Efficient and Scalable Pareto Front Generation for Energy and Makespan in Heterogeneous Computing Systems

Kyle M. Tarplee, Ryan Friese, Anthony A. Maciejewski and Howard Jay Siegel

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 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. 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.

Keywords High performance computing · Scheduling · Bag-of-tasks · Scalable · Efficient · Heterogeneous computing

K.M. Tarplee (✉) · R. Friese · A.A. Maciejewski · H.J. Siegel
Department of Electrical and Computer Engineering,
Colorado State University, Fort Collins, CO 80523, USA
e-mail: kyle.tarplee@colostate.edu; ktarplee@me.com

R. Friese
e-mail: ryan.friese@colostate.edu

A.A. Maciejewski
e-mail: aam@colostate.edu

H.J. Siegel
Department of Computer Science, Colorado State University,
Fort Collins, CO 80523, USA
e-mail: hj@colostate.edu

© Springer International Publishing Switzerland 2015

S. Fidanova (ed.), *Recent Advances in Computational Optimization*,
Studies in Computational Intelligence 580, DOI 10.1007/978-3-319-12631-9_10

1 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 naïve 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 Sect. 2.2. Then algorithms are presented in Sects. 2.3–2.5 that reconstruct a feasible schedule from the lower bound. In Sect. 2.6, the complexity of the algorithm is analyzed. Algorithm scaling quality and runtime results are shown in Sect. 3. Section 4 shows how the lower bounds can be used with any scalarization technique to form a Pareto front. Section 5 compares these algorithms to the NSGA-II algorithm. Section 6 discusses related work and Sect. 7 concludes while presenting some ideas for future work.

2 Approximation Algorithms

2.1 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. Machines that have virtually indistinguishable performance and power properties with respect to the workload are said to be the same *machine type*. Machines within a machine type 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 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 holds well when the number of tasks assigned to each machine type is large. For this approximation, machine types 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 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 Lower Bound

The lower bound is given by the solution to a linear bi-objective optimization (or 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 type j, where x_{ij} is the primary decision variable in the optimization problem. Let $\boldsymbol{ETC}$ be a $T \times M$ matrix where ETC_{ij} is the *estimated time to compute* for a task of type i on a machine of type j. Similarly let $\boldsymbol{APC}$ be a $T \times M$ matrix where APC_{ij} is the *average power consumption* for a task of type i on a machine of type j. These matrices are frequently used in scheduling algorithms [4, 6–8]. $\boldsymbol{ETC}$ and $\boldsymbol{APC}$ are generally determined empirically.

The lower bound of the finishing times of a machine type 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 type j finish at the same time. The finishing time of machine type j, denoted by F_j is given by

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

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 is:

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

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 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. When no task is executing on machine j, the power

consumption is given by the idle power consumption, $APC_{\emptyset j}$. The equation for the lower bound on the energy consumed, incorporating idle power, is given in:

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

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

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

$$\begin{aligned} & \underset{\boldsymbol{x},\, MS_{LB}}{\text{minimize}} \quad \begin{pmatrix} E_{LB} \\ MS_{LB} \end{pmatrix} \\ & \text{subject to: } \forall i \quad \textstyle\sum_j x_{ij} = T_i \\ & \qquad\qquad\quad \forall j \quad F_j \le MS_{LB} \\ & \qquad\qquad\quad \forall i, j \quad x_{ij} \ge 0. \end{aligned} \tag{4}$$

The objective of Eq. (4) is to minimize E_{LB} and MS_{LB}, where $\boldsymbol{x}$ is the primary decision variable. MS_{LB} is an auxiliary decision variable necessary to model the objective function in Eq. (2). The first constraint ensures that all tasks in the bag are assigned to some machine type. 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 or *linear program*. Methods to find a collection of solutions are presented in Sect. 4.

Ideally this vector optimization problem would be solved optimally with $x_{ij} \in \mathbb{Z}_{\ge 0}$. 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 feasible solutions exist. The next few sections describe how to take an infeasible real-valued solution from the linear program and build a complete feasible allocation.

2.3 Recovery Algorithm

An algorithm is necessary to recover a feasible solution or full resource allocation from each infeasible solutions obtained from the lower bound solutions of Eq. (4). 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 recovery algorithm is decomposed into two phases. The first phase rounds the solution while taking care to maintain feasibility of Eq. (4). The second phase assigns tasks to actual machines to build the full task allocation. The next two sections detail the two phases of this recovery algorithm.

2.4 Rounding

Due to the nature of the problem, the optimal solution $\boldsymbol{x}^*$ often has few non-zero elements per row. Usually all the tasks of one type will be assigned to a small number of machine types. In the original problem, tasks are not divisible so one needs to have an integer number of tasks to assign to a machine type. When tasks are split between machine types, 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 $\hat{\boldsymbol{x}}$ 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: $\quad n \leftarrow T_i - \sum_j \lfloor x^*_{ij} \rfloor$
3: $\quad \forall j \quad f_j \leftarrow x^*_{ij} - \lfloor x^*_{ij} \rfloor$
4: $\quad$ Let set K be the indices of the n largest f_j
5: $\quad$ **if** $j \in K$ **then**
6: $\quad\quad \hat{x}_{ij} \leftarrow \lceil x^*_{ij} \rceil$
7: $\quad$ **else**
8: $\quad\quad \hat{x}_{ij} \leftarrow \lfloor x^*_{ij} \rfloor$
9: $\quad$ **end if**
10: **end for**

Algorithm 1 operates on each row of $\boldsymbol{x}^*$ 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 $\hat{\boldsymbol{x}}$ that still assigns all tasks properly and is near to the original solution from the lower

bound. Algorithm 1 minimizes the L_1 norm between the integer solution and the real-valued solution because the L_1 norm is separable and this algorithm chooses the K dimensions to round up that will introduce the least error per dimension.

To illustrate the behavior of the algorithm, let the input be given by Eq. (5). The values in bold indicate assignments that are to be rounded up. The output of the algorithm is given in Eq. (6). The first row does not need to be rounded. The second row rounds up 9.6 because $0.6 \geq 0.4$ and rounds every other component down. The third row shows that the algorithm is not traditional rounding because it rounds up 11.4 due to $0.4 \geq 0.3$. The last row shows how the algorithm might round up two values ($n = 2$).

$$\boldsymbol{x}^* = \begin{pmatrix} 3 & 0 & 9 & 11 & 0 & 0 \\ 3 & 0 & \mathbf{9.6} & 11.4 & 0 & 0 \\ 3 & 15.3 & 9.3 & \mathbf{11.4} & 0 & 0 \\ 3 & 15.2 & \mathbf{9.9} & \mathbf{11.4} & 2.3 & 4.2 \end{pmatrix} \tag{5}$$

$$\hat{\boldsymbol{x}} = \begin{pmatrix} 3 & 0 & 9 & 11 & 0 & 0 \\ 3 & 0 & 10 & 11 & 0 & 0 \\ 3 & 15 & 9 & 12 & 0 & 0 \\ 3 & 15 & 10 & 12 & 2 & 4 \end{pmatrix} \tag{6}$$

The makespan computed from the integer solutions produced by Algorithm 1 may still not be realizable, even though an integer number of tasks is assigned to machine types. To obtain the makespan of the integer solution, computed by Eq. (2), 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. The local assignment algorithm, discussed in the next subsection, will remedy this by forcing each task to be wholly assigned to a single machine.

2.5 Local Assignment

The last phase in recovering a feasible assignment solution is to schedule 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 case because the execution and energy characteristics of 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 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 $\hat{x}_{ij}$ 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 the earliest ready time machine of type j
9: Update ready time
10: **end for**
11: **end for**

Each column of $\hat{\boldsymbol{x}}$ is processed independently. List z contains $\hat{x}_{ij}$ 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 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 diminishes as the problem sizes become large.

2.6 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 $M \max_j \left(n_j \log n_j + n_j \log M_j\right)$.

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 inde-

pendent 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 a small number of task types. 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.

3 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 Sect. 2.6. $\boldsymbol{ETC}$ and $\boldsymbol{APC}$ are needed to test the algorithms. A set of five benchmarks executed over nine machine types where used to construct the initial matrices [11]. Then the method found in [7] was used to construct larger $\boldsymbol{ETC}$ and $\boldsymbol{APC}$ matrices. Nominally there are 1,100 tasks made up of 30 task types. The number of tasks per task type varies from 11 to 75. There are nine machine types with four machines of each type for a total of 36 machines. 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 intractable to solve for the optimal makespan. It is even too expensive to solve the linear programming relaxation of the assignment of individual tasks to individual machines for this system. This highlights the need for much more scalable solutions. Even though the optimal solution is not known it is still possible to 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 where 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.

Figure 1a 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 type system. 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

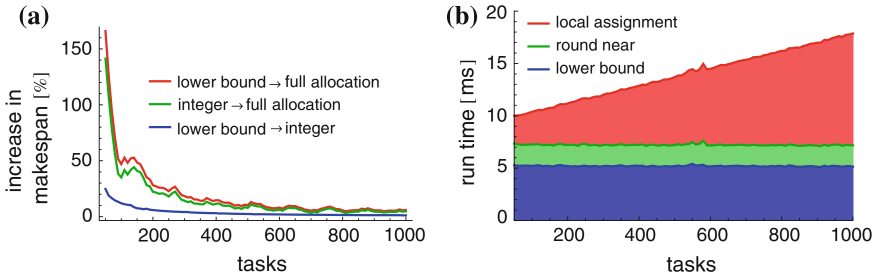

Fig. 1 Sweeping the *total number of tasks*: **a** shows the relative percent increase in makespan. The quality of the solution improves as more tasks are used. **b** shows the algorithms' run time. Both the lower bound and the rounding algorithms are independent of the number of tasks. The local assignment, used to obtain the full allocation, is linearly dependent on the number of tasks

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. 1a 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. 1b 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.

The relative increase in makespan is shown in Fig. 2a. The figure shows the same three curves as Fig. 1a, however this time varying the total number of machines. The number of tasks, machine types, and task types are held constant. As the number of machines increases the increase in makespan due to the full allocation step increases 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 2b 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 full allocation step is roughly linear in the number of machines. This corresponds to the analysis in Sect. 2.6.

Figure 3a shows the same three curves as Fig. 1a, however this time varying the number of task types. The number of tasks, machines, and machine types are held constant for this experiment. Figure 3a shows that again the local assignment algorithm is causing most of the degradation in makespan. The relative error in makespan

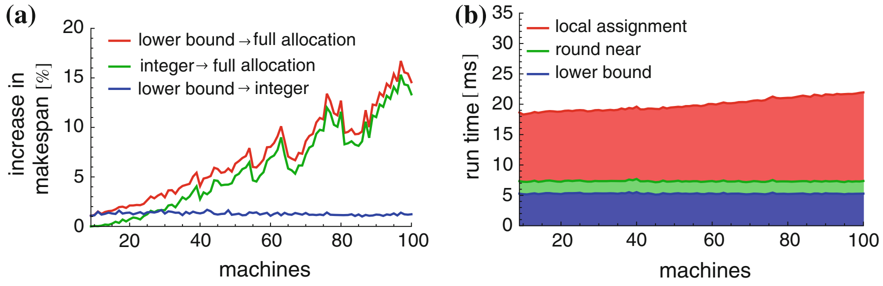

Fig. 2 Sweep of the *total number of machines*: **a** shows the relative percent increase in makespan. The quality of the solution decreases as more machines are used. **b** shows the algorithms' run time. Both the lower bound and the rounding algorithms are independent of the number of machines. The local assignment, used to obtain the full allocation, is linearly dependent on the number of machines

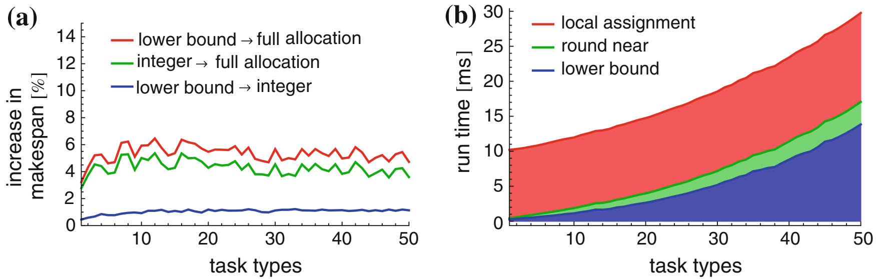

Fig. 3 Sweeping *number of task types*: **a** shows the relative percent increase in makespan. Quality of the solutions are roughly independent of the number of task types. **b** shows the algorithm's run time. The complexity of the lower bound and rounding algorithms grows super linearly with the number of task types. The local assignment algorithm run time is independent of the number of task types

does not tend to zero because increasing the number of task types does not improve the quality of the approximation.

Figure 3b 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 machine type, regardless of the type of task.

Figure 4a 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 type to be zero is troublesome due to Eq. (1) because some constraint coefficients will be ∞ in the linear programming problem. Practically, an $M_j = 0$ means that the jth column of $\boldsymbol{ETC}$ and $\boldsymbol{APC}$ should simply be removed

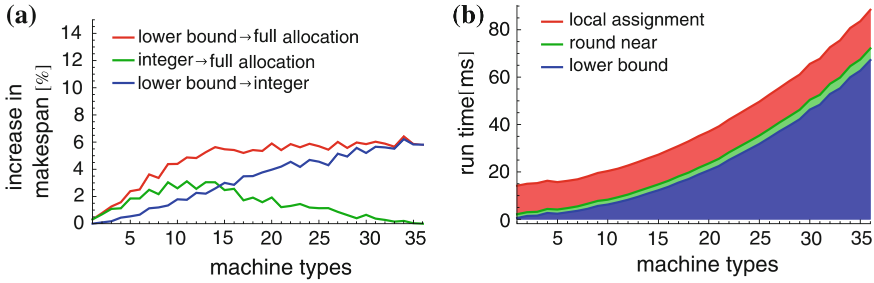

Fig. 4 Sweeping the *number of machine types*: **a** shows the relative percent increase in makespan. Overall performance is roughly independent of the number of machine types. **b** shows the algorithm's run time. The lower bound algorithm's complexity is super linear in the number of machines types. The rounding and local assignment algorithms are roughly independent of the number of machine types

and the solution should never assign a task to that machine type because it has no machines. To avoid this case altogether each machine type is forced to have at least one machine (so that there are no degenerate machine types). Figure 4a 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. At 36 machine types there is exactly one machine per machine type. 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.

Figure 4b shows the run time as the number of machine types is increased. As expected, the lower bound grows roughly cubicly. The rounding algorithm grows roughly linearly also as expected. The time spent performing local assignment for each machine type 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 $P = NP$. Even though Figs. 1–4 suggest that one can do better then 3/2, this is only the case on average. In the next section these algorithms are used to generate Pareto fronts.

4 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 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. When all objectives should be minimized, a feasible solution x dominates a feasible solution y when:

$$\begin{aligned} \forall i \quad & f_i(x) \leq f_i(y) \\ \exists i \quad & f_i(x) < f_i(y) \end{aligned} \tag{7}$$

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

Finding the Pareto front can be computationally expensive because it involves solving numerous variations of the optimization problem. 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 [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. 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 [19]. Weighted sum is used for the linear convex problem in Eq. (4) 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 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 [20], thus finding the optimal (true) Pareto front is NP-Hard as well. 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 and the upper bound curve.

4.1 Weighted Sum

The weighted sum algorithm simply 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. Next let ΔE_{LB} be the difference in energy between these two lower bound solutions. Likewise, let ΔMS_{LB} be the difference in makespan between these two lower bound solutions. The scalarized objective is:

$$\min \frac{\alpha}{\Delta E_{LB}} E_{LB} + \frac{1-\alpha}{\Delta MS_{LB}} MS_{LB}. \quad (8)$$

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., $\boldsymbol{x}$ 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.

4.2 Non-dominated Sorting Genetic Algorithm II

NSGA-II [21] 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.

5 Pareto Front Results

The system used for these experiments is the same as in Sect. 3, unless stated otherwise. All 1,100 tasks, 30 task types, 36 machines, and nine machines types are used and are 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 2013 Dell XPS'15 with an Intel i7-4702HQ 2.2 GHz CPU. The NSGA-II code is implemented in C++.

Figure 5 shows Pareto fronts for four different systems. Pareto fronts for each of the algorithms are shown for each system. All the systems have zero idle power consumption.

Figure 5a shows the Pareto fronts generated from the lower bound, round near, local assignment, and NSGA-II algorithms. The figure shows the actual solutions as markers that are connected by lines. The legend shows the total time duration

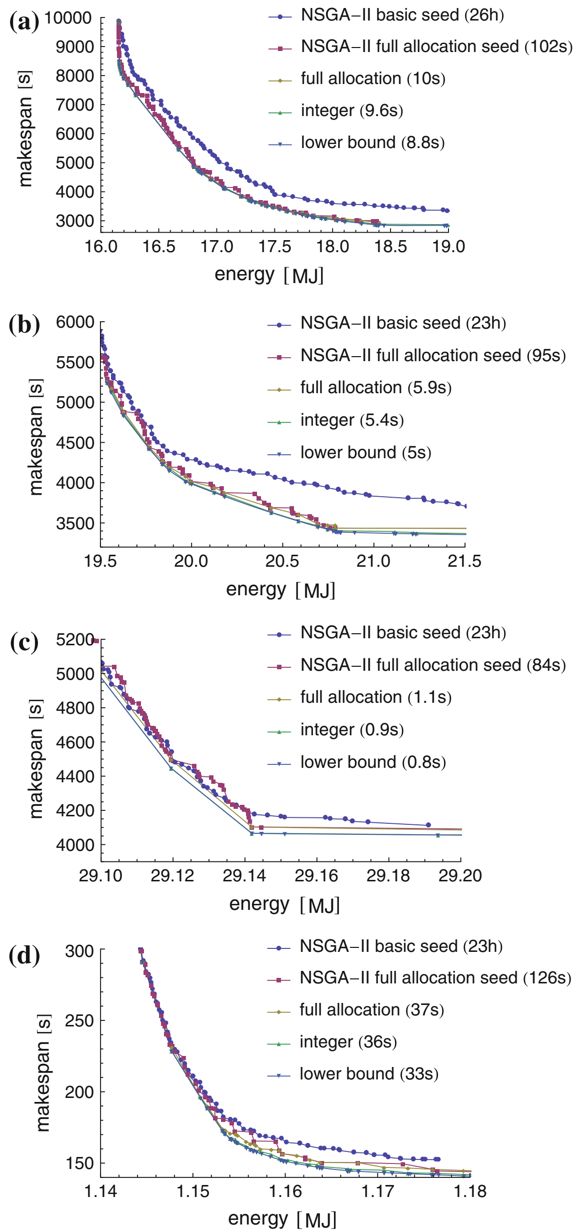

Fig. 5 Pareto front for lower bound, integer, upper bound, and NSGA-II: the lower bound does truly lower bound the other curves. The full allocation or upper bound is very near the lower bound so the optimal Pareto front is tightly bounded. The times shown in the parenthesis in the legend indicate the total time to compute the solution. 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 **a** nine machine type system, **b** six machine type system, **c** two machine type system, **d** synthetic ten machine type system

for computing a given Pareto front. For instance having already computed the lower bound the additional time necessary to compute the full allocation is 1.2 s. The lower bound, integer, and 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 by all other curves is the Pareto front generated by the NSGA-II using the first seeding method. The NSGA-II took hours to find that sub-optimal Pareto front. In contrast, the lower and upper bounds were found in approximately 10 s. 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 decrease the run time necessary to converge. The NSGA-II attempts to evenly distribute the solutions along the Pareto front as can be seen in Fig. 5a. 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. Figure 5a shows that all the algorithms agree in the energy dimension, however in the makespan dimension there are significant distinctions in solution quality. The new algorithms produce better quality Pareto fronts in significantly less time.

A few different systems are used to further illustrate the applicability of the LP-based Pareto front generation technique. Figure 5b shows a system composed of the first six machine types from the previous system, with six machines per type. Figure 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 can still outperform the NSGA-II algorithm even when the number of machines types become small.

The results in Fig. 5d are based on an entirely different system that was previously used in [3]. The system has 50 machines composed from ten machine types. There are 1,000 tasks made from 50 task types. The ETC and APC matrices were generated randomly with the Coefficient of Variation (COV) method described in [22]. Even though this system is very different from the previous systems, the LP-based algorithm produces a superior Pareto front in significantly less time.

Figure 6a 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 $\boldsymbol{x}$ is rounded, machines will finish at different times 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. 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.

Figure 6b 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 may have machines idle for some amount time.

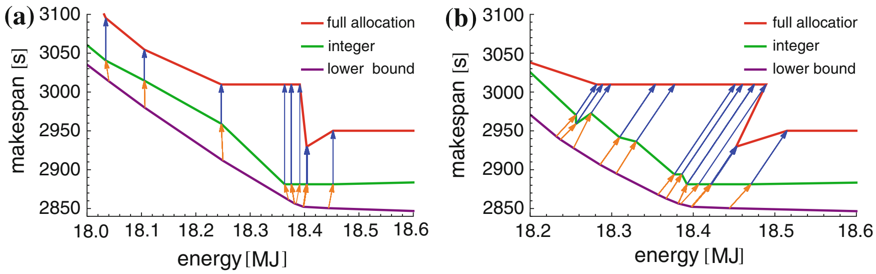

Fig. 6 Progression of solutions from lower bound to integer to upper bound **a** no idle power **b** 10 % idle power

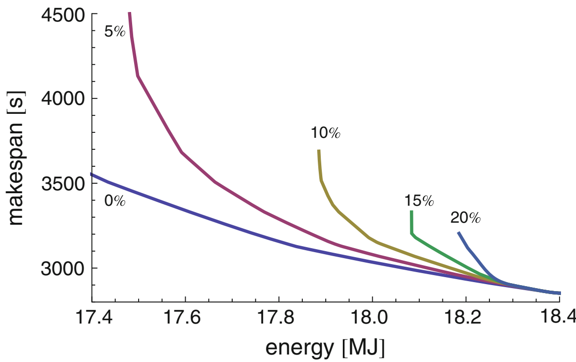

Fig. 7 Pareto front lower bounds when sweeping idle power: Idle power is swept from 5 % increments as labeled on the figure. As idle power increases the reward for improving makespan also increases

Figure 7 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.

6 Related Work

Techniques for generating Pareto fronts have been well studied [3, 7, 8, 15, 21]. This work achieves huge gains in performance over prior algorithms by exploiting properties specific to this static scheduling problem.

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 how to find high quality solutions

on average, whereas [20, 23] 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 [20, 23].

While this paper deals with scheduling tasks to entire machines, the algorithms in this paper 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 [24] that deal with scheduling on a single machine by using dynamic voltage and frequency scaling (DVFS).

7 Conclusion

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.

This work could be extended by considering alternative scalarization techniques to hopefully 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.

Acknowledgments 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 National Science Foundation. A preliminary version of portions of this work have been previously presented in [25].

References

1. Koomey, J.: Growth in data center electricity use 2005 to 2010, pp. 1. Analytics Press (2011)
2. Cameron, K.W.: Energy oddities, part 2: why green computing is odd. Computer **46**(3), 90–93 (2013)
3. Friese, R., Brinks, T., Oliver, C., Siegel, H.J., Maciejewski, A.A.: 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. 81–89 (2012)
4. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **61**(6), 810–837 (2001)
5. Bharadwaj, V., Robertazzi, T.G., Ghose, D.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press, Los Alamitos (1996)
6. Al-Qawasmeh, A.M., Maciejewski, A.A., Wang, H., Smith, J., Siegel, H.J., Potter, J.: Statistical measures for quantifying task and machine heterogeneities. J. Supercomput. **57**(1), 34–50 (2011)
7. Friese, R., Khemka, B., Maciejewski, A.A., Siegel, H.J., Koenig, G.A., Powers, S., Hilton, M., Rambharos, J., Okonski, G., Poole, S.W.: An analysis framework for investigating the trade-offs between system performance and energy consumption in a heterogeneous computing environment. In: IEEE 27th International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Heterogeneity in Computing Workshop, IEEE, pp. 19–30 (2013)
8. Friese, R., Brinks, T., Oliver, C., Siegel, H.J., Maciejewski, A.A., Pasricha, S.: 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)
9. Bertsimas, D., Tsitsiklis, J.N.: Introduction to Linear Optimization. Optimization and Neural Computation. Athena Scientific (1997)
10. Graham, R.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**(2), 416–429 (1969)
11. Phoronix Media: Intel core i7 3770k power consumption, thermal. http://openbenchmarking.org/result/1204229-SU-CPUMONITO81 (May 2013)
12. Tarplee, K.M.: Energy and makespan bi-objective optimization data. http://goo.gl/3Ik8eC (November 2013)
13. Hall, J.: Coin-or clp. https://projects.coin-or.org/Clp (March 2013)
14. Lenstra, J., Shmoys, D., Tardos, É.: Approximation algorithms for scheduling unrelated parallel machines. Math. Program. **46**(1–3), 259–271 (1990)
15. Pareto, V.: Cours d'economie Politique. F. Rouge, Lausanne (1896)
16. Ehrgott, M.: Multicriteria Optimization. Springer-Verlag New York Inc, Secaucus (2005)
17. Benson, H.: An outer approximation algorithm for generating all efficient extreme points in the outcome set of a multiple objective linear programming problem. J. Global Optim. **13**(1), 1–24 (1998)
18. Löhne, A.: Vector Optimization with Infimum and Supremum. Vector Optimization. Springer, Berlin Heidelberg (2011)
19. Eichfelder, G.: Adaptive Scalarization Methods in Multiobjective Optimization. Springer (2008)
20. Jansen, K., Porkolab, L.: Improved approximation schemes for scheduling unrelated parallel machines. Math. Oper. Res. **26**(2), 324–338 (2001)
21. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
22. Ali, S., Siegel, H.J., Maheswaran, M., Hensgen, D., Ali, S.: Representing task and machine heterogeneities for heterogeneous computing systems. Tamkang J. Sci. Eng. **3**(3), 195–208 (2000)

23. Shmoys, D.B., Tardos, É.: Scheduling unrelated machines with costs. In: Fourth Annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics. 448–454 (1993)
24. Li, D., Wu, J.: Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms. In: 41st International Conference on Parallel Processing (ICPP), pp. 430–439 (2012)
25. Tarplee, K.M., Friese, R., Maciejewski, A.A., Siegel, H.J.: 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. 401–408 (2013)