUE. There may be an overhead cost, oh, associated with each bag-of-tasks to cover billing activities or book-keeping. These overhead costs can be modeled by subtracting $oh/MS(\mathbf{x})$ from the profit per unit time. The wear and tear on the servers can also be modeled. Let $wear_j$ be the maintenance cost per unit time of operating a machine of type j, then the effect of this cost can be modeled by subtracting

$$\frac{\sum_j F_j M_j wear_j}{MS(\mathbf{x})} \tag{33}$$

from the profit per unit time. Purchasing of hardware can be modeled as a depreciation. Let $cost_j$ and λ_j be the purchase price and mean time to failure of machine type j respectively. To model this one can subtract

$$\sum_j \frac{M_j cost_j}{\lambda_j} \tag{34}$$

from the profit per unit time.

All of these operating expenses simply modify the objective function of Equation (29) and can be converted to a linear optimization problem similar to Equation (32).

4.7. Related Work

This work focused on maximizing profit given a fixed price per bag and fixed cost for energy. In practice, the cost of energy can fluctuate and decreases during off-peak hours. Scheduling work among many data centers is shown in [56] to reduce the cost of electricity for web server workloads. Models where the price to complete an HPC workload varies based on the market are considered in [60, 61].

Our work is a generalization of the classic optimization problem of minimizing makespan and cost [14, 31]. Our approach takes advantage of the common property that each machine in an HPC system is not unique but belongs to one of a few machine types. Our work is also focused on very large-scale systems and how to find high quality solutions on average, whereas [14, 31] are concerned with worst-case performance of the scheduling algorithms.

This chapter deals with scheduling tasks to entire machines but it could also be applied to scheduling tasks to cores within a machine or across cores on many machines. The full allocation recovery algorithm we use is similar to the algorithms presented in [33] that deal with scheduling on a single machine by using dynamic voltage and frequency scaling (DVFS).

4.8. Conclusions and Future Work

As the operating costs of HPC systems grow, new scheduling algorithms are necessary to incorporate these costs into the task scheduling process. This work incorporates the concept of profit into HPC scheduling. A novel algorithm was presented that efficiently computes a near-optimal profit schedule. This algorithm computationally scales very well as the number of tasks grows. In addition, the quality of the solution actually improves as the problem size increases. The maximum profit solutions are along the energy and makespan Pareto front when there is positive profit. The profit can be negative when there is idle power consumption and the price per bag is sufficiently low. In this negative profit case, the proposed algorithm still finds the maximum profit solution which is not on the energy and makespan Pareto front.

As mentioned earlier, the price per bag in practice fluctuates based on the market. This research can be extended to model a dynamic price per bag by taking into account backlog and dynamic energy costs. This algorithm is fast enough that it can also be used for online batch scheduling where the tasks arrive randomly and the algorithm must determine on the fly how to assign tasks to machines.

CHAPTER 5

Resource Provisioning and Planning [1]

5.1. Introduction

Some HPC users are turning to cloud providers to complete their work due to the potential cost effectiveness and/or ease of use of cloud computing. The ability to provision hardware on-demand from a pre-defined set of different machine types, known as *instance types*, is very powerful. In fact, a proof of concept cluster was built by Amazon Web Services from their high performance instance types composed of over 26,000 cores with nodes connected via 10G Ethernet. This cluster ranked 101 on the Top 500 list for November 2014 [63]. Cloud infrastructure as a service (IaaS) providers [64] charge for the amount of time a virtual machine, known as an *instance*, is allocated (idle or active). This means that it is advantageous to terminate some or all instances once the workload has been processed. Leaving instances idle in the cloud is usually not cost effective. Once a new set of work needs to be processed, the decision of what instance types to start can be reevaluated each time, considering the size and composition of the workload and the current prices of the available instance types. Selecting the ideal number of instances of each instance type a user needs is challenging.

The approach to provisioning computational resources given in this chapter not only applies to cloud resource provisioning but also to selecting physical machines to purchase for use within HPC systems. The goal for provisioning HPC systems is to determine how to originally select or upgrade a system in such a way that maximizes the performance of the resultant system while meeting specific requirements that often include a budget constraint.

[1]This work is under review with co-authors Anthony A. Maciejewski, and Howard Jay Siegel [62].

The instance types available in the cloud have widely varying capabilities, by design, so that users can choose the resources that best match their workload and in doing so minimize the cost. For example, there is no need to provision high memory instance types if the workload does not require large amounts of memory. The cost for the smaller memory instance type will often be significantly less and provide nearly identical performance assuming all else is equal. Within a single IaaS provider, instance types vary in the amount of memory, number and type of CPUs, disk capacity and performance, and network performance. All of these properties of instance types affects the performance of the workload executing on the instances [65]. Due to the availability of heterogeneous resources, IaaS is inherently a HC system.

This chapter focuses on bag-of-tasks or many-task computing (MTC) workloads composed of a large number of many independent tasks. Each task is processed on a single machine. Bag-of-tasks workloads are commonly run on MTC systems [57].

In MTC and high-throughput computing (HTC), the usual goal is to maximize the number of completed tasks or jobs per unit time. In this research, a steady-state model of MTC is presented and used to formulate a linear optimization problem that can be used to optimize the number of tasks completed per unit time. In this work, types of tasks are assigned different rewards for completing. The *reward rate* is the reward earned per unit time by the system. In some situations, maximizing solely the reward rate is not desirable. Sometimes a conflicting objective such as the upgrade cost should be optimized along with the reward rate. The optimization problem we pose has multiple objectives from which any subset can be chosen, or new objectives added, as needed. The four objectives in the multi-objective optimization problem are reward rate, upgrade cost, failure rate, and power consumption.

When making a purchasing decision not all the information is necessarily available nor is the information perfectly accurate. For example, arrival rates of tasks or the performance of the machines might not be known perfectly. In fact, studies have shown that instances of the same instance type can vary significantly in performance as discussed in Section 5.2. Often only the distributional assumption can be made. That is, the probability distribution of key parameters is known but the actual value of the parameter is unknown. This uncertainty is incorporated in our steady-state model. A multi-objective stochastic programming problem formulation is presented that incorporates the uncertainty in the parameters. Stochastic programming techniques are applied to this provisioning problem to handle uncertainty.

In summary, the contributions of this chapter are:

(1) the formulation of an energy-aware steady-state model for MTC,

(2) the design of a linear optimization problem for resource provisioning (in the cloud or physical hardware procurement),

(3) a model of uncertainty and a procedure for fitting this model,

(4) a stochastic programming formulation that combines the steady-state model and the uncertainty model,

(5) an orthogonal weighted sum algorithm for generating Pareto fronts that illustrates potential trade-offs between conflicting objectives, and

(6) a performance evaluation of the stochastic programming formulation.

The rest of this chapter is organized as follows. First some related work is given in Section 5.2. The steady-state model of MTC is in Section 5.3. Section 5.4 presents the model of uncertainty and Section 5.5 the stochastic programming formulation. Quantitative measures for comparing the stochastic programming solution to other approaches is given in Section 5.6. Section 5.7 describes the heuristics implemented for comparison purposes. A technique to solve multi-objective stochastic programs is in Section 5.8. In Section 5.9, the simulation results are presented. Section 5.10 concludes this study and presents some ideas for future work.

5.2. Related Work

Scheduling resources on the cloud is not new. An interesting technique was presented in [52] that uses ordinal optimization to approximately solve a multi-objective optimization problem. The approach schedules tasks to virtual clusters in the cloud. Scheduling preemptable tasks on cloud resources has been studied recently as well [66]. Their approach uses information from the actual execution times to improve the subsequent resource allocations. The HARNESS project is currently designing algorithms and implementing software to provision resources on the cloud that benefit from highly heterogenous resources such as hardware accelerators and SSDs [67]. The uncertainty-aware scheduling approach presented here could one day be incorporated into such tools to improve the provisioning of resources.

A case study of using the cloud for HPC applications is in [68]. They show that the performance degradation due to virtualization is low but the networking performance can become a performance bottleneck if one is not careful.

The issue of HPC cluster reliability is addressed in [12]. This paper forms a bi-objective optimization problem to schedule tasks onto the cluster that has machines that vary in reliability. The author tries to minimize the maximum of all task completion times, known as the *makespan*, and maximize the reliability. The reliability measure described in Section 5.3 is similar to this measure.

We consider, in part, minimizing power consumption due to the explosive growth of power consumption in data centers and HPC systems over recent years [2]. Energy usage is becoming a major operating cost that requires algorithms to consider this from the start.

A technique to automatically scale the number of resources up or down based on the dynamic arrival of workloads is presented in [69]. Their technique utilizes the inherit heterogeneity in the cloud offerings to select instance types. In a closely related paper [70], automatic scaling is addressed. The paper models the uncertainty in the execution times of the tasks but does not consider the heterogeneous aspects of IaaS.

Task scheduling is NP-hard so reasonable approximations to model the problem, and scalable algorithms for its solution, are sought [14]. To design a scalable algorithm we use a steady-state scheduling algorithm within the resource provisioning problem. A motivation for using steady-state scheduling is given in [71]. Our steady-state model is inspired by the Linear Programming Affinity Scheduling (LPAS) algorithm [72].

The performance variation within the cloud, from instance to instance, is high compared to traditional hardware as studied in [4–6]. In [4], the measured CoV was up to 24 % for Amazon Web Service's EC2 instances. They also showed that the distribution of instance performance can be bi-modal. Co-location of instances on the same physical hardware can cause variation in performance of up to 2.5 times [6]. Another surprising finding is that IO contention between VMs can cause upto a factor of 5 in performance degradation [5]. In [73], some uncertainty in the performance of an instance was taken into account when determining both a set of instance types and task schedule via a particle swarm optimization problem.

Due to this inherit uncertainty in cloud instance performance and task arrivals, the stochastic nature of the problem cannot be ignored. Stochastic programming is the approach we take to rigorously account for the stochastic nature of the problem [74]. Stochastic programming has been used for capacity expansion in telecommunications systems [75] which is a similar application to computational resource provisioning.

5.3. Steady-State Model

Often there are millions of tasks and thousands of machines in large scale scheduling problems. To build scheduling and planning algorithms that have manageable run times for large problems, we use a steady-state formulation of the problem. This formulation focuses on the behavior of the system on average and avoids considering the scheduling of each task onto a particular machine. The steady-state model allows our algorithm's computational complexity to be independent of the number of tasks and machines in the problem. To build a compact steady-state model, we assume that the tasks of the workload can be grouped into a relatively small number of task types. All tasks belonging to a task type have similar run time and power consumption properties. This is often the case in scientific workloads. For instance, Monte Carlo simulations consist of a large number of tasks in the workload that can usually all be considered a single task type. Machines (or instances in the cloud) have a similar natural grouping called machine types (or instance types in the cloud). Task types and machine types will be used to reduce the computational complexity and enable a steady-state formulation of the problem.

The following steady-state model can be used for either determining which instance types to launch for cloud resource provisioning or determining which physical machines to purchase. Let there be T task types and M machine types. Let ETC_{ij} be the estimated time to compute (ETC) a task of type i running on a machine of type j. Likewise let APC^{d}_{ij} be the average dynamic power consumption of a task of type i running on a machine of type

j. The static power consumption of a machine of type j is given by $APC_{\emptyset j}$. The **ETC** and $\mathbf{APC^d}$ matrices are commonly used in scheduling applications for HC systems. These matrices are often found by benchmarking the tasks. To model machines being turned off when not in use let $APC_{\emptyset j} = 0$.

This steady-state model can be used for either cloud provisioning or physical hardware purchasing by correctly defining the *cost* of the machines. Let β_j^B be the buying price or *cost* of a machine of type j. For the physical hardware purchasing problem, the *cost* is the total cost of ownership of the hardware (likely on a depreciation schedule) including support and maintenance. The *cost* for a cloud instance is its cost per unit time that a user pays for running an instance of type j. Let β_j^S be the selling price of a machine of type j. The selling price is only applicable to the purchasing of physical hardware. Typically $\beta_j^B > \beta_j^S \geq 0$.

Cloud IaaS providers often limit the number of instances a user can have without prior approval. Let M_j^{cur}, M_j^{min}, and M_j^{max} be the current, minimum, and maximum number of machines of type j, respectively. Let M^{min} and M^{max} be the overall minimum and maximum number of machines, respectively. These parameters can be used to require the solution to adhere to those type of restrictions. When purchasing physical hardware these parameters may map to restrictions on the number of rack units available. Let M_j^B and M_j^S be the number of machines of type j to buy and sell, respectively. The total number of machines of type j is then $M_j = M_j^{\text{cur}} + M_j^B - M_j^S$. Due to the buying price being higher than the selling price, it makes no sense to buy and sell the same machine type.

Each task that is completed earns a reward based on the task type. Let r_i be the reward earned for completing a task of type i. The number of tasks of type i arriving per unit time is given by λ_i.

To compute the reward rate, failure rate, and power consumption, one must have a schedule that maps tasks to machines. In the steady state, it is sufficient to know the fraction of time a task of type i is running on a machine of type j. Let p_{ij} be this fraction of time. The matrix $\mathbf{p}$ is referred to as the schedule as it denotes the fraction of time each task type will be running on each machine type.

Sometimes it is useful to control the system failure rate. Machine failures when not executing a task are ignored as they have little consequence. Let ν_j be the failure rate of a machine of type j then the overall system failure rate is $\sum_i \sum_j \nu_j M_j p_{ij}$.

The optimization problem to determine the number of machines to buy and sell (i.e., $\mathbf{M^B}$ and $\mathbf{M^S}$) and the resultant schedule (i.e., $\mathbf{p}$) when maximizing the reward rate is given by:

$$\underset{\mathbf{M^B}, \mathbf{M^S}, \mathbf{p}}{\text{maximize}} \quad \sum_i r_i \sum_j \frac{1}{ETC_{ij}} M_j p_{ij} \tag{35a}$$

subject to:

$$\forall j \quad M_j^B \geq 0,\ M_j^S \geq 0 \tag{35b}$$

$$\forall i \quad \sum_j \frac{1}{ETC_{ij}} M_j p_{ij} \leq \lambda_i \tag{35c}$$

$$\forall j \quad \sum_i p_{ij} \leq 1 \tag{35d}$$

$$\forall i,j \quad p_{ij} \geq 0 \tag{35e}$$

The optimization problem in Equation (35) will maximize the the reward earned per unit time, namely the *reward rate*. The number of machines to buy and sell is required to be nonnegative, Equation (35b). The arrival rate constraint is given by Equation (35c). The machine utilization constraint, Equation (35d), ensures that machines work no more then 100 % of the time on processing tasks. The last constraint, Equation (35e), ensures that the fraction of time a machine is processing tasks is nonnegative.

The optimization problem in Equation (35) has a non-linear objective Equation (35a) and a non-linear constraint Equation (35c) due to the terms that contain $M_j p_{ij}$. Both M_j and p_{ij} are decision variables in the optimization problem. Solving non-linear optimization problems is considerably more computationally expensive than linear optimization problems. Fortunately, this non-linear problem can be transformed into an equivalent linear optimization problem. One can replace $M_j p_{ij}$ with a single variable, $\tilde{p}_{ij}$. The variable $\tilde{p}_{ij}$ can be interpreted as the effective number of machines of type j that are running tasks of type i. The constraint, Equation (35d), can be rewritten as

$$\forall_j \sum_i \tilde{p}_{ij} \leq M_j \tag{36}$$

because $M_j \geq 0$ and $p_{ij} \geq 0$.

After adding the objectives (e.g., cost, failure rate, and power) and constraints, and converting the non-linear problem in Equation (35) to a linear optimization problem, the steady-state model based optimization problem becomes

$$\text{(37a)} \quad \underset{\mathbf{M^B}, \mathbf{M^S}, \tilde{\mathbf{p}}}{\text{minimize}} \begin{pmatrix} -\sum_i r_i \sum_j \frac{1}{ETC_{ij}} \tilde{p}_{ij} \\ \sum_j M_j^B \beta_j^B - \sum_j M_j^S \beta_j^S \\ \sum_i \sum_j \nu_j \tilde{p}_{ij} \\ \sum_i \sum_j APC_{ij}^d \tilde{p}_{ij} + \sum_j APC_{\emptyset j} M_j \end{pmatrix}$$

subject to:

$$\text{(37b)} \quad \forall j \quad M_j^{\min} \leq M_j \leq M_j^{\max}$$

$$\text{(37c)} \quad M^{\min} \leq \sum_j M_j \leq M^{\max}$$

$$\text{(37d)} \quad \forall j \quad M_j^B \geq 0,\ M_j^S \geq 0$$

$$\text{(37e)} \quad \forall i \quad \sum_j \frac{1}{ETC_{ij}} \tilde{p}_{ij} \leq \lambda_i$$

$$\text{(37f)} \quad \forall j \quad \sum_i \tilde{p}_{ij} \leq M_j$$

$$\text{(37g)} \quad \forall i, j \quad \tilde{p}_{ij} \geq 0$$

$$\text{(37h)} \quad \sum_j M_j^B \beta_j^B - \sum_j M_j^S \beta_j^S \leq \beta$$

$$\text{(37i)} \quad \sum_i \sum_j \nu_j \tilde{p}_{ij} \leq \nu_{\max}$$

$$\text{(37j)} \quad \sum_i \sum_j APC_{ij}^d \tilde{p}_{ij} + \sum_j APC_{\emptyset j} M_j \leq P_{\max}$$

In Equation (37), all the objectives are to be minimized. The first objective in Equation (37a) is the negative of the reward rate. The second objective is the upgrade cost. This is the cost of purchasing machines minus the cost of selling machines. The third objective is the system failure rate. The last objective is the power consumption of the system including static power consumption.

The constraints Equations (37b) and (37c) limit the number of machines of each type and total, respectively. The constraints in the original problem correspond to Equations (37d) to (37g). It is often beneficial to include bounds on the objective functions in multi-objective optimization problems. The linear optimization problem Equation (37) has three additional constraints corresponding to a bound on the upgrade cost with budget β, a failure rate bound $\nu_{\max}$, and a power consumption bound $P_{\max}$ as constraints Equations (37h) to (37j).

The optimization problem in Equation (37) is a linear programming problem [21]. Single objectives of this problem can be quickly solved with either the simplex algorithm or interior point algorithm. A discussion of how to solve the multi-objective problem is presented in Section 5.8.

5.4. Parameter Uncertainty Model

5.4.1. Overview. Uncertainty is a fact of life. Few parameters are known perfectly, so employing Equation (37) is difficult because the effect of the unknown parameters on the solution is hard to ascertain. A model of the uncertainty in the parameters is needed to rigorously find and evaluate the solution to Equation (37). A major source of uncertainty is the execution time and power consumption of the tasks running on the various machines. Section 5.4.2 describes a model that decomposes $\mathbf{ETC}$ and $\mathbf{APC^d}$ into parts. These parts are used in Section 5.4.3 to model the randomness in the system due to uncertainty in execution time.

5.4.2. Execution Time and Power Consumption Models. This model decomposes **ETC** into a linear combination of abstract computational operations. As we will see, these abstract operations need not map to physical instructions.

Let the number of abstract operation types be L, which is as small as possible to permit a sufficiently accurate model. Let η_{il} be the number of abstract operations of type l necessary to complete a task of type i. Let τ_{lj} be the seconds per abstract operation of type l on a machine of type j. Then the $ETC_{ij} = \sum_l \eta_{il}\tau_{lj}$. In matrix form this is

$$\mathbf{ETC} = \boldsymbol{\eta}\boldsymbol{\tau} \quad . \tag{38}$$

This model can represent task heterogeneity and machine heterogeneity within the **ETC** matrix. For $L > 1$, the model allows for arbitrary task machine affinity [76]. This minimal model is a mixture model that has the necessary properties for characterizing the execution time characteristics of an HC system. The model splits the **ETC** into two components. The first component is $\boldsymbol{\eta}$ that represents the size of the tasks in terms of operations and is only dependent on the task types' properties. The second component $\boldsymbol{\tau}$ is a property of only the machine types. This decomposition is useful in representing the components of **ETC** as correlated random variables. In practice an accurate model can be found with a small value for L.

The $\mathbf{APC^d}$ matrix can be decomposed similarly. Let ψ_{lj} be the dynamic energy to execute an abstract operation of type l on a machine of type j. The energy of type l abstract operations is then $\eta_{il}\psi_{lj}$. The total energy can be computed as $\sum_l \eta_{il}\psi_{lj}$. The total average dynamic power consumption is

$$APC^d_{ij} = \frac{\sum_l \eta_{il}\psi_{lj}}{\sum_l \eta_{il}\tau_{lj}} \quad . \tag{39}$$

Equation Equation (39) can be represented in matrix form, where division is element-wise, as

$$\mathbf{APC^d} = \frac{\boldsymbol{\eta}\boldsymbol{\psi}}{\boldsymbol{\eta}\boldsymbol{\tau}} \ . \tag{40}$$

Generally only **ETC** and $\mathbf{APC^d}$ are available so the above model parameters, namely $\boldsymbol{\eta}$, $\boldsymbol{\tau}$, and $\boldsymbol{\psi}$, must be derived. These three parameter matrices have all nonnegative elements. The first step is then to compute the non-negative matrix factorization (NNMF) of **ETC** to find $\boldsymbol{\eta}$ and $\boldsymbol{\tau}$ [77]. The NNMF is similar to the singular value decomposition but the NNMF produces nonnegative matrices for the decomposition. The energy can be written as $\mathbf{APC^d} * \mathbf{ETC} = \boldsymbol{\eta}\boldsymbol{\psi}$ so we can approximate $\boldsymbol{\psi}$ via least squares. In the E3 environment, described in Section 5.9.4, the relative errors from this approach for **ETC** and $\mathbf{APC^d}$ are 0.8 % and 1.5 %, respectively.

5.4.3. Parameter Distributions. The dominant sources of uncertainty are generally from the arrival rates $\boldsymbol{\lambda}$, **ETC**, and $\mathbf{APC^d}$ parameters. If the probability distributions of the **ETC** and $\mathbf{APC^d}$ model are known then they can be used directly. If instead only the CoV is known then the following procedure can be used to estimate the variances of the elements of $\boldsymbol{\tau}$. Here we assume that all the uncertainty is in the machines and not in the tasks as was discussed in [4].

The mean and variance of the **ETC** entries are

$$\mathrm{E}[ETC_{ij}] = \sum_l \eta_{il}\mathrm{E}[\tau_{lj}] \tag{41}$$

$$\mathrm{Var}[ETC_{ij}] = \sum_l \eta_{il}^2\mathrm{Var}[\tau_{lj}] \tag{42}$$

assuming all the elements of $\boldsymbol{\tau}$ are independent. Using the definition of CoV, then substituting from Equation (42), and finally squaring both sides we have the following three equations

$$\sqrt{\mathrm{Var}[ETC_{ij}]} = \mathrm{CoV}_{ij}\mathrm{E}[ETC_{ij}] \tag{43}$$

$$\sqrt{\sum_{l} \eta_{il}^{2}\mathrm{Var}[\tau_{lj}]} = \mathrm{CoV}_{ij}\mathrm{E}[ETC_{ij}] \tag{44}$$

$$\sum_{l} \eta_{il}^{2}\mathrm{Var}[\tau_{lj}] = \mathrm{CoV}_{ij}^{2}\mathrm{E}^{2}[ETC_{ij}] \ . \tag{45}$$

Converted to matrix form this is

$$\boldsymbol{\eta}^{2}\boldsymbol{\sigma}^{2} = \mathbf{CoV}^{2} * \mathbf{ETC}^{2} \tag{46}$$

where the squares are element-wise. One can then compute $\boldsymbol{\sigma}^{2}$ via least squares. For the E3 environment described in Section 5.9.4 the relative error of this approach is 0.8 % for the variance of $\boldsymbol{\tau}$.

Nearly any probability distribution can be used within the stochastic programming formulation to model uncertainty. Even multi-modal distributions can be used. The parameters in the steady-state model are averages of execution times and arrival rates. There are no parameters that model temporal changes in arrival rate as it is a steady-state model. Thus, for our simulations we used uniform distributions for all uncertain parameters. Each uniform distribution is specified by a mean and a variance. To ensure the mean is preserved and that parameters only takes on positive values, the variance was capped as necessary.

5.5. Stochastic Programming

Stochastic linear programming is an extension of linear programming, where some of the coefficients in the objective and the constraints are random variables. For more information on stochastic programming see [78].

The particular stochastic program we use is the *recourse problem (RP)* given in standard form as

$$
\begin{aligned}
&\underset{\mathbf{x}}{\text{minimize}} && \mathbf{c}^T\mathbf{x} + \mathrm{E}_{\boldsymbol{\xi}}\left[Q(\mathbf{x},\boldsymbol{\xi})\right] \\
&\text{subject to:} && \mathbf{A}\mathbf{x} = \mathbf{b} \\
& && \mathbf{x} \geq \mathbf{0} \\
&\text{where:} && Q(\mathbf{x},\boldsymbol{\xi}) = \min_{\mathbf{y}} \mathbf{q}(\boldsymbol{\xi})^T\mathbf{y}(\boldsymbol{\xi}) \\
&\text{such that:} && \mathbf{T}(\boldsymbol{\xi})\mathbf{x} + \mathbf{W}(\boldsymbol{\xi})\mathbf{y}(\boldsymbol{\xi}) = \mathbf{h}(\boldsymbol{\xi}) \\
& && \mathbf{y}(\boldsymbol{\xi}) \geq \mathbf{0}
\end{aligned}
\tag{47}
$$

where $\boldsymbol{\xi}$ is a random vector representing the uncertain parameters.

For the RP in Equation (47), the first stage decision variable, $\mathbf{x}$ is a flattened version of $\mathbf{M^B}$ and $\mathbf{M^S}$. The second stage decisions, $\mathbf{y}$ are flattened versions of the schedule $\mathbf{p}$. The constraints of the steady-state model that contain random parameters or elements of $\mathbf{p}$, namely Equations (37e) to (37g), (37i) and (37j) are represented by $\mathbf{T}$, $\mathbf{W}$, and $\mathbf{h}$. The constraints without any random variables and thus no dependence on the scenarios, Equations (37b) to (37d) and (37h), define $\mathbf{A}$ and $\mathbf{b}$ in Equation (47). The objective coefficients are separated in a similar way. The coefficients that are deterministic are incorporated into $\mathbf{c}$ and the coefficients that are random are incorporated into $\mathbf{q}$.

This linear RP is similar to a linear program except for the expectation of the value function, $Q(\mathbf{x},\boldsymbol{\xi})$, in the objective. The RP in Equation (47) is known as a *two-stage RP*.

The optimization problem finds the optimal $\mathbf{x}$ that minimizes the sum of the linear function $\mathbf{c}^T\mathbf{x}$ and the expectation of $Q(\mathbf{x}, \boldsymbol{\xi})$.[2] The second stage optimization problem finds optimal $\mathbf{y}$ given a fixed realization of $\boldsymbol{\xi}$. The random vector $\mathbf{y}$ is known as the recourse decision vector. This RP finds a robust solution for $\mathbf{x}$ in the sense that the objective value will on average be minimal when the optimal value of $\mathbf{x}$ is used. The solution $\mathbf{x}$ is robust to unknown values of the parameters $\boldsymbol{\xi}$. The vector $\mathbf{x}$ is often referred to as a strategy of the RP.

There are many ways to solve stochastic programs. We will only describe a rather versatile approach to solving large scale stochastic programs that utilizes sample average approximation (SAA) to build the deterministic equivalent program (DEP).[3] The primary issue with stochastic programming is accurately computing the expectation in the objective function. The SAA approach uses many samples of $\boldsymbol{\xi}$ to compute the expectation as a sample average. Realizations of $\boldsymbol{\xi}$ are called scenarios. The process of creating scenarios is known as *scenario generation*. When using SAA, generating a reasonably small set of representative scenarios is important. Let there be K scenarios. For scenario k the probability of occurring is given by p_k. With SAA, the scenarios are generated by randomly sampling $\boldsymbol{\xi}$ according to its distribution, thus all the samples are equally probable. The expectation operator is linear and being applied to a linear function, namely $q^T\mathbf{y}(\boldsymbol{\xi})$. This lends itself to a linear program that is much larger, yet is equivalent to the stochastic program (in the limit as $K \to \infty$) given in Equation (47). The DEP is given by

[2]This formulation of the value function is risk neutral. Risk seeking and risk adverse formulations are also possible.

[3]The cloud resource provisioning problem is a two-stage stochastic program with relatively complete recourse [78].

$$
\begin{array}{lllllll}
\underset{\mathbf{x},\mathbf{y}_k}{\text{minimize}} & \mathbf{c}^T\mathbf{x} & +p_1\mathbf{q}_1^T\mathbf{y}_1 & \cdots & +p_K\mathbf{q}_K^T\mathbf{y}_K & & \\
\text{subject to:} & \mathbf{A}\mathbf{x} & & & & = \mathbf{b} \\
 & \mathbf{T}_1\mathbf{x} & +\mathbf{W}_1\mathbf{y}_1 & & & = \mathbf{h}_1 \\
 & \vdots & & \ddots & & \vdots \\
 & \mathbf{T}_K\mathbf{x} & & & +\mathbf{W}_K\mathbf{y}_K & = \mathbf{h}_K \\
 & \mathbf{x}, & \mathbf{y}_1, & \cdots & \mathbf{y}_K & \geq 0
\end{array} \tag{48}
$$

Each scenario (i.e., realization of the $\boldsymbol{\xi}$) defines a set of matrices $\mathbf{T}_k$ and $\mathbf{W}_k$, and vectors $\mathbf{q}_k$ and $\mathbf{h}_k$. Each scenario also introduces a new vector of decision variables y_k into the problem.

The SAA is an unbiased estimator of the mean. In practice, it converges to the mean quickly in K. The DEP can have a large number of variables and constraints. For very large problems a technique called the L-method can be used to exploit the block structure of the constraint matrix to distribute the work of solving the linear program to many nodes [79].

The problem is broken into two coupled decisions. The first is what to provision or purchase, namely $\mathbf{M}^{\mathbf{B}}$ and $\mathbf{M}^{\mathbf{S}}$. Then the random variables in the problem are realized and the second decision, known as the recourse decision, can be made. For this work, the random variables are the arrival rates, execution times, and power consumption, but virtually any other parameter in the model can be converted to a random variable. The recourse decision involves selecting the schedule $\mathbf{p}$ that is optimal for the actual arrival rates, execution times, and power consumption of the tasks.

5.6. Value of Information

To provide insight into different decision making approaches, we will evaluate other approaches besides the standard RP. Let $z(\mathbf{x}, \boldsymbol{\xi})$ be the objective value using the optimal second stage decision given the first stage decision $\mathbf{x}$ and a particular scenario $\boldsymbol{\xi}$.

The wait-and-see (WS) solutions are found by waiting until $\boldsymbol{\xi}$ is realized then computing the optimal solution. Formally, the objective value of the WS solution is given by

$$\mathcal{WS} = \mathrm{E}_{\boldsymbol{\xi}}\left[\min_{\mathbf{x}} z(\mathbf{x}, \boldsymbol{\xi})\right] \quad . \tag{49}$$

This objective value is generally unachievable because it requires perfect information about the random variable. Thus, it provides an unachievable lower bound on the problem.

The RP is found by solving Equation (47) or Equation (48). The objective value of the RP is given by

$$\mathcal{RP} = \min_{\mathbf{x}} \mathrm{E}_{\boldsymbol{\xi}}\left[z(\mathbf{x}, \boldsymbol{\xi})\right] \quad . \tag{50}$$

This objective value is achievable and is the optimal strategy or solution to the problem.

An optimization problem that is often used in lieu of the RP is the expected value problem. This problem is also know as the mean value problem (MVP) because it uses only the means of all parameters to pose the optimization problem. Specifically the objective value for the MVP is

$$EV = \min_{\mathbf{x}} z\left(x, \mathrm{E}_{\boldsymbol{\xi}}[\boldsymbol{\xi}]\right) \quad . \tag{51}$$

Let $\mathbf{x}_{EV}$ be the optimal solution to the MVP in Equation (51). To compare this solution to the WS and RP solutions one has to take the expected value over $\boldsymbol{\xi}$ of using $\mathbf{x}_{EV}$. Specifically this is

$$EEV = \mathrm{E}_{\boldsymbol{\xi}}\left[z(\mathbf{x}_{EV}, \boldsymbol{\xi})\right] \quad . \tag{52}$$

This equation uses the optimal second stage decision that is using a potentially suboptimal (based on the mean value of the parameters) decision for the first stage.

There are two standard measures used to compare these three standard approaches. The first is the expected value of perfect information (EVPI) defined as $EVPI = \mathcal{RP} - \mathcal{WS} \geq 0$. The second is the value of the stochastic solution (VSS) defined as $VSS = EEV - \mathcal{RP} \geq 0$. The EVPI is the amount the objective value for the RP, on average, would be lowered (i.e., improved) if the random vector $\boldsymbol{\xi}$ is known perfectly. VSS is the expected improvement in the objective value over the MVP if the uncertainty in the parameters is handled properly by using the RP.

5.7. Traditional Heuristic Strategies

Traditional strategies for purchasing hardware or provisioning cloud resources usually involve selecting the single machine that has the "best" desired property. Often the price and performance are used to select the machine to purchase. For comparison, we define two common heuristics followed by one heuristic specific to our problem formulation. For each heuristic, the machine type to purchase, j^*, is selected by

$$\begin{aligned}
&\text{H1:}\quad j^* = \arg\max_j \sum_i \frac{1}{ETC_{ij}} \\
&\text{H2:}\quad j^* = \arg\max_j \frac{1}{\beta_j^B} \sum_i \frac{1}{ETC_{ij}} \\
&\text{H3:}\quad j^* = \arg\max_j \frac{1}{\beta_j^B} \sum_i \lambda_i r_i \frac{1}{ETC_{ij}}
\end{aligned} \tag{53}$$

The three heuristics will be referred to as H1, H2, and H3. For the selected machine type, the next step is to find the maximum number of machines that do not violate any constraints (such as budget and power). The strategy is simply to let $M^B_{j^*}$ equal the maximum number of machines of that type that is feasible.

The first heuristic, H1 selects the machine that performs the best across all task types. Heuristic H2 uses the price and performance ratio to select the machine to purchase. Heuristic H3 weights the machine performance by the arrival rate and the reward for each task type. There are clear limitations of these heuristics in terms of flexibility and performance. H1 and H2 do not consider workload or differing reward between task types. None of the heuristics incorporate reliability or the option to potentially sell machines. The heuristics only select one machine type to purchase. As the results in Section 5.9 will show, typically the best solution is found by combining multiple machine types to match the load. None of the heuristics account for uncertainty in the parameters.

These strategies simply define the first stage of the problem. They do not describe how to schedule the tasks after such a purchasing strategy is executed. To provide a fair comparison, we use the optimal schedule from the MVP given that this particular strategy was chosen. Then we compute the expected value of the objective using Equation (52) with $\mathbf{x}_{EV}$ replaced by the purchasing decision made by the heuristic.

5.8. Multi-Objective Stochastic Programming

5.8.1. Introduction. The RP derived from the base problem in Equation (37) is a multi-objective stochastic programming problem. In this section, we extend the scalar stochastic programming problem described in Section 5.5 to the multi-objective case [80, 81].

Multi-objective optimization is challenging because there is usually no single solution that is superior to all others. Instead, there is a set of superior feasible solutions that are referred to as the *non-dominated* solutions [42]. When all objectives are to be minimized, a feasible solution x_1 dominates a feasible solution x_2 when

$$\begin{aligned} &\forall d \quad f_d(x_1) \le f_d(x_2) \\ &\exists d \quad f_d(x_1) < f_d(x_2) \end{aligned} \tag{54}$$

where $f_d(\cdot)$ is the d^{th} objective function. Feasible solutions that are dominated are generally of little interest because one can always find a better solution from the non-dominated set. The non-dominated solutions, also known as *outcomes* and *efficient points*, compose the Pareto front.

There are many techniques for solving multi-objective optimization problems. For linear optimization problems, there are two primary approaches. The first is known as Benson's algorithm that iteratively refines the Pareto front [44, 45]. The second is a technique that converts the multi-objective problem into a set of scalar optimization problems through a process called scalarization. There are many scalarization techniques but most are specializations of Pascoletti-Serafini scalarization [46], such as the weighted sum algorithm. We use the weighted sum algorithm, described in Section 5.8.2, to compute the Pareto front for the multi-objective stochastic program.

5.8.2. Weighted Sum Algorithm.

The weighted sum algorithm forms the positive convex combination of the objectives and then sweeps the weights to generate the Pareto front [82]. The optimization problem in Equation (48) is linear and convex, thus by Theorem 3.15 in [43] the weighted sum algorithm can find all of the non-dominated solutions.

Consider a D-objective optimization problem. Let the weight vector be $\boldsymbol{\omega}$ that is used to combine the objectives. To avoid a degenerate objective function, exclude $\boldsymbol{\omega} = \mathbf{0}$ from the set of weights by imposing a somewhat arbitrary constraint that the $\sum_{i=1}^{D} \boldsymbol{\omega}_i = 1$. The vector $\boldsymbol{\omega}$ is in a $D-1$ dimensional linear subspace of $\mathbb{R}^D$.

The first step in the weighted sum algorithm is to compute the *utopia* and *nadir* points. Let the optimal solution vector for objective d be $\mathbf{q}^d = \arg\min f_d(\cdot)$. The d^{th} element of the utopia and nadir points are then computed as

$$\forall_d \quad y_d^{\text{utopia}} = \min\left(f_d(\mathbf{q}^1), \ldots, f_d(\mathbf{q}^D)\right) = f_d(\mathbf{q}^d) \tag{55}$$

$$\forall_d \quad y_d^{\text{nadir}} = \max\left(f_d(\mathbf{q}^1), \ldots, f_d(\mathbf{q}^D)\right) \quad . \tag{56}$$

In other words, the utopia point is the best possible value for all objectives and the nadir point is the worst possible value from optimizing each objective individually. These two vectors are used to normalize the objective functions to better span the space. They also remove all units from the objective making the scalarized objective unitless. The normalized objective function is defined as

$$\bar{\mathbf{f}}(\mathbf{x}) = \frac{\mathbf{f}(\mathbf{x}) - \mathbf{y}^{\text{nadir}}}{\mathbf{y}^{\text{nadir}} - \mathbf{y}^{\text{utopia}}} \tag{57}$$

where the division is taken to be element-wise.

The second step in the weighted sum algorithm is traditionally done by recursively combing the objectives while ensuring that the weights sum to one [82]. Let $\boldsymbol{\omega}^1 = 1$, then the recursion is

$$\forall d = \{2, 3, \cdots, D\} \quad \boldsymbol{\omega}^d = \begin{bmatrix} \alpha_{d-1}\boldsymbol{\omega}^{d-1} \\ \\ 1 - \alpha_{d-1} \end{bmatrix} \tag{58}$$

The final weight vector is $\boldsymbol{\omega}^D \in \mathbb{R}^D$. To sweep the space, each α_d is varied uniformly from 0 to 1. This approach produces duplicate weight vectors and performs non-uniform sampling in the subspace defined by $\sum_{i=1}^{D} \omega_i = 1$.

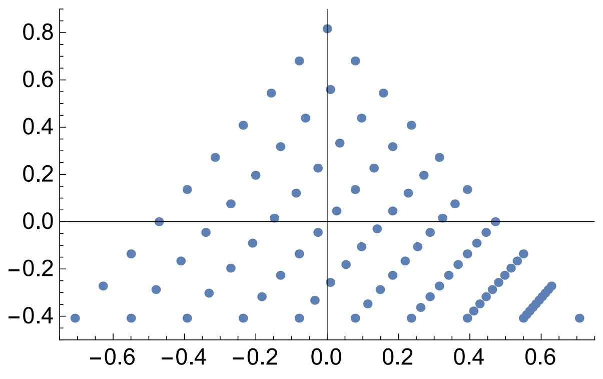
The orthogonal weighted sum algorithm finds an orthonormal basis (i.e., spanning set) for the null space of $\mathbf{1}_D$ and sweeps independently in the $D-1$ dimensional space defined by the basis vectors. Weight vectors with any negative component are dropped. This sweeps the subspace uniformly, therefore, it does not prefer any objective to any other. To ensure the whole space is swept, one must sweep α_d from $-\Delta$ to $+\Delta$ where $\Delta = \sqrt{1 - \frac{1}{D}}$.

The third step in the weighted sum algorithm is to solve the optimization problem for each weight computed in step two. Specifically, for each l solve

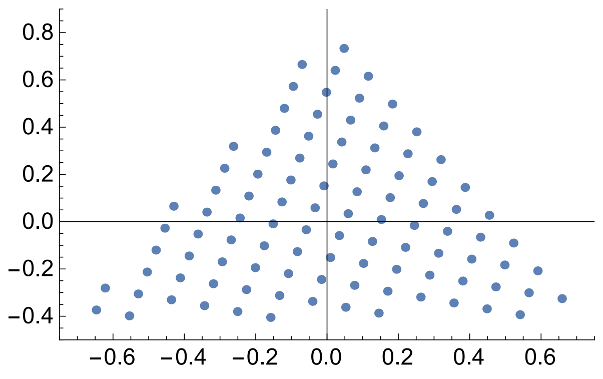
$$\min_{\mathbf{x}} \boldsymbol{\omega}_l^T \overline{\mathbf{f}}(\mathbf{x}) \tag{59}$$

to find the Pareto front.

To compare the recursive and the orthogonal sweeping methods, Figure 5.1 shows the weights in the subspace where the weights sum to one. The recursive algorithm in Figure 5.1a uses more samples in the bottom right corner than it does elsewhere. That region is no more important than the rest of the space so it is wasted sampling. Using roughly the same number of points, Figure 5.1b more uniformly distributes the samples. The orthogonal weighted sum algorithm was used to generate the Pareto fronts in Section 5.9.



(A) recursive



(B) orthogonal

FIGURE 5.1. Comparison of the recursive and the orthogonal weights drawn in the 2D plane that they span: The recursive algorithm unnecessarily puts more samples in the bottom right quadrant while the orthogonal algorithm more uniformly spans the space.

5.8.3. CONFIDENCE REGIONS. While solving a single stochastic programming problem it is useful to know the quality of the solution. To increase the quality of the solutions, the number of generated scenarios K can be increased but only at the cost of increased run time of the linear programming solver.

Confidence intervals are often used as a measure of the quality of these types of algorithms. For multi-objective optimization, a multi-dimensional confidence region is necessary. By the central limit theorem, the sample mean will converge to the multivariate normal distribution as $K \to \infty$. Let $\bar{\mathbf{y}}$ and $\mathbf{S}$ be the sample mean and unbiased covariance matrix of the samples of $Q(\mathbf{x}, \boldsymbol{\xi})$, respectively. Then the $100(1-\alpha)\%$ confidence region of the true mean, $\boldsymbol{\mu}$, is defined by

$$(\bar{\mathbf{y}} - \boldsymbol{\mu})^T \mathbf{S}^{-1} (\bar{\mathbf{y}} - \boldsymbol{\mu}) \leq \frac{(n-1)p}{(n-p)n} F_{p,n-p}(1-\alpha) \tag{60}$$

where $F_{p,n-p}(\cdot)$ is the inverse CDF of the F-ratio distribution with p numerator and $n-p$ denominator degrees of freedom [83]. The boundary of the confidence region is an ellipse in 2D and an ellipsoid in 3D. In 1D, Equation (60) collapses down to regular confidence intervals. Confidence regions will be plotted in Section 5.9.

5.9. Results

5.9.1. Overview.

Three very different environments are used to analyze the behavior of the proposed algorithms for resource provisioning. The heuristic-based algorithms H1, H2, and H3, and the RP that uses stochastic programming are compared. The first environment is a small example used to illustrate the behavior of the algorithms. The second environment is a larger environment. The third environment was built based on benchmark data. A complete description of the system parameters and simulation results are available in the supplementary material. The data is provided as CSV and JSON files and further described in the accompanying README.txt file.

The steady-state model and the scenario generation are written in C++. To generate the DEP in Equation (48) the Coin-OR Stochastic Modeling interface (SMI) is used [84]. The underlying linear programming problem is solved with the Coin-OR Linear Programming (CLP) solver [49]. CLP is a high quality open-source solver written in C++. All the simulations where run on an Apple MacBook Pro Mid 2014, 2.2 GHz Intel Core i7. The solver is single threaded so timing results are for one core.

To compare MVP, RP, and WS against the heuristics described in Section 5.7, one must be able to compute all the objectives including reward rate. The reward rate is a function of the steady-state schedule. To allow heuristics to perform as best as possible we use the optimal schedule from the steady-state model by solving a linear programming problem where the $\mathbf{M^B}$ is fixed by the heuristic and $\mathbf{M^S} = \mathbf{0}$. When computing the expected objective values the optimal schedule is used for each scenario.

Table 5.1. ETC for E1

	M1	M2	M3
T1	1	3	100
T2	100	3	1.1

Table 5.2. Solutions for E1

	H1	H2	H3	MVP	RP
M1	3.5	3.5	3.5	1.	1.9
M2	0	0	0	0	0
M3	0	0	0	2.5	1.6

5.9.2. Highly Heterogeneous Environment (E1). This environment is composed of two task types and three machine types. The **ETC** matrix is given in Table 5.1. Machine types 1 and 3 are special purpose machines and machine type 2 is a general purpose machine. Machines of type 1 can execute tasks of type 1 rapidly but are slow to process tasks of type 2. The reverse is true for other special purpose machine type (i.e., type 3).

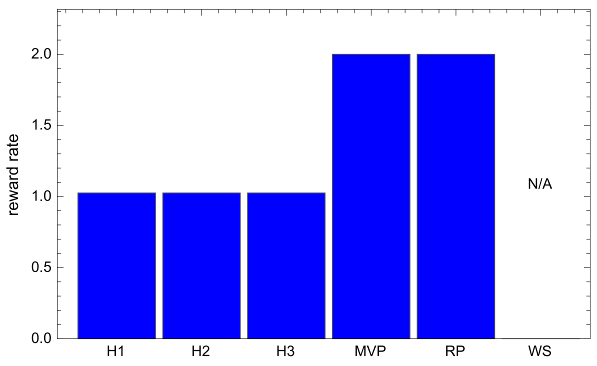
The cost for each type of machine is one per unit time (e.g, \$1/hour instance on AWS EC2 for a particular instance type). One task of each task type arrives (on average) every time unit. There are no pre-existing machines in the environment. The power and failure rate constraints are inactive. The only uncertainties in this environment are the arrival rates of the tasks. The arrival rate for tasks of type 1 is uniform from 0 to 2. For tasks of type 2 the arrival rate is uniform from $1 - 0.547$ to $1 + 0.547$. The budget is set to 3.5 so a total of 3.5 machines can be purchased.

Table 5.2 shows the number of machines of each type that the algorithms chose. All three heuristics select machine type 1 to buy because it has the highest machine performance Equation (53). Machine type 3 is the other special purpose machine but it has a slightly lower machine performance, therefore, is not selected by the heuristics. MVP and RP both select the special purpose machines but they differ on quantity of the machines.

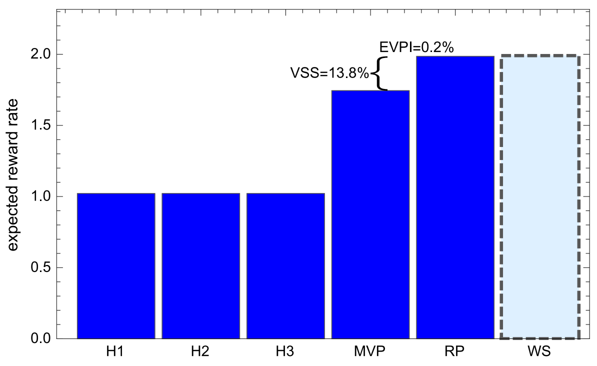
Figure 5.2 shows the performance of the three heuristics (H1, H2, and H3), MVP, RP, and WS. Figure 5.2a shows the reward rates when using the mean value of the parameters for evaluating the reward rate based on the model in Section 5.3. The WS solution is not available because there is one solution (e.g., buying strategy) for each scenario and not one solution for all scenarios. This is the same reason why the WS algorithm is not realizable but can be used as an upper bound on performance. The heuristics all achieve a reward rate of about 1.0 because they can only efficiently process tasks of type 1, the reward per task is 1.0, and the mean arrival rate is 1.0. The MVP and RP purchase machines of different types and achieve the maximum achievable reward rate of 2.0.

Figure 5.2b shows the expected value of the reward rate. This is the average reward one could expect if the given algorithm was employed to select the machines to purchase. All the solutions use the full budget that was allotted. The VSS and EVPI are also shown in the figure. The MVP and RP perform much better then the heuristics because the heuristics only choose one special purpose machine. The RP performs 13.8 % better than the MVP indicated by the VSS. This is due to RP selecting more of machine type 1 because task type 1 has a much larger uncertainty in the arrival rate compared to task type 2. In this example the EVPI is very small indicating that having fully realized parameters would not improve the solution any further then what RP already found.

The budget has a strong influence on the expected reward rate. Figure 5.3 shows the expected reward rate for the algorithms for different values of the budget. The results are averaged over ten Monte Carlo trials with 3,000 scenarios each. With no budget, there is no possibility of any reward. As the budget increases the reward increases until the maximum achievable reward rate is reached at 2.0. The heuristics flatten out at around 1.0 because they only process tasks of type 1. As the budget increases the heuristics buy more machines



(A) objective value with mean parameters



(B) expected objective value

FIGURE 5.2. Reward rates for various solutions for the E1 environment: (a) shows the reward rate computed with the mean of the parameters. (b) shows the expected reward rate over all uncertainty in the parameters.

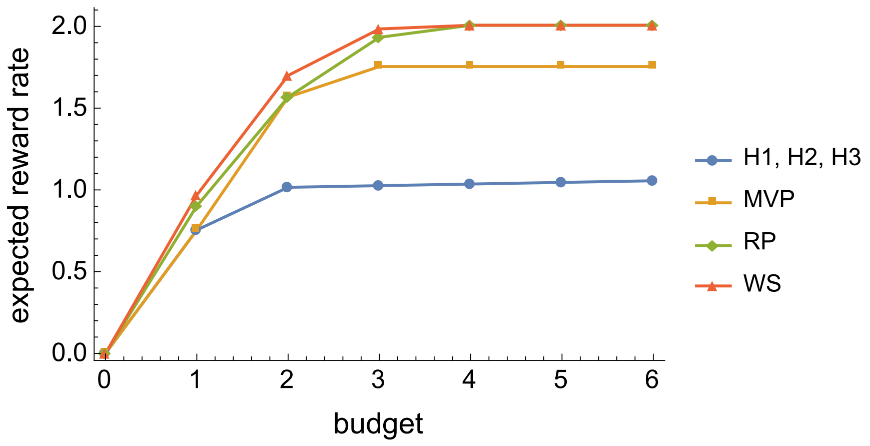


FIGURE 5.3. Expected reward rate for different budgets for the E1 environment: The reward rate asymptotes at 2.0 with RP approaching WS. The other algorithms never reach the maximum reward rate.

of type 1 but provide no value. MVP starts hitting a limit on the reward rate once it has enough budget to purchase all the necessary machines to run the average number of tasks. MVP does not consider the uncertainty in the arrival rates and so has no means of being robust against this uncertainty. RP, on the other hand, fully considers the uncertainty in the arrival rates to make the best use of the budget.

5.9.3. MEDIUM SIZED PROBLEM (E2). Environment E2 has ten task types and five machine types. The number of abstract operations is two. This environment is a representative

Table 5.3. ETC for E2

	M1	M2	M3	M4	M5
T1	5	10	10	101	30
T2	15	30	30	300	90
T3	50	101	11	1010	303
T4	50	101	100	1010	303
T5	505	1010	1001	10100	3030
T6	15	30	12	300	90
T7	55	110	110	1100	330
T8	17	34	16	340	102
T9	6	11	10	110	33
T10	5	10	10	100	30

system and workload used to show some interesting properties of the different approaches. In this environment, the arrival rates, $\boldsymbol{\eta}$, $\boldsymbol{\tau}$, and $\boldsymbol{\psi}$ are all stochastic with known means and a CoV of 100 %. There are ten machines of type five in the initial environment that can be retained or sold by the algorithms. The **ETC** matrix is given in Table 5.3.

To understand how the stochastic programming approach scales, Figure 5.4 shows the confidence in the solutions and run time for solving the RP for various numbers of scenarios. The results shown in Figure 5.4 are the average of ten Monte Carlo trials. The one-sided confidence interval in Figure 5.4a shows that a $\pm 1.2\,\%$ confidence can be obtained after about 8,000 scenarios. It takes only about four minutes to compute that solution. These algorithms are meant to be used offline so these run times are reasonable.

The solutions from the different algorithms is in Table 5.4. The heuristics only select a single machine type to buy and retain all 10 machines of type five. H2 and H3 both decided on the same machine to purchase so they have identical results in Figure 5.5. The MVP solution chose mostly machines of type three, but some type 2 machines were selected. The MVP solution decided to sell all 10 of the existing type five machines indicated by the minus sign. RP picks mostly type 2 machines but some of type 1 and 3. Only 6.7 of the existing type five machines where sold.

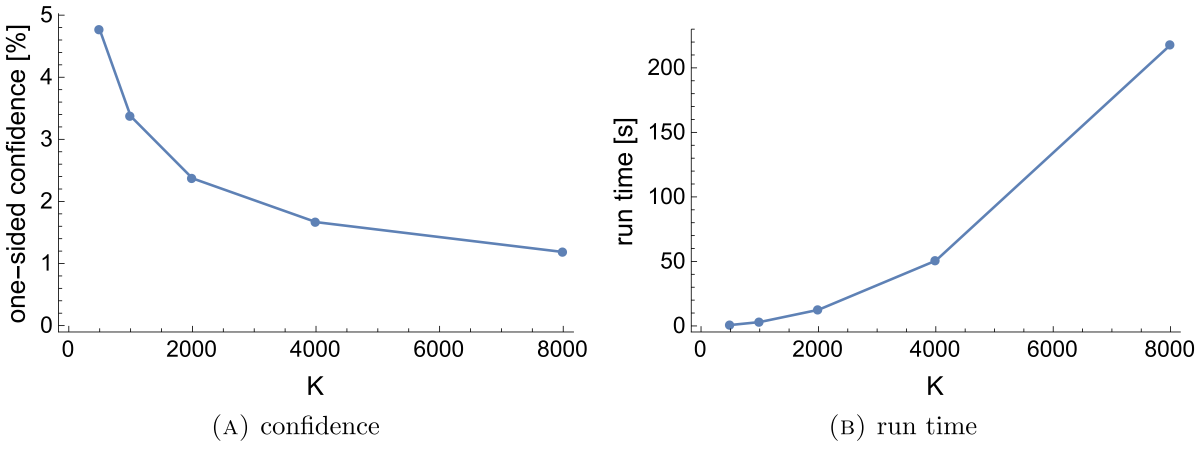


FIGURE 5.4. Confidence interval (a) and algorithm run time (b) versus the number of scenarios for the E2 environment: The quality of the solution is again acceptable after only a relatively small number of scenarios.

TABLE 5.4. Solutions for E2

	H1	H2	H3	MVP	RP
M1	31.8	0	0	0	10.
M2	0	67.3	67.3	8.5	26.9
M3	0	0	0	32.	11.5
M4	0	0	0	0	0
M5	0	0	0	-10.	-6.7

Similar to Figure 5.2, the raw objective value and expected objective value are show in Figure 5.5 for the E2 environment with 8,000 scenarios. When considering the performance of each algorithm using the mean of the parameters, Figure 5.5a, the MVP performs slightly better then all other algorithms. It even appears to outperform the RP solution. This is misleading because the mean of the parameters is not a good measure of the expected performance. It is only one possible realization of the parameters of the problem.[4] Many more realizations or scenarios are possible that are completely ignored in this measure, however this is what is commonly used by practitioners. A better measure is the true expected reward rate shown in Figure 5.5b. Due to the steady-state model and the stochastic model

[4]This is assuming the mean parameters have non-zero probability density. For multimodal or discrete distributions the mean value of the parameters might not even be realizable.

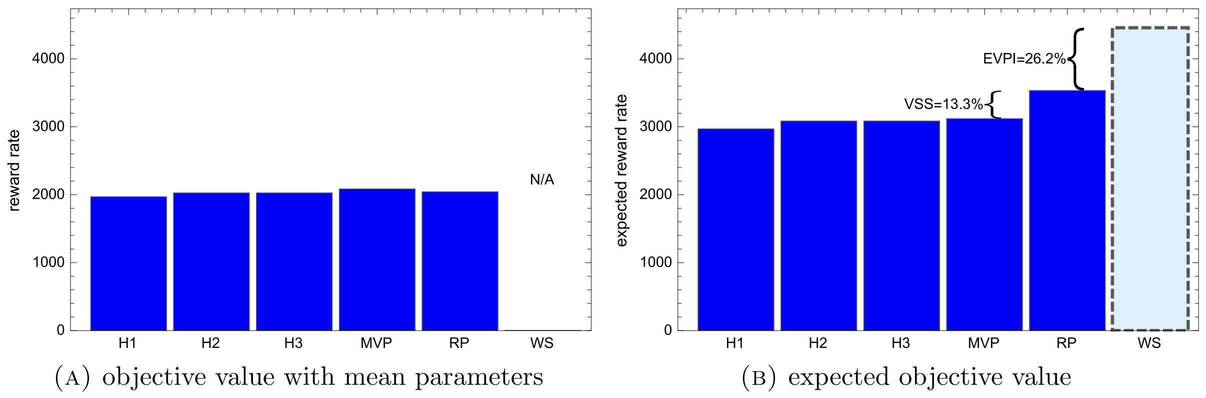


FIGURE 5.5. Reward rates for various solutions for the E2 environment: (a) shows the reward rate computed with the mean of the parameters. (b) shows the expected reward rate over all uncertainty in the parameters.

presented in Sections 5.3 and 5.4, computing the expected reward rate is easily accomplished. The RP's expected reward rate is over 13 % (given by VSS) higher then the MVP and the heuristics. If the parameters could be perfectly known at the time of making the decision then an additional 26.2 % (given by EVPI) improvement can be expected. The diversity in the machine types for the MVP and the RP allowed the resultant systems to better match the workload characteristics. In the case of the RP the workload was matched not only for the mean parameters but for nearly all possible realizations of the parameters. It is not clear how one could modify the heuristic based algorithms to better match the workload.

There are many possible scenarios that are possible in this environment. RP is guaranteed to be the best performing solution on average but that does not imply it is the best for each possible scenario. Figure 5.6 illustrates this by plotting the probability distribution of the relative improvement that RP has over the other algorithms (i.e. H1, H2, H3, MVP). The width of the glyphs represent the normalized probability density of the relative increase in reward rate. RP can out perform the H1 heuristic by up to 300 %. For some scenarios, RP can also underperform H1 by 40 %. However, on average, RP produces significantly higher reward rates as seen by the positive mean of the distribution and by Figure 5.5b.

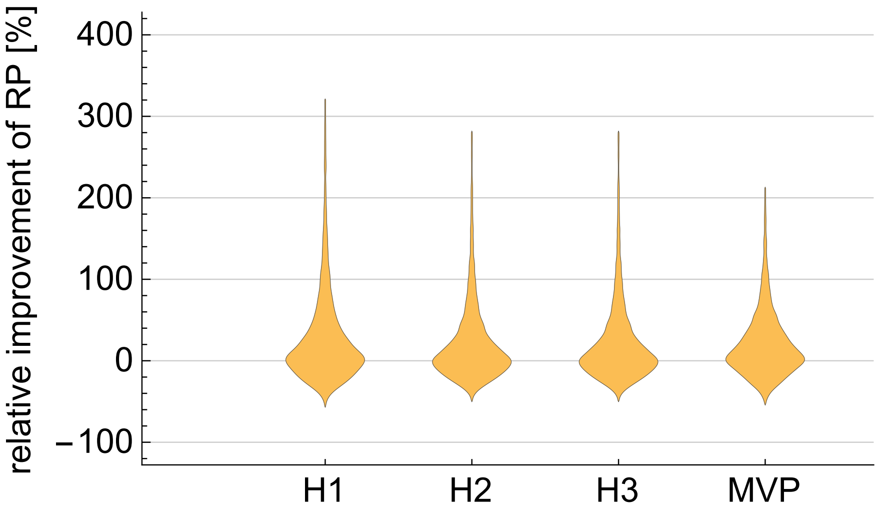


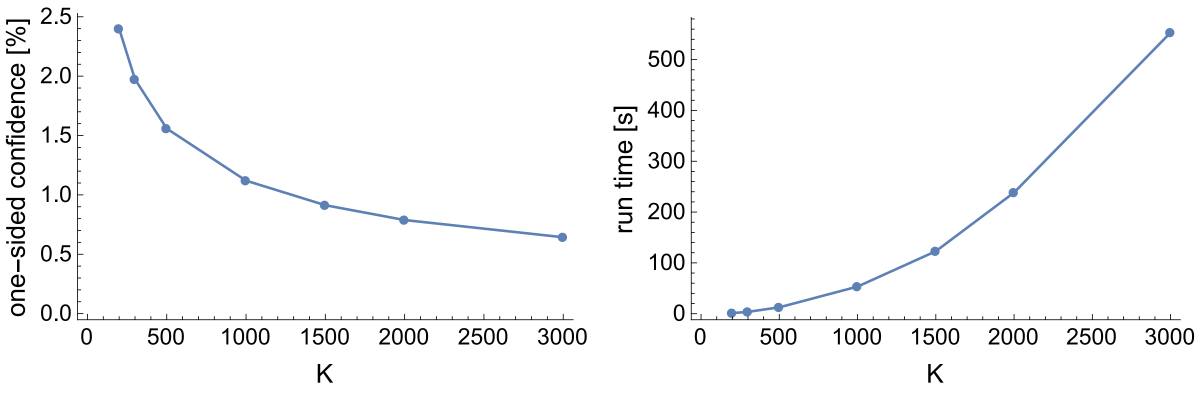
FIGURE 5.6. Distributions of the relative improvement of RP over the other algorithms for the E2 environment: The RP for certain scenarios can perform as much as 300 % times better then the heuristics, but RP also performs worse for some scenarios by as much as 40 %.

5.9.4. BENCHMARK BASED ENVIRONMENT (E3). This environment is based on a set of five benchmark applications that were run on nine different types of hardware. The execution time and power consumption was recorded for these systems [23]. The method in [24] was then used to increase the number of task types to ten. This environment has nine machine types and ten task types that define the **ETC** and $\mathbf{APC^d}$ matrices. The **ETC** matrix is shown in Table 5.5. The algorithm in Section 5.4.2 was used to generate $\boldsymbol{\eta}$, $\boldsymbol{\tau}$, and $\boldsymbol{\psi}$. Based on [4], the CoV for the **ETC** elements was taken to be 25 %. The algorithm in Section 5.4.3 is used to generate the variance of $\boldsymbol{\tau}$. The number of abstract operations, L, is three. The arrival rates have a known mean with a CoV of 25 %. There is no power constraint in this environment. The budget is \$400,000 so hundreds of machines are provisioned by the algorithms.

Figure 5.7a shows the one sided confidence interval for different number of scenarios. The results shown in Figure 5.7a are the average of ten Monte Carlo trials. At 1,500 scenarios, the error in the reward rate is under ±1 %. Figure 5.7b shows the corresponding run times

Table 5.5. ETC for E3

	M1	M2	M3	M4	M5	M6	M7	M8	M9
T1	57	28	72	45	41	19	27	28	26
T2	98	50	120	77	70	37	49	49	45
T3	463	303	362	342	314	311	303	290	264
T4	165	113	113	120	111	122	114	108	98
T5	167	91	185	129	118	74	90	88	81
T6	162	87	185	125	114	68	85	84	77
T7	45	22	57	36	33	15	22	22	20
T8	57	28	74	45	41	18	27	27	25
T9	59	36	54	44	41	34	36	35	32
T10	39	22	41	30	27	19	21	21	19



(a) confidence (b) run time

Figure 5.7. Confidence interval (a) and algorithm run time (b) versus the number of scenarios for the E3 environment: Computing an accurate solution is fast for this larger problem even though the number of machines being provisioned and scheduled is large.

for solving RP and takes less then a minute in all cases. Due to the (at least) quadratic growth of the run time w.r.t. the number of scenarios, it is important to use as few scenarios as possible. At 3,000 scenarios, the DEP is rather large with 63,011 constraints (i.e., rows) and 270,018 decision variables (i.e., columns). The constraint matrix in the DEP is very sparse so solving this large linear programming problem only consumes about 180 MB.

For these simulations, MVP, RP, and WS are all trying to maximize reward rate but as a secondary objective they are also trying to reduce the cost. This is accomplished by weighting the reward rate with 1.0 and the upgrade cost objective with a small positive

Table 5.6. Solutions for E3

	H1	H2	H3	MVP	RP
M1	0	0	0	0	0
M2	0	168.8	168.8	0	0
M3	0	0	0	0	4.3
M4	0	0	0	0	0
M5	0	0	0	0	0
M6	121.2	0	0	0	12.7
M7	0	0	0	0	4.5
M8	0	0	0	100.	73.8
M9	0	0	0	0	1.1

constant. Figure 5.8 gives the expected negative of the reward rate for 3,000 scenarios. The upgrade cost is indicated below the labels at the bottom of the graph.

Table 5.6 shows the solution for each of the algorithms. All the proposed algorithms can be solved with integer constraints on these variables however at a huge computational cost. Even though the number of machines is fractional, a simple rounding policy can easily be applied to the solutions in Table 5.6 when integers are required with negligible effect on the reward rate.

All the heuristics use up the entire budget and still perform worse then MVP and RP. The MVP solution does not use all the available budget. In fact, it only used \$291K of the budget. This is because the MVP solution does not provision for the uncertainty in the arrival rates of tasks nor in the machine performance. In the MVP formulation, there is no benefit to buying more machines once the workload can be handled hence it does not use all the available budget. The MVP has extra degrees of freedom to improve the solution but has no practical way of determining how to best select the machines. From the MVP perspective, the algorithm is already achieving maximal reward. The RP takes the uncertainty of the arrival rates and the machines into account and can achieve the same performance as the MVP solution but at a lower cost of \$274K. The RP solution uses many different types of

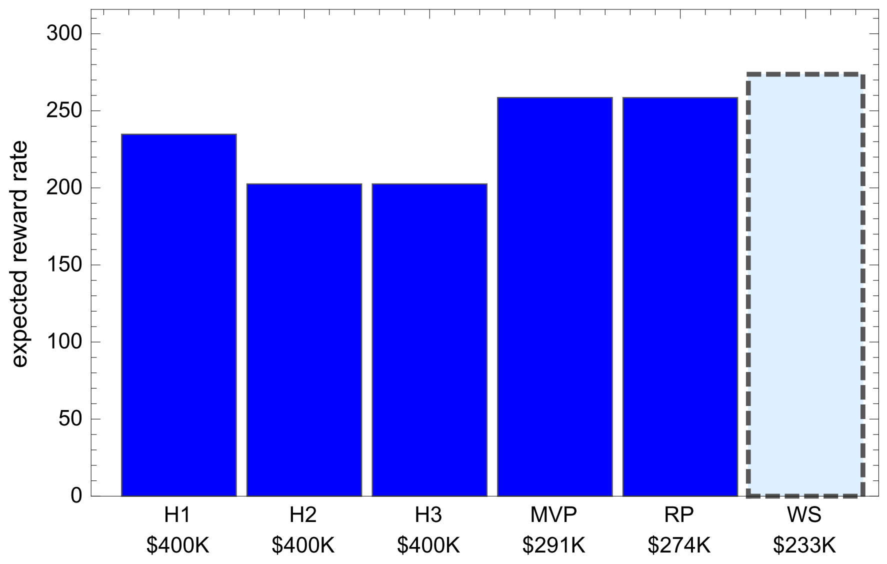


FIGURE 5.8. Expected reward rates for various solutions for the E3 environment: The cost for each solution is indicated at the bottom of the figure. RP ties for the best performance with MVP however with a solution that has a lower cost.

machines to reduce the cost and still achieve the optimal performance. The upgrade cost under the WS solution is the cost one would pay on average if they could select the machines after knowing precisely the execution time of each task type on all machine types and the arrival rates of the tasks.

5.9.5. PARETO FRONTS. Pareto fronts are useful tools to quantify the trade-off between conflicting objectives. The weighted sum algorithm described in Section 5.8.2 is used to generate the Pareto front for the reward rate and power consumption objectives. Figure 5.9 shows the Pareto front for the E3 environment with 1,000 scenarios. This took only eight minutes to compute. The solutions found by the weighted sum algorithm are blue dots. The 95 % confidence regions are shown as red ellipses centered at the blue dots. The confidence regions are relatively small. The light blue shaded region is the feasible objective space for this problem. Objective values outside this feasible space are not possible due to one or more

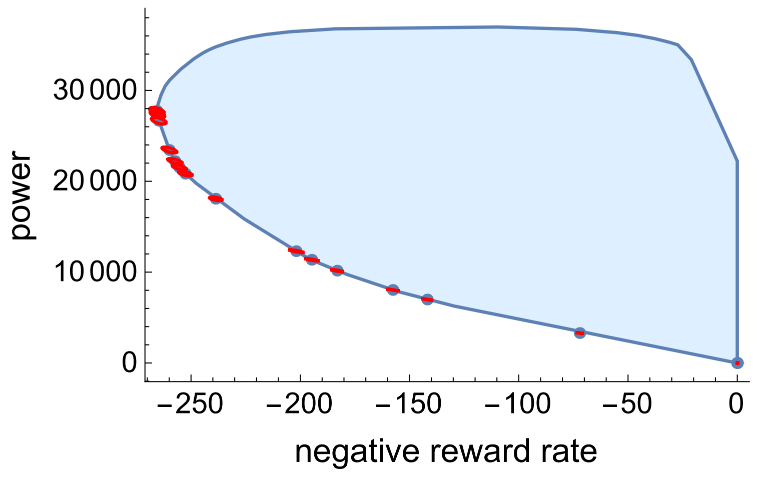


FIGURE 5.9. Reward rate and power consumption Pareto front and feasible objective space for the E3 environment: The relatively small 95 % confidence ellipses are shown in red. The blue shaded region is the feasible region of the objective space. The blue dots are solutions that the weighted sum algorithm found.

of the constraints in the problem. The feasible region is computed by solving the RP with

$$\boldsymbol{\omega} = \begin{pmatrix} \cos\theta \\ \\ \sin\theta \end{pmatrix} \tag{61}$$

for values of θ from 0 to 2π. More details on computing the feasible regions for linear programming problems is available in [59].

5.10. Conclusions and Future Work

Stochastic programming is a powerful tool that can be applied to make robust decisions in the midst of the inherit uncertainty in computing systems in both IaaS provider clouds and traditional environments. The linear steady-state model and representative stochastic model enables the use of an efficient two-stage stochastic program for solving the machine provisioning problem. The new algorithms were compared to heuristic based algorithms in a few different environments. The heuristic approaches tend to perform poorly when considering their average performance. RP produces the best quality solution on average compared to the heuristics and the MVP. The inherit uncertainty in the execution times and arrival rates necessitates algorithms that can incorporate random variables and their statistics. RP can be used to reduce the upgrade cost for a computing system by exploiting the uncertainty in the environment while maintaining the optimal level of performance. The multi-objective optimization problem can be used to quickly generate Pareto fronts.

In the future, we would like to adapt this model to include aspects of spot pricing and the uncertainty surrounding the price of the instances. We would also like to adapt these concepts to existing cloud provisioning tools and computational models. Risk adverse formulations are also possible that minimize the expected reward rate and the variance of the reward rate.

CHAPTER 6

Conclusions

Scheduling for large-scale systems is very challenging. This dissertation focused on addressing the run time and solution quality issues associated with scheduling very large numbers of tasks to large numbers of heterogeneous machines. The approach in this dissertation leverages the techniques found in divisible load theory and steady-state scheduling to improve the scaling properties of the algorithms. Many different objectives are considered in the scheduling problems such as makespan, power, profit, reward rate, reliability, and cost.

It was shown that with this new approach the quality of the solution improves as the problem size becomes large. For larger problems, these new algorithms were compared to prior art and found to be significantly faster and produce higher quality solutions.

Efficient algorithms were developed to simultaneously optimize multiple objectives to build Pareto fronts. These Pareto fronts can be used to analyze the trade-off between the conflicting objectives. Furthermore, tight upper and lower bounds on the optimal Pareto fronts were derived. These bounds were used to design a new measure of the quality of an estimate of the Pareto front. The quality measure is then used to compare Pareto front generation techniques. The Pareto front generation technique in this dissertation is faster to compute and of higher quality than prior art.

Execution times, arrival rates, and power consumption parameters are never known perfectly in reality. A stochastic model was developed to represent the important aspects of the hardware resource provisioning problem such as the correlation between execution times of different tasks running on different machines. Then an algorithm was designed to account for the uncertainty to find schedules and hardware provisions that have the highest expected value. Simulations showed that the solutions from the new algorithm produced better solutions, on average, than the prior art.

Bibliography

[1] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel AES algorithm for fast data encryption on GPU," in *2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 6, April 2010.

[2] J. Koomey, "Growth in data center electricity use 2005 to 2010," *The New York Times*, vol. 49, no. 3, 2011.

[3] K. Cameron, "Energy oddities, part 2: Why green computing is odd," *Computer*, vol. 46, pp. 90–93, March 2013.

[4] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *VLDB Endowment*, vol. 3, pp. 460–471, Sept. 2010.

[5] M. Rehman and M. Sakr, "Initial findings for provisioning variation in cloud computing," in *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 473–479, Nov 2010.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *8th USENIX Conference on Operating Systems Design and Implementation*, (Berkeley, CA, USA), USENIX Association, 2008.

[7] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Scalable linear programming based resource allocation for makespan minimization in heterogeneous computing systems," *under journal review*, Submitted 2014.

[8] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Efficient and scalable computation of the energy and makespan pareto front for heterogeneous computing systems," in *Federated Conference on Computer Science and Information Systems, Workshop on Computational Optimization*, pp. 401–408, 2013.

[9] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Efficient and scalable pareto front generation for energy and makespan in heterogeneous computing systems," in *Recent Advances in Computational Optimization: Results of the Workshop on Computational Optimization WCO 2013* (S. Fidanova, ed.), vol. 580 of *Studies in Computational Intelligence*, pp. 161–180, Springer, 2015.

[10] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810–837, June 2001.

[11] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, "Heterogeneous computing: challenges and opportunities," *Computer*, vol. 26, pp. 18–27, June 1993.

[12] E. Jeannot, E. Saule, and D. Trystram, "Optimizing performance and reliability on heterogeneous parallel systems: Approximation algorithms and heuristics," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 268–280, 2012.

[13] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, pp. 260–274, Mar 2002.

[14] K. Jansen and L. Porkolab, "Improved approximation schemes for scheduling unrelated parallel machines," *Mathematics of Operations Research*, vol. 26, no. 2, pp. 324–338, 2001.

[15] V. Bharadwaj, T. G. Robertazzi, and D. Ghose, *Scheduling Divisible Loads in Parallel and Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[16] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Computing*, vol. 6, no. 1, pp. 7–17, 2003.

[17] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, pp. 78–86, June 1993.

[18] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, pp. 42–50, Jul 1998.

[19] M. K. Dhodhi, I. Ahmad, A. Yatama, and I. Ahmad, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 62, pp. 1338–1361, Sept. 2002.

[20] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, pp. 42–50, Jul 1998.

[21] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization.* Athena Scientific, 1st ed., 1997.

[22] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[23] "Intel core i7 3770k power consumption, thermal." `http://openbenchmarking.org/result/1204229-SU-CPUMONIT081`, May 2013.

[24] R. Friese, B. Khemka, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, J. Rambharos, G. Okonski, and S. W. Poole, "An analysis framework for investigating the trade-offs between system performance and energy consumption in a heterogeneous computing environment," in *27th International Parallel and Distributed Processing Symposium Workshops and Phd Forum (IPDPSW), Heterogeneity in Computing Workshop*, pp. 19–30, IEEE Computer Society, 2013.

[25] R. S. Witte and J. S. Witte, *Statistics.* Wiley, 10 ed., 2014.

[26] 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, pp. 107–131, Nov. 1999.

[27] O. H. Ibarra and C. E. Kim, “Heuristic algorithms for scheduling independent tasks on nonidentical processors,” *Journal of the ACM (JACM)*, vol. 24, pp. 280–289, Apr. 1977.

[28] P. E. Black, “Ragged matrix.” Dictionary of Algorithms and Data Structures, NIST, December 2004.

[29] 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 Tamkang University* 50th *Anniversary Issue, Invited*, vol. 3, no. 3, pp. 195–208, 2000.

[30] J. K. Lenstra, D. B. Shmoys, and E. Tardos, “Approximation algorithms for scheduling unrelated parallel machines,” *Mathematical Programming*, vol. 46, pp. 259–271, Feb. 1990.

[31] D. B. Shmoys and E. Tardos, “Scheduling unrelated machines with costs,” in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’93, (Philadelphia, PA, USA), pp. 448–454, Society for Industrial and Applied Mathematics, 1993.

[32] M.-Y. Wu and W. Shu, “A high-performance mapping algorithm for heterogeneous computing systems,” in *Proceedings 15th International Parallel and Distributed Processing Symposium*, pp. 74–80, Apr 2001.

[33] D. Li and J. Wu, "Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms," in *41st International Conference on Parallel Processing (ICPP)*, pp. 430–439, 2012.

[34] G. Aupy, A. Benoit, and Y. Robert, "Energy-aware scheduling under reliability and makespan constraints," in *19th International Conference on High Performance Computing (HiPC)*, pp. 1–10, Dec 2012.

[35] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 962–974, 2010.

[36] M. Stillwell, F. Vivien, and H. Casanova, "Virtual machine resource allocation for service hosting on heterogeneous distributed platforms," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 786–797, May 2012.

[37] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques," *under journal review*, 2014.

[38] D. J. Brown and C. Reams, "Toward energy-efficient computing," *Communications of the ACM*, vol. 53, pp. 50–58, Mar. 2010.

[39] R. Friese, T. Brinks, C. Oliver, H. J. Siegel, and A. A. Maciejewski, "Analyzing the trade-offs between minimizing makespan and minimizing energy consumption in a heterogeneous resource allocation problem," in *INFOCOMP, The Second International Conference on Advanced Communications and Computation*, pp. 81–89, 2012.

[40] A. M. Al-Qawasmeh, A. A. Maciejewski, H. Wang, J. Smith, H. J. Siegel, and J. Potter, "Statistical measures for quantifying task and machine heterogeneities," *The Journal of Supercomputing*, vol. 57, pp. 34–50, July 2011.

[41] R. Friese, T. Brinks, C. Oliver, H. J. Siegel, A. A. Maciejewski, and S. Pasricha, "A machine-by-machine analysis of a bi-objective resource allocation problem," in *International Conference on Parallel and Distributed Processing Technologies and Applications (PDPTA)*, 2013.

[42] V. Pareto, *Cours d'economie Politique*. Lausanne: F. Rouge, 1896.

[43] M. Ehrgott, *Multicriteria Optimization*. Springer Science & Business Media, 2006.

[44] H. Benson, "An outer approximation algorithm for generating all efficient extreme points in the outcome set of a multiple objective linear programming problem," *Journal of Global Optimization*, vol. 13, no. 1, pp. 1–24, 1998.

[45] A. Löhne, *Vector Optimization with Infimum and Supremum*. Vector Optimization, Berlin, Heidelberg: Springer, 2011.

[46] G. Eichfelder, *Adaptive Scalarization Methods in Multiobjective Optimization*. Springer, 2008.

[47] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, Apr 2002.

[48] I. Y. Kim and O. De Weck, "Adaptive weighted-sum method for bi-objective optimization: Pareto front generation," *Structural and Multidisciplinary Optimization*, vol. 29, no. 2, pp. 149–158, 2005.

[49] "Coin-OR CLP." `https://projects.coin-or.org/Clp`, January 2015.

[50] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems* (K. C. Giannakoglou,

D. T. Tsahalis, J. Périaux, K. D. Papailiou, and T. Fogarty, eds.), (Athens, Greece), pp. 95–100, International Center for Numerical Methods in Engineering, 2001.

[51] S. U. Khan and C. Ardil, "A weighted sum technique for the joint optimization of performance and power consumption in data centers," *International Journal of Electrical, Computer, and Systems Engineering*, vol. 3, no. 1, pp. 35–40, 2009.

[52] F. Zhang, J. Cao, K. Li, S. U. Khan, and K. Hwang, "Multi-objective scheduling of many tasks in cloud platforms," *Future Generation Computer Systems*, vol. 37, pp. 309–320, 2014.

[53] D. Zwillinger, *CRC standard mathematical tables and formulae.* CRC press, 2011.

[54] S. P. Boyd and L. Vandenberghe, *Convex optimization.* Cambridge university press, 2004.

[55] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Energy-aware profit maximizing scheduling algorithm for heterogeneous computing systems," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Extreme Green and Energy Efficiency in Large Scale Distributed Systems Workshop (ExtremeGreen 2014), (Chicago, IL), IEEE, May 2014.

[56] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment," in *IEEE INFOCOM*, 2010.

[57] A. Iosup and D. Epema, "Grid computing workloads," *IEEE Internet Computing*, vol. 15, pp. 19–26, March 2011.

[58] A. Charnes and W. W. Cooper, "Programming with linear fractional functionals," *Naval Research Logistics Quarterly*, vol. 9, no. 3-4, pp. 181–186, 1962.

[59] T. Huynh, C. Lassez, and J.-L. Lassez, "Practical issues on the projection of polyhedral sets," *Annals of Mathematics and Artificial Intelligence*, vol. 6, no. 4, pp. 295–315, 1992.

[60] H. Zhao and X. Li, "Auctionnet: Market oriented task scheduling in heterogeneous distributed environments," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[61] M. Salehi and R. Buyya, "Adapting market-oriented scheduling policies for cloud computing," in *Algorithms and Architectures for Parallel Processing* (C.-H. Hsu, L. Yang, J. Park, and S.-S. Yeo, eds.), vol. 6081 of *Lecture Notes in Computer Science*, pp. 351–362, Springer, 2010.

[62] K. M. Tarplee, A. A. Maciejewski, and H. J. Siegel, "Robust performance-based resource provisioning using a steady state model for multi-objective stochastic programming," *under journal review*, 2015.

[63] "Top 500 List." `http://www.top500.org/system/178321`, October 2014.

[64] J. Gibson, R. Rondeau, D. Eveleigh, and Q. Tan, "Benefits and challenges of three cloud computing service models," in *Fourth International Conference on Computational Aspects of Social Networks (CASoN)*, pp. 198–205, 2012.

[65] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance implications of multi-tier application deployments on infrastructure-as-a-service clouds: Towards performance modeling," *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1254–1264, 2013.

[66] J. Li, M. Qiu, Z. Ming, G. Quan, X. Qin, and Z. Gu, "Online optimization for scheduling preemptable tasks on iaas cloud systems," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 666–677, 2012.

[67] J. G. F. Coutinho, O. Pell, E. O'Neill, P. Sanders, J. McGlone, P. Grigoras, W. Luk, and C. Ragusa, "HARNESS project: Managing heterogeneous computing resources for a cloud platform," in *Reconfigurable Computing: Architectures, Tools, and Applications*, pp. 324–329, Springer, 2014.

[68] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running HPC applications in public clouds," in *19th ACM International Symposium on High Performance Distributed Computing*, pp. 395–401, 2010.

[69] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, Nov 2011.

[70] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, Nov 2012.

[71] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Steady-state scheduling on heterogeneous clusters," *International Journal of Foundations of Computer Science*, vol. 16, no. 2, pp. 163–194, 2005.

[72] I. Al-Azzoni and D. G. Down, "Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 1671–1682, 2008.

[73] M. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Transactions on Cloud Computing*, vol. 2, pp. 222–235, April 2014.

[74] S. Ahmed, A. J. King, and G. Parija, “A multi-stage stochastic integer programming approach for capacity expansion under uncertainty.” Stochastic Programming E-Print Series, `http://www.speps.org`, 2001.

[75] M. Riis and J. Lodahl, “A bicriteria stochastic programming model for capacity expansion in telecommunications,” *Mathematical Methods of Operations Research*, vol. 56, no. 1, pp. 83–100, 2002.

[76] A. M. Al-Qawasmeh, A. A. Maciejewski, R. G. Roberts, and H. J. Siegel, “Characterizing task-machine affinity in heterogeneous computing environments,” in *International Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), Heterogeneity in Computing Workshop*, pp. 34–44, IEEE Computer Society, May 2011.

[77] D. D. Lee and H. S. Seung, “Algorithms for non-negative matrix factorization,” in *Advances in Neural Information Processing Systems*, pp. 556–562, 2001.

[78] J. R. Birge and F. Louveaux, *Introduction to Stochastic Programming.* Operations Research and Financial Engineering, Springer, 2011.

[79] R. M. Van Slyke and R. Wets, “L-shaped linear programs with applications to optimal control and stochastic programming,” *SIAM Journal on Applied Mathematics*, vol. 17, no. 4, pp. 638–663, 1969.

[80] R. B. Franca, E. C. Jones, C. N. Richards, and J. P. Carlson, “Multi-objective stochastic supply chain modeling to evaluate tradeoffs between profit and quality,” *International Journal of Production Economics*, vol. 127, no. 2, pp. 292–299, 2010.

[81] J. Teghem, D. Dufrane, M. Thauvoye, and P. Kunsch, “STRANGE: An interactive method for multi-objective linear programming under uncertainty.,” *European Journal of Operational Research*, vol. 26, pp. 65–82, 1986.

[82] I. Y. Kim and O. De Weck, "Adaptive weighted sum method for multiobjective optimization: a new method for pareto front generation," *Structural and Multidisciplinary Optimization*, vol. 31, no. 2, pp. 105–116, 2006.

[83] V. Chew, "Confidence, prediction, and tolerance regions for the multivariate normal distribution," *Journal of the American Statistical Association*, vol. 61, no. 315, pp. 605–617, 1966.

[84] "Coin-OR SMI." `https://projects.coin-or.org/Smi`, January 2015.

Glossary

APC: average power consumption. 41

bag-of-tasks: a collection of independent tasks that may be executed in any order. 4, 9, 14, 20, 28, 36, 38, 42, 76, 98

CoV: coefficient of variation. 24, 59, 101, 109, 110, 125, 128

CSV: comma separated value. 121

DEP: deterministic equivalent program. 112, 113, 121, 129

DLT: divisible load theory. 6, 7, 39, 41, 134

DVFS: dynamic voltage and frequency scaling. 35, 68

ETC: estimated time to compute. viii, 8, 41, 102, 122, 125, 129

EVPI: expected value of perfect information. 115, 123, 127

GA: genetic algorithm. 35, 52

HC: heterogeneous computing. 1, 4, 6, 38, 40, 98, 103, 108

HPC: high-performance computing. 1–5, 17, 22, 25, 27, 34, 37–39, 59, 66, 68, 75, 97, 100, 101

HTC: high-throughput computing. 98

IaaS: infrastructure as a service. 97, 98, 101, 103, 132

JSON: javascript object notation. 121

linear program: linear programming problem. 9, 43

LP: linear programming. ix, 5, 7, 9, 14, 15, 17, 19, 23, 25, 27, 29, 30, 34, 39, 41, 43, 44, 51, 57–59, 63–68, 74–76

LPT: longest processing time. 12, 13

machine type: machines that have identical relevant properties such as run time and power consumption across all tasks. 3, 6, 40, 97, 102, 104, 108, 115, 116, 122, 124, 128

makespan: the maximum of all the finishing times across all machines. 4, 5, 22, 38

max-min: maximum MCT scheduling algorithm. ix, 18, 22–25, 34, 36

MCT: minimum completion time. 19, 20, 23, 148

MET: minimum execution time. ix, 14–16

MILP: mixed integer linear programming. 69

min-min: minimum MCT scheduling algorithm. ix, 18–25, 34, 36, 52

MTC: many-task computing. 98, 99

MVP: mean value problem. 114–116, 121–127, 129–131, 133

NNMF: non-negative matrix factorization. 109

NSF: National Science Foundation. iii

NSGA-II: non-dominated sorting genetic algorithm II. 39, 52, 57–59, 64, 65, 69, 75

ready time: time a machine has completed previously scheduled work and is ready for new work. 13

RP: recourse problem. xi, 111, 112, 114–116, 121–133

SAA: sample average approximation. 112, 113

scenario: single realization of the random variables used in a stochastic programming problem. A scenario represents one possible outcome of a random trial.. 112

SPEA2: improved strength pareto evolutionary algorithm. 69

static scheduling: resource management where the collection of tasks and the available set of machines is fixed and known at the time of scheduling the tasks. 4, 5, 7, 34, 38, 40, 41

task type: tasks that have identical relevant properties such as run time and power consumption across all machines. 7, 40, 102–104, 108, 116, 122, 124, 128

VSS: value of the stochastic solution. 115, 123, 127

WS: wait-and-see. 114, 121, 123, 124, 129, 131