

Efficient and Scalable Computation of the Energy and Makespan Pareto Front for Heterogeneous Computing Systems

Kyle M. Tarplee*, Ryan Friese*, Anthony A. Maciejewski*, and Howard Jay Siegel*[†]

*Electrical and Computer Engineering Department

[†]Computer Science Department

Colorado State University

Fort Collins, CO 80523

Email: {kyle.tarplee, ryan.friese, aam, hj}@colostate.edu

Abstract—The rising costs and demand of electricity for high-performance computing systems pose difficult challenges to system administrators that are trying to simultaneously reduce operating costs and offer state-of-the-art performance. However, system performance and energy consumption are often conflicting objectives. Algorithms are necessary to help system administrators gain insight into this energy/performance trade-off. Through the use of intelligent resource allocation techniques, system administrators can examine this tradeoff 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. A novel algorithm is presented that efficiently computes tight lower bounds and high quality solutions for energy and makespan. These solutions are used to bound the Pareto front to easily trade-off energy and performance. These new algorithms are shown to be highly scalable in terms of solution quality and computation time compared to existing algorithms.

I. INTRODUCTION

THE race for increased performance in high-performance computing (HPC) systems has resulted in a large increase in the power consumption of these systems [1]. 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 [2]. 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 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 desirable run times 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.

The focus of our work is on a common scheduling problem where the users submit a set of independent tasks known as a *bag of tasks* [3]. The tasks will run on a dedicated set of interconnected machines. A task runs on only one machine and, likewise, a machine may only process one task at any one time. This class of scheduling problems is often referred to as *static scheduling* because the full bag of tasks is known a priori [4]. Task execution and power consumption are deterministic in this model. The HPC systems of primary interest have highly heterogeneous task run times, machines, and power consumption which are known as *heterogeneous computing* (HC) systems. Some machines in the HC systems are often special purpose machines that can perform specific tasks quickly, while other tasks might not be able to run at all on that hardware. Another cause of heterogeneity is differing computational requirements, input/output bottlenecks, or memory limitations, and therefore cannot take full advantage of the machine. The machines may further differ in the average power consumed for each task type. Machines may have different architectures, leading to vastly different power consumption characteristics. For instance, a task that runs on a GPU might consume less energy to complete, but often more power, than the same task run on a general purpose machine, due to the shorter execution time. We assume one objective is to minimize the maximum finishing time of all tasks, which is known as the *makespan*. The heterogeneity in execution time of the tasks provides the scheduler degrees of freedom to greatly improve the makespan over a naive scheduling algorithm. Similarly the heterogeneity in the power consumption allows the schedulers to decrease the energy consumption.

The contributions of this paper are:

- 1) The formulation of an algorithm that efficiently computes tight lower bounds on the energy and makespan and quickly recovers near optimal feasible solutions.
- 2) Finding a high quality bi-objective Pareto front.
- 3) An evaluation of the scaling properties of the proposed algorithms.

- 4) The addition of idle power consumption to the formulation of the energy/makespan problem in [3].

The rest of this paper is as follows: first the lower bound on the objectives is described in Subsection II-B. Then algorithms are presented in Subsections II-C, II-D, and II-E that reconstruct a feasible schedule from the lower bound. In Subsection II-F, the complexity of the algorithm is analyzed. Algorithm scaling quality and runtime results are shown in Section III. Section IV shows how these bounds can be used with any scalarization technique to form a Pareto front. Section V compares these algorithms to the NSGA-II algorithm.

II. APPROXIMATION ALGORITHMS

A. Approach

The fundamental approach of this paper is to apply *divisible load theory* (DLT) [5] to ease the computational requirements of computing a lower bound solution on the energy and makespan. For the lower bound, a single task is allowed to be divided and scheduled onto any number of machines. After the lower bound on the energy and makespan is computed, a two phase algorithm is used to recover a feasible solution from the infeasible lower bound solution. The feasible solution serves as the upper bound on the optimal energy and makespan.

Often HC systems have groups of machines, usually purchased at the same time, that have identical or nearly identical performance and power characteristics. Even when every machine is different, the uncertainty in the system often allows one to model similar machines as groups of machines of the same type. A *machine group* is a collection of machines that have virtually indistinguishable performance and power properties with respect to the workload. Machines within a machine group may differ vastly in feature sets so long as the task 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. In fact, having groupings for tasks and for machines permits less profiling effort to estimate the run time and power consumption for each task on each machine.

Traditionally this 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. The decision variables would be binary valued (assigned or not assigned) 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. Instead, the problem is posed as determining the number of tasks of *each type* to assign to each *machine group*. 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 holds well when the number of tasks assigned to each machine group is large. For this approximation, machine groups need not be large. In addition to easing the recovery of the integer solution, another benefit

of this formulation is that it is much less computationally intensive due to solving the higher level assignment of tasks types to machine groups 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.

B. Lower Bound

The lower bound is given by the solution to a linear bi-objective optimization (a.k.a. vector optimization) problem and is constructed 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 x_{ij} be the number of tasks of type i assigned to machine group j , where x_{ij} 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 for task type i on machine type j . Similarly let **APC** be a $T \times M$ matrix where APC_{ij} is the average power consumption for task type i on machine type j . These matrices are frequently used in scheduling algorithms [4], [6]–[8].

The lower bound of the finishing time of a machine group is found by allowing tasks to be divided among all machines to ensure the minimal finishing time. With this conservative approximation all tasks in machine group j finish at the same time, namely:

$$F_j = \frac{1}{M_j} \sum_i x_{ij} ETC_{ij}. \quad (1)$$

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 group, the tightest lower bound on the makespan is:

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

Energy consumed by the bag of tasks is $\sum_i \sum_j x_{ij} APC_{ij} ETC_{ij}$. To incorporate idle power consumption, one must have a time duration for machines not running any tasks. In this model, the makespan is used for the time the machines' power must be accumulated. Not all machines will finish executing tasks at the same time. All but the last machine(s) to finish will accumulate idle power. When no task is executing on machine j , the power consumption is given by the idle power consumption, APC_{0j} . The equation for the lower bound on the energy consumed, incorporating idle power, is given in:

$$\begin{aligned} E_{LB} &= \sum_i \sum_j x_{ij} APC_{ij} ETC_{ij} \\ &\quad + \sum_j M_j APC_{0j} (MS_{LB} - F_j) \\ &= \sum_i \sum_j x_{ij} ETC_{ij} (APC_{ij} - APC_{0j}) \\ &\quad + \sum_j M_j APC_{0j} MS_{LB} \end{aligned} \quad (3)$$

where the second term in the first equation accounts for the idle power.

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

$$\begin{aligned} & \underset{x_{ij}, \text{MS}_{\text{LB}}}{\text{minimize}} && \begin{pmatrix} E_{\text{LB}} \\ \text{MS}_{\text{LB}} \end{pmatrix} \\ & \text{subject to:} && \forall i \quad \sum_j x_{ij} = T_i \\ & && \forall j \quad F_j \leq \text{MS}_{\text{LB}} \\ & && \forall i, j \quad x_{ij} \geq 0 \end{aligned} \quad (4)$$

The objective of (4) is to minimize E_{LB} and MS_{LB} , where \mathbf{x} is the primary decision variable. MS_{LB} is an auxiliary decision variable necessary to model the objective function in (2). The first constraint ensures that all tasks in the bag are assigned to a machine group. 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 groups. 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 IV.

Ideally this vector optimization problem would be solved optimally with $x_{ij} \in \mathbb{Z}_{\geq 0}$. However, for practical scheduling problems, finding the optimal integral solution is often not possible due to the high computational cost. Fortunately, efficient algorithms to produce high quality sub-optimal solutions exist.

C. Allocation Reconstruction

For infeasible solutions obtained from (4), an algorithm is necessary to recover from each a feasible solution or full allocation. Numerous approaches have been proposed in the literature for solving integer LP problems by first relaxing them to real-valued LP problems [9]. The approach here follows this common technique combined with computationally inexpensive techniques tailored to this particular optimization problem. The problem is broken up into two phases. The first phase rounds the solution while taking care to maintain feasibility of (4). The second phase assigns tasks to actual machines to build the full task allocation.

D. Rounding

Due to the nature of the problem, the optimal solution \mathbf{x}^* often has few nonzero elements per row. Usually all the tasks of one type will be assigned to a small number of machine groups. In the original problem, tasks are not divisible so one needs to have an integer number of tasks to assign to a machine group. When all the tasks of a given task type are assigned to one machine group, that row of \mathbf{x} has one nonzero value which is equal to T_i , an integer. When tasks are split between machine groups, an algorithm is needed to compute an integer solution from this real-valued solution. The following algorithm finds $\hat{x}_{ij} \in \mathbb{Z}_{\geq 0}$ such that it is near x_{ij}^*

while maintaining the task assignment constraint. Algorithm 1 finds $\hat{\mathbf{x}}$ that minimizes $\|\hat{x}_{ij} - x_{ij}^*\|_1$ for a given i .

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

```

1: for  $i = 1$  to  $T$  do
2:    $n \leftarrow T_i - \sum_j \lfloor x_{ij}^* \rfloor$ 
3:    $f_j \leftarrow x_{ij}^* - \lfloor x_{ij}^* \rfloor$ 
4:   Let set  $K$  be the indices of the  $n$  largest  $f_j$ 
5:   if  $j \in K$  then
6:      $\hat{x}_{ij} \leftarrow \lceil x_{ij}^* \rceil$ 
7:   else
8:      $\hat{x}_{ij} \leftarrow \lfloor x_{ij}^* \rfloor$ 
9:   end if
10: end for

```

Algorithm 1 operates on each row of \mathbf{x}^* 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 that must be assigned to machine j . The algorithm simply rounds up those n assignments that have the largest fractional parts. Everything else is rounded down. The result is an integer solution $\hat{\mathbf{x}}$ that still assigns all tasks properly and is near to the original solution from the lower bound.

E. Local Assignment

The last phase in recovering a feasible assignment solution is to schedule the tasks already assigned to each machine group to actual machines within that group. This scheduling problem is much easier than the general case because all machines in a group are the same. This problem is formally known as the multiprocessor scheduling problem [10]. One must schedule a set of heterogeneous tasks on a set of identical machines. The *longest processing time* (LPT) algorithm is a very common algorithm for solving the multiprocessor scheduling problem [10]. Algorithm 2 uses the LPT algorithm to independently schedule each machine group.

Algorithm 2 Assign tasks to machines using LPT per machine group

```

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

```

Each column of $\hat{\mathbf{x}}$ is processed independently. List z contains task type i , \hat{x}_{ij} times. The tasks are then sorted

in descending order by execution time to find y . Next the algorithm loops over y one element (task) at a time and assigns it 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. Algorithms exist that will produce a more optimal solution, but it will be shown that the effect of the sub-optimality of this algorithm on the overall performance of the systems considered is insignificant.

F. Complexity Analysis

The complexity analysis of this algorithm shows some desirable properties that are now discussed. One must solve a real-valued LP problem to compute the lower bound. 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. 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 Algorithm 1 is $T(M \log M)$. The task assignment algorithm outer loop is run M times. Inside this loop there are two steps. The first step is sorting $n_j = \sum_i x_{ij}$ items which takes $n_j \log n_j$ time. The second step is a loop that iterates n_j times and must find the machine with the earliest ready time each iteration, which is a $\log M_j$ time operation. The worst case complexity of Algorithm 2 is thus $M \max_j (n_j \log n_j + n_j \log M_j)$.

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 Algorithm 1 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 systems. Millions of tasks and machines can be handled easily so long as the machines can be reasonably placed in a small number of homogeneous groups and, likewise, tasks can be grouped by type. Only the upper bound'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. Furthermore, Algorithm 2 can be trivially parallelized because each machine group is scheduled independently. The lower bound can be used to analyze much of the behavior of the system at less computational cost.

III. SCALING RESULTS

An important property of a scheduling algorithm is its ability to scale well as the size of the problem grows. Simulation experiments were carried out to quantify how the relative error and the computational cost of the algorithm scales. These experiments are used to validate the complexity analysis results from Section II-F. **ETC** and **APC** are needed to test

the algorithms. **ETC** and **APC** are based on a set of five benchmarks executed over nine machine types [11]. Then the method found in [7] was used to construct larger **ETC** and **APC** matrices. Nominally there are 1100 tasks made up of 30 task types. There are 36 machines made up of nine machine types. A complete description of the systems and output from the algorithms are available in [12].

The number of tasks, task types, and machine types are swept independently to generate a family of figures. For this size system it is too expensive to solve for the optimal makespan but one can compare bounds on the makespan to gain insight into the algorithm. Each of the three parameter sweeps is computed by taking random subsets with replacement to handle the sweep variable. These results are averaged over 50 Monte Carlo trials. The experiments were performed on a mid-2009 MacBook Pro with a 2.5 GHz Intel Core 2 Duo processor. The code is written in Mathematica 9 and the LP solver uses the simplex method which forwards to the C++ COIN-OR CLP solver [13]. The scaling experiments all optimize makespan while ignoring the energy objective.

Fig. 1 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 original nine machine system. The relative increase in makespan is shown from the 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 Algorithm 2. However, Fig. 1 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 increase.

The run time of the scheduler as a function of the number of tasks is shown in Fig. 2 to quantify computational efficiency of the various algorithms. The blue (bottom) portion of the graph is the time taken to compute the lower bound (solve the LP problem). The green (middle) portion is the time it takes 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. The red (top) portion of the figure shows the full allocation that seems to scale linearly with the number of tasks. Recall that the complexity of Algorithm 2 has a dependency on the number of tasks which is linear or log linear depending on the parameters.

Fig. 3 shows the same three curves as Fig. 1, however this time varying the number of task types. The number of tasks, machines, and machine types are held constant for this experiment. Fig. 3 shows that again the local assignment algorithm is causing most of the degradation in makespan. The relative error in makespan does not tend to zero because increasing the number of task types does not improve the quality of the approximation.

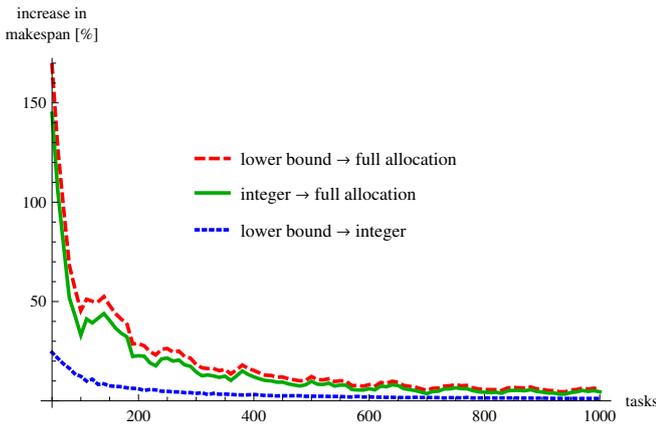


Fig. 1. Relative percent increase in makespan as a function of the total number of tasks: The quality of the solution improves as more tasks are used.

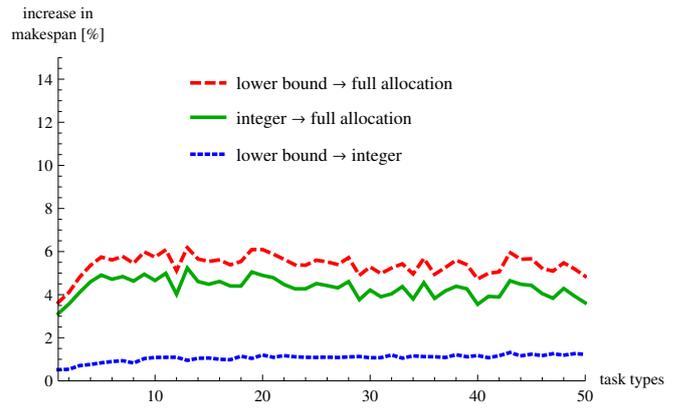


Fig. 3. Relative percent increase in makespan as a function of the number of task types: Quality of the solutions are roughly independent of the number of task types.

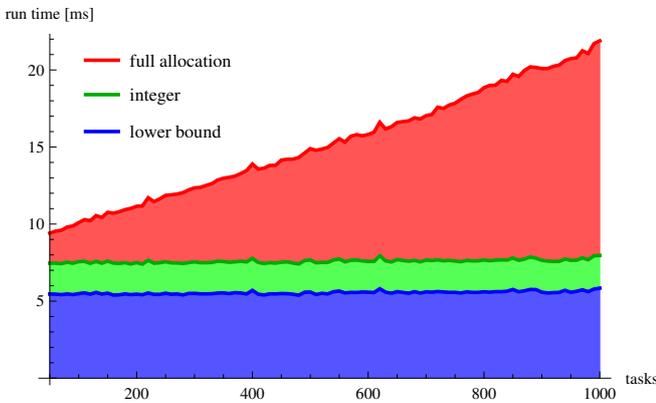


Fig. 2. Algorithm run time versus total number of tasks: Both the lower bound and the rounding algorithms are independent of the number of tasks. The local assignment, used to obtain the full allocation, is linearly dependent on the number of tasks.

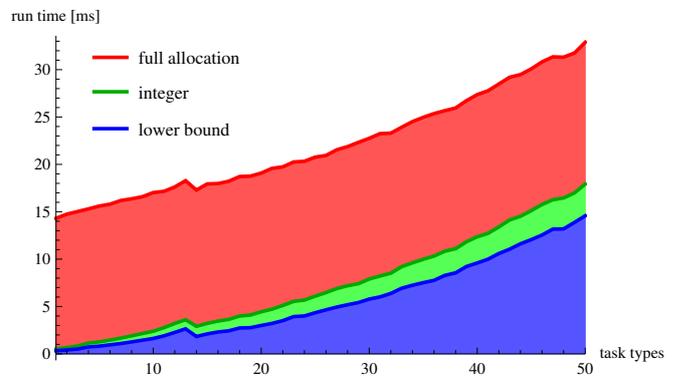


Fig. 4. Algorithm run time as a function of number of task types: The complexity of the lower bound and rounding algorithms grows super linearly with the number of task types.

Fig. 4 shows the run time of the three phases. Here the lower bound has super linear dependence on the number of task types. According to the complexity analysis this should be cubic. The rounding algorithm seems to increase linearly, which corresponds to the analysis. The full allocation phase seems to be independent of the number of task types. This agrees with the analysis because the complexity is not a function of the number of task types T , but instead a function of the number of tasks n_j assigned to a group, regardless of the type of task.

Fig. 5 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 in a machine group to be zero is more difficult due to (1) because some constraint coefficients will be ∞ in the linear programming problem. Practically, an $M_j = 0$ means that the j^{th} column of **ETC** and **APC** should simply be removed and the solution should never assign a task to that group because it has no machines. To avoid this case altogether each machine

group has to have at least one machine (so that there are no degenerate machine groups). Fig. 5 also shows that the quality of the rounding algorithm decreases as the number of machine groups increase. This is expected because there are less tasks to assign to each machine’s group making the approximation weaker. At 36 machine types there is one machine per group. There is only one solution to that scheduling problem (assign all tasks to the one machine), resulting in no increase in makespan in that phase.

Fig. 6 shows the run time as the number of machine types is increased. As expected, the lower bound grows roughly cubically. The rounding algorithm grows roughly linearly also as expected. The time spent scheduling for each group decreases because fewer tasks are scheduled to less machines as the number of machine types increases so it effectively has little dependence on the number of machine types.

Even though the performance of these polynomial time algorithms are desirable, there is some prior work on theoretical bounds that should be noted. In [14] 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

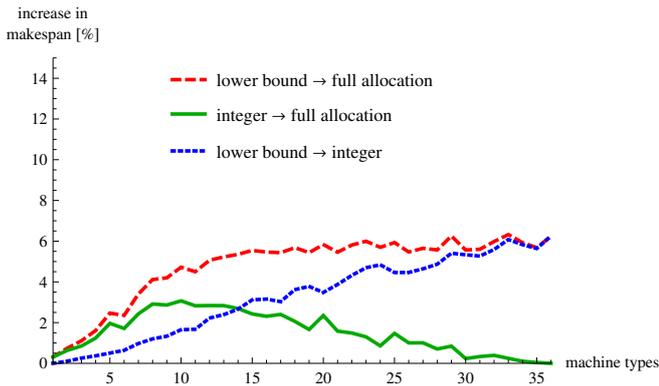


Fig. 5. Relative percent increase in makespan as a function of the *number of machine types*: Overall performance is roughly independent of the number of machine types.

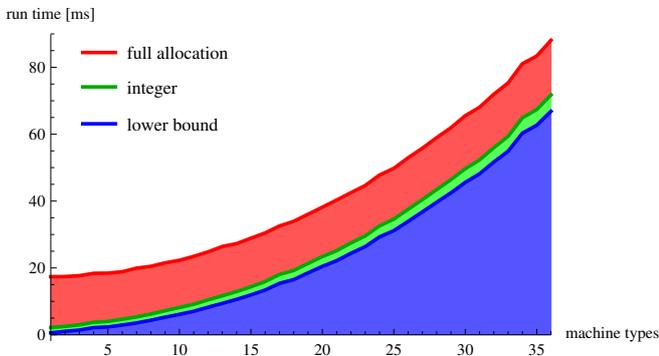


Fig. 6. Algorithm run time versus the *number of machine types*: Lower bound algorithm complexity is super linear in the number of machines types. The rounding and local assignment algorithms are roughly independent.

$P = NP$. Even though Figures 1-6 suggest that one can do better than $3/2$, this is only the case on average. In the next section these algorithms are used to generate the Pareto fronts.

IV. PARETO FRONT GENERATION

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 [15]. Feasible solutions that are dominated are of little interest because one can always find a better solution in all objectives by picking a solution from the non-dominated set. Formally, a feasible solution x dominates a feasible solution y when:

$$\begin{aligned} \forall i \quad f_i(x) &\leq f_i(y) \\ \exists i \quad f_i(x) &< f_i(y) \end{aligned} \quad (5)$$

where $f_i(\cdot)$ is the i^{th} objective function. The non-dominated solutions, also known as *outcomes*, compose the Pareto front.

Finding the Pareto front can be computationally expensive because it means solving variations of the optimization problem numerous times. Most algorithms use scalarization techniques to convert the multi-objective problem into a set of

scalar optimization problems. Major approaches to scalarization include the hybrid method [16], elastic constraint method [16], Benson's algorithm [17] [18], and Pascoletti-Serafini scalarization [19]. Pascoletti-Serafini scalarization is a generalization of many common approaches such as normal boundary intersection, ϵ -constraint, and weighted sum. The focus of our work is on the weighted sum algorithm. The weighted sum algorithm can find all the non-dominated solutions for problems with a convex constraint set and convex objective functions [19]. Weighted sum is used for the linear convex problem in (4) thus all non-dominated solutions can be found. A known issue with the weighted sum algorithm is that it does not uniformly distribute the solutions along the Pareto front. The solutions are often clustered together, but this does not present a problem for our particular use case.

Finding the optimal schedule for makespan alone is NP-Hard in general [4], thus finding the optimal (true) Pareto front is NP-Hard as well. 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 which do not dominate any Pareto optimal solutions.

A. Weighted Sum

The weighted sum algorithm simply forms the positive convex combination of the objectives and sweeps the weights to generate the Pareto front. The first phase is to compute the lower bound solution for energy and makespan independently of each other. Next ΔE_{LB} , which is the difference between the maximum and minimum values of energy, is computed. Likewise, ΔMS_{LB} is computed. The scalarized objective is:

$$\min \frac{\alpha}{\Delta E_{LB}} E_{LB} + \frac{1 - \alpha}{\Delta MS_{LB}} MS_{LB} . \quad (6)$$

A lower bound on the Pareto front can be generated by using several values of $\alpha \in [0, 1]$. Weighted sums will produce duplicate solutions (i.e., x is identical for neighboring values of α). Duplicate solutions are removed to increase the efficiency of the subsequent algorithms. Each solution is rounded by Algorithm 1 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 Algorithm 2 to create the upper bound on the Pareto front.

B. Non-dominated Sorting Genetic Algorithm II

NSGA-II [20] is an adaptation of the Genetic Algorithm (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 [3], 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 is to use the optimal minimum energy solution, sub-optimal minimum makespan solution (from the Min-Min Completion Time [4] algorithm), and a random population as the initial population. This is the original seeding method used in [3]. The second seeding method is to use the full allocations from Algorithm 2 as the initial population for the NSGA-II.

V. PARETO FRONT RESULTS

The system used for these experiments is the same as in Section III, unless stated otherwise. All 1100 tasks, 30 task types, 36 machines, and nine machines types are used as described in [8]; the complete description of the system and output data files from the new algorithm are available in [12]. The hardware used for running the NSGA-II experiments is a 2011 Sager NP7280 with an Intel Core i7 980 @ 3.33 Ghz. The NSGA-II code is implemented in C++.

Fig. 7 shows the Pareto fronts generated from the various algorithms without idle power consumption. The figure shows the actual solutions as markers that are connected by lines. The lower bound, integer, and full allocation are nearly indistinguishable at the lower portion of the plot. This means that the true Pareto front is tightly bounded even though it is unknown. The curve that is dominated by all other curves is the Pareto front generated by the NSGA-II using the first seeding method. In contrast, the lower and upper bounds were found in ~ 10 seconds. The last Pareto front is the NSGA-II seeded with the full allocation. Seeding with the full allocation allows the NSGA-II to both converge to an improved Pareto front as well as decreasing the run time necessary to converge. The NSGA-II attempts to evenly distribute the solutions along the Pareto front and tries to find solutions that are in the convex regions as can be seen in Fig. 7. All the algorithms seem to perform well when minimizing energy alone because computing the optimal minimum energy solution is relatively easy. One simply assigns each task to the machine that requires the lowest energy to execute that task. Solving for the optimal makespan is difficult in practice. Fig. 7 shows that all the algorithms agree in the energy dimension, however in the makespan dimension there are significant distinctions in solution quality. Pareto fronts for other representative systems were also computed with similar results. The new algorithms produced better quality Pareto fronts in significantly less time.

Fig. 8 illustrates how the solutions progress through the three phases of the algorithm when there is no idle power consumption. 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 but the energy may increase or decrease. As x is rounded, machines will finish at different times increasing the makespan. Each blue arrow represents a solution that is being fully allocated. The energy in this case does not change because the local assignment algorithm does not move tasks across

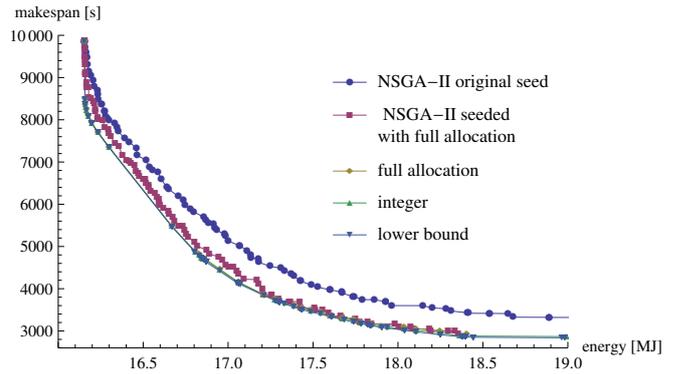


Fig. 7. Pareto front for lower bound, integer, upper bound, and NSGA-II. The lower bound does truly lower bound the other curves. The full allocation or upper bound is very near the lower bound so the optimal Pareto front is tightly bounded. The NSGA-II with the original seed solution quality is rather poor 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 not as good as the full allocation in places.

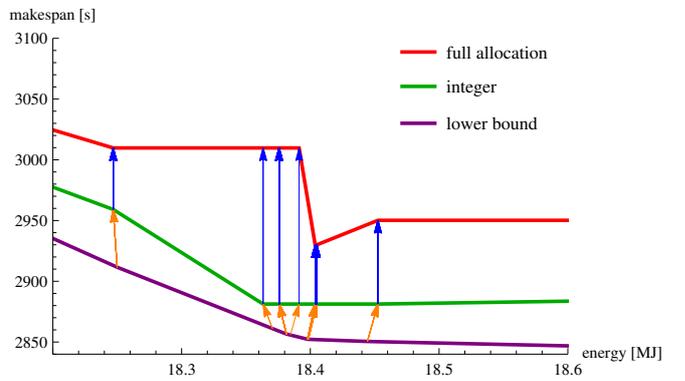


Fig. 8. Progression of solutions from lower bound to integer to upper bound (no idle power)

machine types thus the power consumption cannot change. The makespan increases are highly varying. The full allocation solution second to the right dominates the one on the far right. In this case the solution on the far right is taken out of the estimate of the Pareto front.

Fig. 9 shows the progression of the the solutions with non-zero 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 often have machines idle for some time.

Fig. 10 shows the effect of idle power on the Pareto front. The curves show the lower bound on the optimal Pareto front with different idle powers. The penalty for having a large makespan increases as the idle power increases. The optimal energy solutions now must 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.

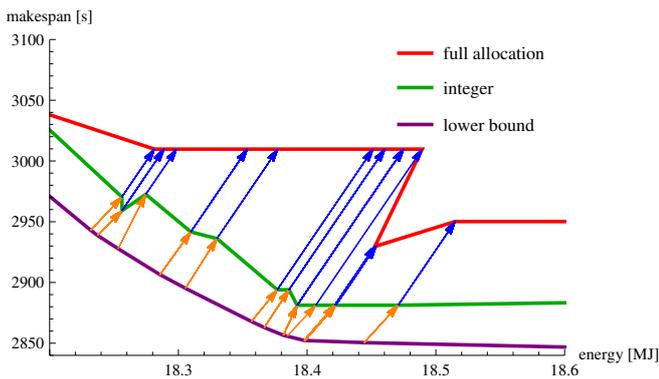


Fig. 9. Progression of solutions from lower bound to integer to upper bound (10% idle power)

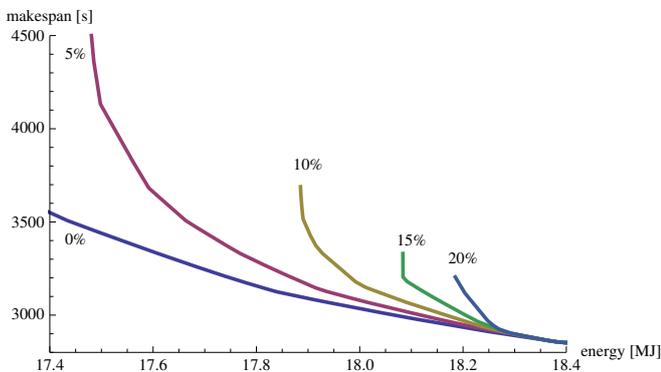


Fig. 10. Pareto front lower bounds when sweeping idle power: Idle power is swept from 5% increments as labelled on the figure. As idle power increases the reward for improving makespan also increases.

VI. 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. The quality of the solution also improves as the size of the problem increases. These two properties make this algorithm perfectly suited for very large scale scheduling problems. Algorithms were also presented that allow one to efficiently recover feasible solutions. These feasible solutions serve as the upper bound on the Pareto front and can be used to seed other algorithms. This upper bound was compared to the solution found with the NSGA-II algorithm and shown to be superior in solution quality and algorithm run time. These algorithms allow the decision makers to more easily trade-off energy and makespan to reduce operating costs and improve efficiency of HPC systems.

ACKNOWLEDGMENT

This work was supported by the Sjoström 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 National Science Foundation. A special thanks to Brian Stefanović, Mark Oxley, and Tim Hansen for their valuable comments.

REFERENCES

- [1] J. Koomey, "Growth in data center electricity use 2005 to 2010," *Analytics Press.*, vol. 1, 2011.
- [2] K. W. Cameron, "Energy oddities, part 2: Why green computing is odd," *Computer*, vol. 46, no. 3, pp. 90–93, 2013.
- [3] 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*, 2012, pp. 81–89.
- [4] 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, no. 6, pp. 810–837, 2001.
- [5] 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.
- [6] A. Al-Qawasmeh, A. Maciejewski, H. Wang, J. Smith, H. Siegel, and J. Potter, "Statistical measures for quantifying task and machine heterogeneities," *The Journal of Supercomputing*, vol. 57, no. 1, pp. 34–50, 2011.
- [7] 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 *IEEE 22nd Heterogeneity in Computing Workshop (HCW)*, 2013.
- [8] 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, accepted.
- [9] D. Bertsimas and J. N. Tsitsiklis, *Introduction to Linear Optimization*, ser. Optimization and Neural Computation. Athena Scientific, 1997.
- [10] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.
- [11] (2013, May) Intel core i7 3770k power consumption, thermal. [Online]. Available: http://openbenchmarking.org/result/1204229-SU-CPUMONITO81#system_table
- [12] (2013, April) Energy and makespan bi-objective optimization data. [Online]. Available: <http://goo.gl/Z2Utv>
- [13] (2013, March) Coin-or clp. [Online]. Available: <https://projects.coin-or.org/Clp>
- [14] J. Lenstra, D. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, no. 1-3, pp. 259–271, 1990.
- [15] V. Pareto, *Cours d'economie Politique*. Lausanne: F. Rouge, 1896.
- [16] M. Ehrgott, *Multicriteria Optimization*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.
- [17] 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.
- [18] A. Löhne, *Vector Optimization with Infimum and Supremum*, ser. Vector Optimization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [19] G. Eichfelder, *Adaptive Scalarization Methods in Multiobjective Optimization*. Springer, 2008.
- [20] 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, no. 2, pp. 182–197, 2002.