J. Parallel Distrib. Comput. 84 (2015) 76–86

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Scalable linear programming based resource allocation for makespan minimization in heterogeneous computing systems

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

[a] Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, United States
[b] Department of Computer Science, Colorado State University, Fort Collins, CO 80523, United States

HIGHLIGHTS

- We present a novel scheduling algorithm for heterogeneous computing environments.
- Uses groupings of similar tasks and machines to reduce the computational complexity.
- Computes upper and lower bounds on the optimal makespan.
- Schedule approaches a lower bound on the makespan as the number of tasks increases.
- Scheduling algorithm run time scales linearly with the number of tasks.

ARTICLE INFO

Article history:
Received 12 September 2014
Received in revised form
15 March 2015
Accepted 7 July 2015
Available online 20 July 2015

Keywords:
High performance computing
Scheduling
Resource management
Bag-of-tasks
Heterogeneous computing
Linear programming

ABSTRACT

Resource management for large-scale high performance computing systems poses difficult challenges to system administrators. The extreme scale of these modern systems require task scheduling algorithms that are capable of handling at least millions of tasks and thousands of machines. Highly scalable algorithms are necessary to efficiently schedule tasks to maintain the highest level of performance from the system. In this study, we design a novel linear programming based resource allocation algorithm for heterogeneous computing systems to efficiently compute high quality solutions for minimizing makespan. The novel algorithm tightly bounds the optimal makespan from below with an infeasible schedule and from above with a fully feasible schedule. The new algorithms are highly scalable in terms of solution quality and computation time as the problem size increases because they leverage similarity in tasks and machines. This novel algorithm is compared to existing algorithms via simulation on a few example systems.

1. Introduction

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

Our work considers a common scheduling model where users submit a set of independent tasks known as a *bag-of-tasks* [7]. We assume that the full bag-of-tasks is known a priori [7] (i.e., *static scheduling*), a task can be scheduled to execute on only one machine, and machines may only process one task at a time. The HPC environments of primary interest have highly heterogeneous tasks and machines and are known as heterogeneous computing (HC) systems [17].

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

* Corresponding author.
E-mail addresses: kyle.tarplee@colostate.edu (K.M. Tarplee), ryan.friese@colostate.edu (R. Friese), aam@colostate.edu (A.A. Maciejewski), hj@colostate.edu (H.J. Siegel).

http://dx.doi.org/10.1016/j.jpdc.2015.07.002
0743-7315/

a naïve scheduling algorithm. The makespan is a very common offline scheduling objective [14,27]. The algorithms in this work can be adapted to online batch mode scheduling algorithms where the makespan is minimized for each batch of tasks. When a new task arrives or a task is removed from the batch because it is now running on a machine, the schedule for the batch of tasks can be recomputed.

Finding the optimal schedule for this static scheduling problem is NP-Hard in general [13]. Therefore we seek to design algorithms that find near-optimal solutions relatively quickly.

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

In summary the contributions of this paper are:

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

The rest of this paper is organized as follows. First an algorithm for minimum makespan scheduling is presented in Section 2. Section 3 describes the nominal HC system and workload used for simulations and evaluation. Bounds on the solution quality are provided by the algorithm and are discussed in Section 4. In Section 5, we compare this algorithm to other heuristic algorithms. The applicability of the algorithm to very large-scale problems is shown in Section 6 along with simulation results for very large system configurations. We discuss related work in Section 7, and Section 8 concludes this study and presents some ideas for future work.

2. Algorithm design

2.1. Approach

The fundamental approach of this paper is to apply divisible load theory (DLT) [5,4] to ease the computational requirements of calculating a solution to the makespan scheduling problem. The technique operates in two steps to calculate the lower and upper bounds on makespan. The first step uses DLT, where we assume a single task is allowed to be divided and scheduled onto any number of machines, to calculate the lower-bound solution. After the lower-bound solution is computed, a two-phase algorithm is used to recover a feasible solution from the infeasible lower-bound solution. The feasible solution will be shown empirically to be a tight upper bound on the optimal makespan.

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

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

2.2. Lower bound

The lower bound on the makespan is given by the solution to an LP problem and is formulated as follows. Let there be T task types and M machine types. Let T_i be the number of tasks of type i and M_j be the number of machines of type j. Let μ_{ij} be the number of tasks of type i assigned to machine type j, where $\mu_{ij} \in \mathbb{R}$ is the primary decision variable in the optimization problem. Let **ETC** be a $T \times M$ matrix where ETC_{ij} is the *estimated time to compute* a task of type i on a machine of type j. The **ETC** matrix is frequently used in scheduling algorithms (e.g., [10,15,7,8,16]). **ETC** is generally obtained from historical data in real environments.

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

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

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

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

The resulting optimization problem for the lower bound is:

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

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

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

2.3. Recovery algorithm

2.3.1. Overview

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

2.3.2. Rounding

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

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

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

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

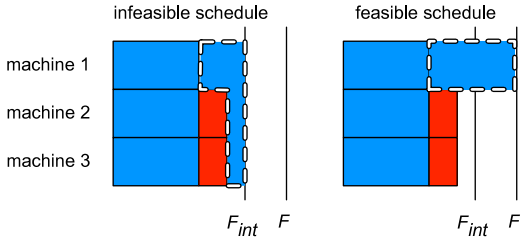


Fig. 1. For any given machine type, even though there are an integer number of tasks of each type (blue and red task types) the lower-bound finishing time of the integer solution, F_{int}, may not be equal to the true finishing time, because the last blue (dashed outline) task on machine 1 would be divided. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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

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

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

The makespan computed from the integer solutions produced by Algorithm 1 may still not be realizable, even though an integer number of tasks are assigned to each machine type. To obtain the makespan of the integer solution, computed similarly to (2) as $\max_j \frac{1}{M_j} \sum_i \hat{\mu}_{ij} ETC_{ij}$, one might still be forced to split tasks among machines of a given machine type to force the finishing times of all the machines to be the same. Having a schedule with a fraction of a task assigned to a machine is not a feasible allocation. Fig. 1 shows an example where four blue tasks and two red tasks are assigned to three machines of the same machine type. Even with an integer number of tasks assigned to the machines, the makespan is still larger than the lower-bound on the finishing time of the integer solution, F_{int}, shown in the figure, because the last blue task (dashed outline) would be divided. In the next subsection we explain our local assignment algorithm that will remedy this by forcing each task to be wholly assigned to a single machine.

2.3.3. Local assignment

The last phase in recovering a feasible assignment solution schedules the tasks, already assigned to each machine type, to specific machines within that group of machines. This scheduling problem is much easier than the general, heterogeneous, case

because the execution characteristics of all machines in a group are the same. This problem is formally known as the multiprocessor scheduling problem [11]. One must schedule a set of heterogeneous tasks onto a set of identical machines. The longest processing time (LPT) algorithm is commonly used for solving the multiprocessor scheduling problem [11]. 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

```
for j = 1 to M do
    Let z be an empty list
    for i = 1 to T do
        z ← join(z, (task type i replicated μ̂_ij times))
    end for
    y ← sort descending by ETC(z)
    for k = 1 to ||y|| do
        assign task y_k to the earliest ready time machine of type j
        update ready time
    end for
end for
```

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

Algorithms exist that will produce better solutions, but it will be shown that the effect of the sub-optimality of this algorithm on the overall performance diminishes as the problem size becomes large. The makespan of this feasible solution is an upper bound on the optimal makespan. The quality of these solutions is evaluated in Section 4.

3. Simulation setup

An **ETC** matrix is needed to evaluate the algorithms. To generate this matrix a set of five benchmarks executed over nine machine types were used to construct the initial matrices [21]. Then the method found in [9] was used to construct a larger **ETC** matrix. Nominally there are 1100 tasks composed of 30 task types. The number of tasks per task type varies from 11 to 75 and was generated by the method used in [9]. There are nine machine types with four machines of each type for a total of 36 machines. This environment will be referred to as the nine machine type environment. For a complete description of this environment and the raw result data see the supplementary material (Appendix A). This environment was chosen to highlight key aspects of the algorithms. The simulations where executed for 200 Monte Carlo trials unless otherwise noted. In Section 6 the size of the environment will be scaled up considerably to show the efficiency of the proposed algorithm.

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

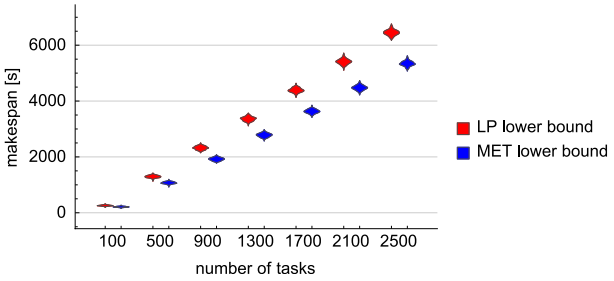


Fig. 2. Distributions of lower bounds from the LP-makespan and MET algorithms: The shape of the glyphs in this figure show the probability density of different y-axis values for a given x-axis value. The broader the shape, the higher the probability at that y-axis value. The LP-makespan lower bound is much tighter than the MET-based lower bound.

4. Minimum makespan quality bounds

4.1. Introduction

In this section, we empirically evaluate the tightness of the bounds computed by the minimum makespan scheduling algorithm described in Section 2 (henceforth referred to as LP-makespan). The lower bound is compared to an alternative lower bound based on minimum execution time (MET) for each task. The lower and upper bounds are compared to each other to show how small the margin for improvement is in the solution quality of LP-makespan. Lastly, we compare the run times of the three phases of the algorithm. The nine machine type environment is used for this set of simulations. These simulations vary the number of tasks to show the scaling trends. The bag-of-tasks is generated by sampling with replacement from the original task type distribution. All other parameters remain unchanged.

4.2. MET lower bound comparison

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

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

Fig. 2 shows the MET-based lower bound alongside the LP-makespan lower bound. The width of the glyphs represent the normalized sample probability density of the makespan. In statistics, these are referred to as relative frequency distributions [28]. The wider the glyph the more probability density that exists at that value for makespan. The glyphs are offset in the x-axis; however, they correspond to the same number of tasks for each lower bound shown. The LP-makespan lower bound is much tighter (i.e., larger makespan).

4.3. Upper and lower bound tightness

LP-makespan produces upper and lower bounds that can be used to determine how much improvement in makespan is theoretically possible. The feasible schedule's makespan cannot be

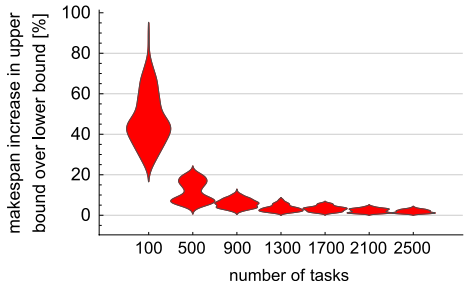


Fig. 3. Distribution of the percent change in the LP-makespan upper bound relative to the LP-makespan lower bound: The room for improvement in the LP-makespan algorithm is less than a few percent as the number of tasks grows large for this particular environment.

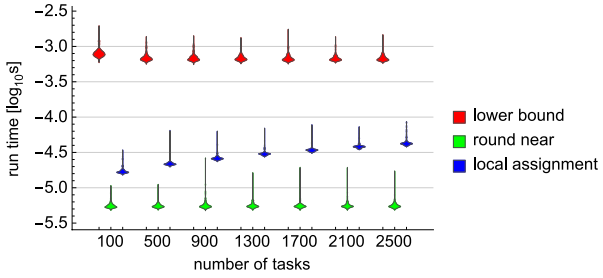


Fig. 4. Distributions of the logarithm of the run time for the three phases of the LP-based algorithm when varying the number of tasks: Lower-bound algorithm and rounding algorithm are not strongly dependent on the number of tasks. The local assignment algorithm run time is linear in the number of tasks. The lower-bound algorithm dominates the run times for the size of problems considered and takes a few milliseconds to complete.

smaller than the LP-based lower bound. This lower bound is only achievable when the optimal schedule has no machine idle for any length of time. Fig. 3 shows the probability distributions of the percent increase in the upper bound's makespan compared to the lower bound as the number of tasks to be executed increases. The gap between the upper and lower bound decreases as the number of tasks increase because the lower bound becomes tighter as the constraint of task indivisibility has less of an effect. The variance in the gap also decreases as the number of tasks increase. On average, only a 1.8% improvement might be possible in the LP-makespan algorithm at 2500 tasks. It is hard to determine where the optimal makespan lies within the lower and upper bounds because it is extremely computationally expensive to compute.

4.4. Run times for the algorithm phases

Fig. 4 shows the probability distributions of the run times of the three phases of the LP-makespan algorithm. The number of tasks is varied to show the dependence on that parameter. The plot is logarithmic in the time axis because the run times of the rounding and local assignment are much shorter than the time required to find the lower bound for these small problem sizes. Only the local assignment has a strong dependence on the number of tasks to be scheduled. The run time of all the phases of the algorithm is reasonably small, taking only a few milliseconds to complete for this HPC environment.

5. Heuristic algorithms comparison

5.1. Overview

It has been shown that the min–min and max–min algorithms are effective heuristics for minimizing makespan within a reasonable amount of computation time for heterogeneous computing systems [7,20]. The solution quality, run time, and scalability of both these heuristic algorithms and the LP-makespan algorithm will be analyzed in this section.

5.2. Classical algorithm

The min–min and max–min algorithms are described in [7]. The max–min algorithm, to be described later, is a variant of the min–min algorithm. In the classical min–min algorithm, there is no assumption that one has groups of task types and machine types [12]. Algorithm 3 is the min–min algorithm designed without any regard to task and machine groups. Let t_{type} and m_{type} be the types of task t and machine m, respectively. Let the ready time of machine m be given by rt_m. The min–min algorithm iteratively assigns the task with the minimum completion time to that task's minimum completion time machine.

Algorithm 3 Classic min–min algorithm

U = set of all tasks from all task types
$\forall m \quad rt_m = 0$
while $U \neq \emptyset$ **do**
  for t in U **do**
    $mct_t \leftarrow \min_m \left(rt_m + ETC_{t_{\text{type}}\, m_{\text{type}}}\right)$
    $m_t \leftarrow \arg\min_m \left(rt_m + ETC_{t_{\text{type}}\, m_{\text{type}}}\right)$
  end for
  $t^* \leftarrow \arg\min_t mct_t$
  $m^* \leftarrow m_{t^*}$
  assign task t^* to machine m^*
  $rt_{m^*} \leftarrow mct_{t^*}$
  $U \leftarrow U \setminus t^*$
end while

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

5.3. Optimized algorithm

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

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

5. Parameters and return values are counts of tasks instead of lists of tasks or task types.

Computing the new best minimum MCT task and machine pair at each iteration of the outer loop of the algorithm is a minor optimization. Each task of the same type has the same execution time properties, thus, when computing a task's best match the algorithm only needs to consider each task type and not each individual task. This computation to find the best candidate match for each task type need not be recomputed if that task type's MCT machine was not assigned the task on the last iteration. Thus, the optimized algorithm stores each task type's best match and removes the match if that machine was assigned in the last iteration. The algorithm also stores the task type list in such a way that the task type entries that have no more tasks to be assigned can be quickly and safely removed from the list to reduce overhead of subsequent iterations. An important improvement in the algorithm was to remove a large amount of dynamic memory allocation in terms of both number of allocations and size of the allocations, for the function parameters and returned schedule. The parameter that described the bag-of-tasks could easily be implemented as a list of tasks to be assigned. This has the downside of requiring a huge amount of storage when scheduling a large number of tasks. Instead an array of length T that contains the number of tasks of each type is used to describe the bag-of-tasks. The mapping of a particular task to a particular machine is irrelevant when that task has the same run time characteristics as all other tasks of the same type. All that is relevant is the number of tasks of type i that are assigned to a machine. As such, no more information than necessary is computed, which further improves the performance. The resultant task assignments are also stored in a single dense ragged (i.e., irregular) [6] array of integers where the first dimension is of size T, the second dimension is of size M, and the last is of size M_j. The entries of this array, denoted y_{ijk}, are the number of tasks of type i assigned to machine type j, machine k. Algorithm 4 incorporates all of these improvements into the min–min algorithm.

Algorithm 4 Optimized min–min algorithm

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

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

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

Algorithm 4 is significantly faster than Algorithm 3, especially as the number of tasks becomes large. The complexity of Algorithm 3 is quadratic in the total number of tasks because both the inner and outer loops effectively iterate over all tasks. Algorithm 4 is only linear in the total number of tasks because the inner loop only iterates over task types. The source code for the optimized algorithm is included in the supplementary material (see Appendix A).

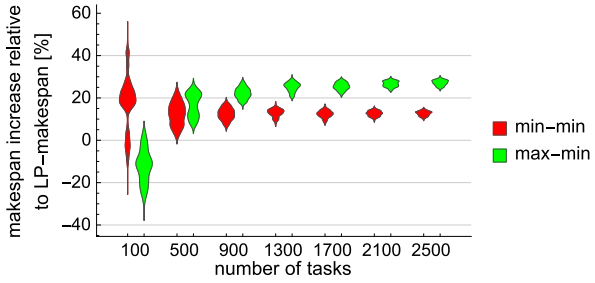


Fig. 5. Distributions of the makespan from min–min and max–min relative to LP-makespan as the number of tasks varies: The LP-makespan algorithm produces better schedules for sufficiently large number of tasks.

5.4. Results

Fig. 5 shows the makespan of the min–min and max–min compared to the makespan of the LP-makespan algorithm for the nine machine type environment. For all but small numbers of tasks the LP-makespan algorithm produces a shorter makespan. For large numbers of tasks, the min–min algorithm produces on average a 13% longer makespan than LP-makespan for this particular HPC environment. Max–min performed even worse as the number of tasks become large, producing schedules that are on average 26% longer than LP-makespan. The LP-makespan algorithm outperforms both heuristics for large problem sizes because it solves a global optimization problem for the relaxation allowing it to make very complex decisions about the allocation to directly minimize makespan. The heuristics only indirectly minimize makespan. When the problem size is small, the task divisibility modeling assumption breaks down leading to poor performance from the LP-makespan algorithm. The variance of the relative makespan distribution is very large for small numbers of tasks, however, the variance decreases rapidly as the number of tasks become larger.

The run time comparison between min–min using Algorithm 4 and LP-makespan is shown in Fig. 6 for the nine machine type environment. Min–min is linearly dependent on the number of

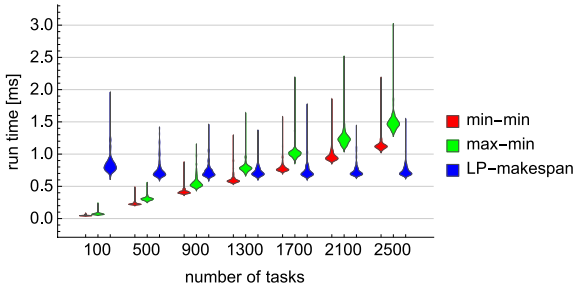


Fig. 6. Distributions of the algorithm run time for min–min, max–min, and LP-makespan as the number of tasks is increased: The LP-makespan algorithm is faster for large numbers of tasks.

tasks, while the LP-makespan algorithm has a fixed run time cost to solve the LP problem but nearly no increase thereafter. LP-makespan is slightly slower than the heuristic algorithms for less than 1300 tasks, but faster for larger numbers of tasks. The max–min algorithm differs from the min–min algorithm in the orientation of a single inequality operator yet its run time is measurably worse. The difference lies in the effectiveness of storing the MCT machine for each task. This storage is invalidated when the machine is this task's MCT machine that was assigned a task in the previous iteration. For min–min 70% were valid whereas for max–min only 60% of the reads were valid. This means that the expensive operation of computing the MCT machine for a task type (iterating over all machines of all types) occurs more often for max–min then it does for min–min for this particular environment. When the MCT machine storage is disabled (i.e., the MCT machine is found every iteration), the algorithms have identical run times. There are some environments where max–min will have a higher percentage of valid reads from the MCT machine storage, so this property is not intrinsic to the algorithms but rather a property of the environment.

A set of randomly generated simulation environments are used to compare the min–min and max–min algorithms with the LP-makespan algorithm. There are 15 task types and ten machine types in these systems. One million tasks were used with each task type being equally likely. One thousand machines were used with each machine type being equally likely. Three different methods are used to generate the **ETC** matrix. The "random" method has independent elements that are uniformly distributed from 1 s to 10 s. The "range" method is the range-based method described in [1,20] with parameters 100 and 10 for tasks and machines respectively. The coefficient of variation (CoV) based method, denoted CVB, is defined in [1] and is based on the gamma distribution. The CoV used for the tasks and machines is 0.6 with a mean of 10 s. Fig. 7 shows the makespan and run time of the min–min and max–min relative to LP-makespan for 200 different systems for each **ETC** generation method.

The LP-makespan algorithm took only 64 ms to schedule one million tasks to one thousand machines in Fig. 7. For ten million tasks and ten thousand machines the LP-makespan algorithm takes only 0.87 s while the min–min takes over 476 s to produce a schedule who's makespan is longer than LP-makespan.

From Figs. 5–7 it can be seen that for large problems the LP-makespan algorithm is preferred. For the HPC environments under consideration, the LP-makespan algorithm has smaller run times and shorter schedules compared to both the min–min and max–min algorithms.

6. Computational complexity

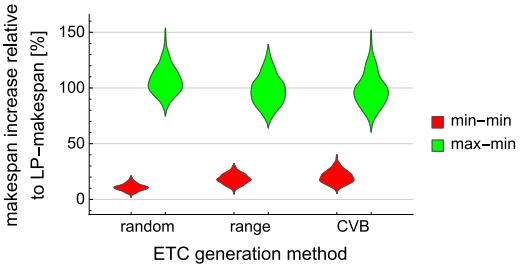
6.1. Analysis

A complexity analysis of each phase of the LP-makespan algorithm reveals desirable properties. A real-valued LP problem must be solved to compute the lower bound on the makespan. Using the simplex algorithm to solve the LP problem yields exponential complexity (i.e., traversing all the vertices of the polytope) in the worst case; however the average case complexity for a very large class of problems is polynomial time [3]. Recall that there are T task types and M machine types. The lower bound LP problem has $T + M$ nontrivial constraints and $TM + 1$ variables. The average case complexity of computing the lower bound is $(T + M)^2(TM + 1)$. Next is the rounding algorithm. The outer loop iterates T times, and the rounding is dominated by the sorting of M items. Thus the complexity of rounding algorithm defined by Algorithm 1 is $\mathcal{O}(T(M \log M))$. The local assignment algorithm defined by Algorithm 2 has an outer loop that is run M times. Inside this loop there are two steps. The first step is sorting at most T items which takes $\mathcal{O}(T \log T)$ time. The second step is a loop that iterates $n_j = \sum_i \mu_{ij}$ times and finds the machine with the earliest ready time each iteration, a procedure with $\mathcal{O}(\log M_j)$ complexity. The worst case complexity of local assignment is thus $\mathcal{O}\left(M \max_j \left(T \log T + n_j \log M_j\right)\right)$.

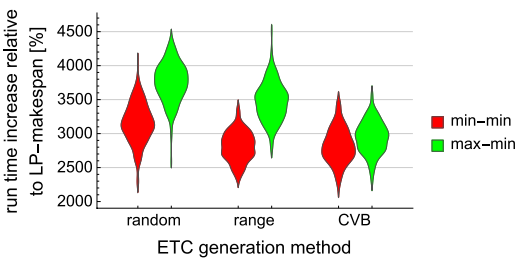
Let $T_{\text{total}} = \sum_i T_i$ be the total number of tasks and $M_{\text{total}} = \sum_j M_j$ be the total number of machines. Assume for the sake of analysis that tasks and machines are evenly distributed across machine types so $n_j \approx \frac{T_{\text{total}}}{M}$ and $M_j \approx \frac{M_{\text{total}}}{M}$.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

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

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



(a) Makespan.



(b) Run time.

Fig. 7. Distributions of the (a) makespan and (b) run time of min–min and max–min relative to LP-makespan: For each **ETC** generation method, 200 different environments were used. LP-makespan has a smaller makespan in every case and is over 20 times faster.

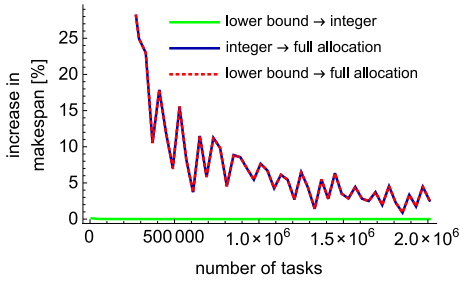


Fig. 8. Relative percent increase in makespan as a function of the *total number of tasks*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. The relative increase in makespan decreases, thus the quality of the solution improves, as more tasks are used.

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

6.2. Results

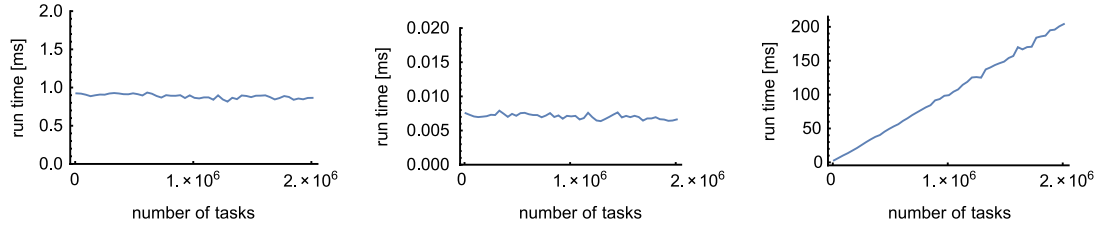
An important property of a scheduling algorithm is its ability to scale well as the size of the problem grows. Simulations were carried out to quantify how the relative error and the computational cost of the algorithm scales. These simulations are used to validate the complexity analysis results from Section 6.1. The environment used for this set of simulations is a scaled-up version of our typical nine machine type environment. The number of machines was increased to 36,000 and the number of tasks was increased to 1,100,000, still with nine machine types and 30 task types, respectively. The distributions of the task types and machines types remain the same as the nine machine type environment.

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

Fig. 8 shows the relative change in makespan as the number of tasks increase. The number of task types, machines, and machine types are held constant and are the same as the nominal environment. The relative increase in makespan is shown from the makespan lower bound, MS_{LB}, to the makespan after rounding. Also shown is the increase in makespan from the integer solution to the full allocation. The relative increase in makespan from the lower bound to the upper bound or full allocation is also shown. The loss in quality of the makespan from the rounding algorithm is relatively low. Most of the increase in makespan is caused by local assignment. However, Fig. 8 also shows that the relative increase in makespan diminishes as the number of tasks increase. This is because the approximation that tasks are divisible has less of an impact on the solution as the number of tasks per machine increases. Fig. 8 shows a cyclical or periodic pattern in the quality of the local assignment algorithm. This pattern is not present in the lower bound or the integer solutions. This pattern is caused by the discrete nature of the problem of assigning tasks to machines. The makespan can increase significantly when just one task is added to the bag-of-tasks that does not pack well onto the machines. Recall that local assignment, by design, only assigns tasks to within a single type of machine so the degrees of freedom are limited in how the algorithm can distribute the load and mitigate the peaks in the relative makespan.

To quantify the computational efficiency of our algorithms, we show the run time of the techniques as a function of the number of tasks in Fig. 9. Fig. 9(a) is the time taken to compute the lower bound (i.e., solve the LP problem). Fig. 9(b) shows the time required to round the solution. Both of the computations required to compute the lower bound and the integer solution do not depend on the number of tasks. This corresponds to the results derived for the complexity of the algorithm. Fig. 9(c) shows that the local assignment algorithm scales linearly with the number of tasks. This also corresponds to the analysis in Section 6.1. Notice that the magnitude of the run times are rather small. Even for 10^8 tasks (not shown in the figure) the total run time is only 8.4 s running on a single core. The LP-makespan algorithm is highly parallelizable so further improvements in runtime could be made if necessary.

The relative increase in makespan when varying the total number of machines is shown in Fig. 10. The figure shows the same three curves as Fig. 8, however in this case, varying the total number of machines. The number of tasks, machine types, and task types are held constant. As the number of machines grow, the increase in makespan due to the local assignment step grows rapidly. This is caused by assigning fewer tasks to each machine as the number of machines increases. The approximation that tasks



(a) Lower bound. (b) Rounding. (c) Local assignment.

Fig. 9. Algorithm run time versus *total number of tasks*: Both the lower bound and the rounding algorithms run time, (a) and (b) respectively, are independent of the number of tasks. The local assignment complexity (c), used to obtain the full allocation, is linearly dependent on the number of tasks.

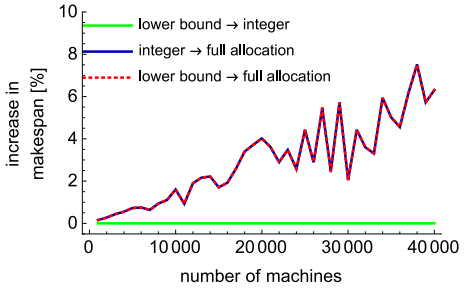


Fig. 10. Relative percent increase in makespan as a function of the *total number of machines*: The relative increase in makespan is shown between the lower bound and integer solutions, the integer and full allocation solutions, and the lower bound and full allocation solutions. The quality of the solution decreases as more machines are used.

are divisible becomes a worse approximation as the number of machines increases relative to the number of tasks.

Fig. 11 shows the run time of the three parts of the scheduling algorithm as the total number of machines is varied. Both the lower bound and the rounding are independent of the number of machines. The local assignment step is approximately logarithmic in the number of machines. This corresponds to the analysis in Section 6.1.

Results for sweeping the task types and machine types are presented in the appendix (see Appendix A).

Even though the run time and solution quality of the polynomial time LP-makespan algorithm is desirable, there is some prior work on theoretical bounds that should be noted. In [18], 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. 8–11 and A.12–A.15 (see Appendix A) suggest that one can do better then 3/2, this is only the case on average.

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

7. Related work

The LP-based approach in this paper achieves significant decrease in run time and increases in solution quality over prior methods by exploiting properties that are common to static scheduling problems. Our approach takes advantage of the common property that each machine in an HPC system is not unique but belongs to one of a few types of machines. Our work also is focused on very large-scale environments and finding high quality solutions on average, whereas [22,13] are concerned with worst-case performance of the scheduling algorithms.

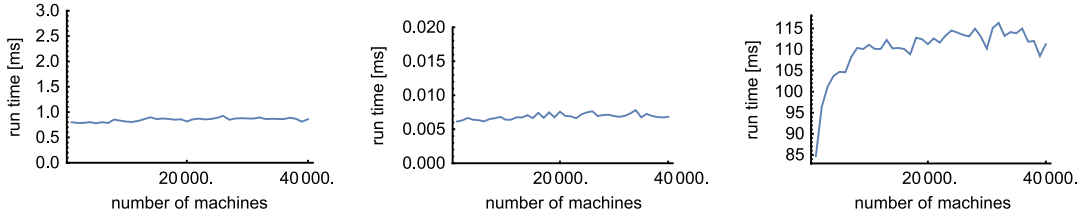
Static scheduling for minimum makespan is surveyed in [7]. Min–min and max–min or a hybrid of both algorithms are found to generally be the best algorithms for this problem domain [29]. Our results in Section 5 show that min–min almost always performs better than max–min. The max–min tends to perform better then min–min when there are many more short running tasks than long running tasks [20]. The min–min algorithm will schedule the shorter tasks to run on all the machines leaving the fewer long tasks to the end, increasing the makespan. In our simulations there are similar numbers of short and long tasks so the key conceptual benefit of max–min cannot be achieved.

While this paper deals with scheduling tasks to entire machines, the algorithms could also be applied to scheduling tasks to cores within a machine or across cores on many machines. The full allocation recovery algorithm we use is conceptually similar to the algorithms presented in [19]; however, those algorithms are designed for scheduling tasks on a single machine with deadlines to determine the best dynamic voltage and frequency scaling (DVFS) parameters to use to minimize energy as a secondary objective. Another related algorithm is presented in [2] that approximates makespan to provide computationally efficient schedules while considering reliability for DVFS scheduling on identical processors.

In [16], the A* search algorithm is used to assign tasks to machines considering task dependencies and communication constraints. This algorithm is very expensive for large numbers of tasks because the algorithm's branching factor is on the order of the number of machines and the depth is on the order of the number of tasks.

Allocating services running within virtual machines to physical machines is addressed in [23]. The services being considered are CPU bound processes that are allocated fractions of machines. Multiple smaller services can be allocated to one machine. Their approach is similar to this paper in that they formulate a linear program, solve the relaxation, and then recover a feasible solution. The authors note that using binary variables degrades the quality of the solution from the rounding methods used after solving the linear program. We try to address this issue by formulating the linear program to have decision variables that are large values that round easily to large integers, resulting in little degradation in the quality of the solution. They also propose a genetic algorithm (GA) and heuristic algorithms to solve the problem faster and with a higher quality than rounding the result of their linear program. The work in [23] is extended from a single homogeneous set of machines to a heterogeneous collection of machines in [24]. Our work focuses on highly scalable algorithms whereas [23,24] focus on algorithms that work on relatively small problem sizes and have non-negligible run times for the schedulers.

Our work in [26] presents algorithms and techniques for building Pareto fronts to trade-off energy and makespan along with bounds on those Pareto fronts. We also derive the upper and lower bounds on the true Pareto front.



(a) Lower bound. (b) Rounding. (c) Local assignment.

Fig. 11. Algorithm run time versus *total number of machines*: Both the lower bound and the rounding algorithm run times, (a) and (b) respectively, are independent of the number of machines. The local assignment complexity (c), used to obtain the full allocation, is logarithmically dependent on the number of machines.

8. Conclusions

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

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

Acknowledgments

This work was supported by the Sjostrom Family Scholarship, Numerica Corporation, the National Science Foundation (NSF) under grants CNS-0905399 and CCF-1302693, a NSF Graduate Research Fellowship, and by the Colorado State University George T. Abell Endowment. Any opinions, 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 has been presented in [25]. A special thanks to Mark Oxley and Bhavesh Khemka for their valuable comments.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.jpdc.2015.07.002.

References

[1] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang Journal of Science and Engineering 3 (3) (2000) 195–208. Special Tamkang University 50th Anniversary Issue, Invited.

[2] G. Aupy, A. Benoit, Y. Robert, Energy-aware scheduling under reliability and makespan constraints, in: 19th International Conference on High Performance Computing (HiPC), 2012, pp. 1–10. http://dx.doi.org/10.1109/HiPC.2012.6507482.

[3] D. Bertsimas, J. Tsitsiklis, Introduction to Linear Optimization, first ed., Athena Scientific, 1997.

[4] V. Bharadwaj, D. Ghose, T.G. Robertazzi, Divisible load theory: A new paradigm for load scheduling in distributed systems, Cluster Comput. 6 (1) (2003) 7–17.

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

[6] P.E. Black, Ragged matrix, in: Dictionary of Algorithms and Data Structures, NIST, 2004, URL http://www.nist.gov/dads/HTML/raggedmatrix.html.

[7] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, R.F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 61 (6) (2001) 810–837. http://dx.doi.org/10.1006/jpdc.2000.1714.

[8] M.K. Dhodhi, I. Ahmad, A. Yatama, I. Ahmad, An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems, J. Parallel Distrib. Comput. 62 (9) (2002) 1338–1361. http://dx.doi.org/10.1006/jpdc.2002.1850.

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

[10] A. Ghafoor, J. Yang, A distributed heterogeneous supercomputing management system, IEEE Comput. 26 (6) (1993) 78–86. http://dx.doi.org/10.1109/2.214443.

[11] R. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429. http://dx.doi.org/10.1137/0117039.

[12] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, J. ACM 24 (2) (1977) 280–289. http://dx.doi.org/10.1145/322003.322011.

[13] K. Jansen, L. Porkolab, Improved approximation schemes for scheduling unrelated parallel machines, Math. Oper. Res. 26 (2) (2001) 324–338. http://dx.doi.org/10.1287/moor.26.2.324.10559.

[14] E. Jeannot, E. Saule, D. Trystram, Optimizing performance and reliability on heterogeneous parallel systems: Approximation algorithms and heuristics, J. Parallel Distrib. Comput. 72 (2) (2012) 268–280. http://dx.doi.org/10.1016/j.jpdc.2011.11.003.

[15] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurr. 6 (3) (1998) 42–50. http://dx.doi.org/10.1109/4434.708255.

[16] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurr. 6 (3) (1998) 42–50. http://dx.doi.org/10.1109/4434.708255.

[17] A. Khokhar, V. Prasanna, M. Shaaban, C.-L. Wang, Heterogeneous computing: challenges and opportunities, Computer 26 (6) (1993) 18–27. http://dx.doi.org/10.1109/2.214439.

[18] J.K. Lenstra, D.B. Shmoys, E. Tardos, Approximation algorithms for scheduling unrelated parallel machines, Math. Program. 46 (3) (1990) 259–271. http://dx.doi.org/10.1007/BF01585745.

[19] D. Li, J. Wu, Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms, in: 41st International Conference on Parallel Processing (ICPP), 2012, pp. 430–439. http://dx.doi.org/10.1109/ICPP.2012.26.

[20] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, J. Parallel Distrib. Comput. 59 (2) (1999) 107–131. http://dx.doi.org/10.1006/jpdc.1999.1581.

[21] Phoronix Media, Intel core i7 3770k power consumption, thermal, http://openbenchmarking.org/result/1204229-SU-CPUMONITO81 (May 2013).

[22] D.B. Shmoys, E. Tardos, Scheduling unrelated machines with costs, in: Proceedings of the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '93, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993, pp. 448–454. URL http://dl.acm.org/citation.cfm?id=313559.313851.

[23] M. Stillwell, D. Schanzenbach, F. Vivien, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, J. Parallel Distrib. Comput. 70 (9) (2010) 962–974.

[24] M. Stillwell, F. Vivien, H. Casanova, Virtual machine resource allocation for service hosting on heterogeneous distributed platforms, in: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012, pp. 786–797. http://dx.doi.org/10.1109/IPDPS.2012.75.

[25] K.M. Tarplee, R. Friese, A.A. Maciejewski, H.J. Siegel, Efficient and scalable pareto front generation for energy and makespan in heterogeneous computing systems, in: S. Fidanova (Ed.), Recent Advances in Computational Optimization: Results of the Workshop on Computational Optimization WCO 2013, in: Studies in Computational Intelligence, vol. 580, Springer, 2015, pp. 161–180. http://dx.doi.org/10.1007/978-3-319-12631-9_10.

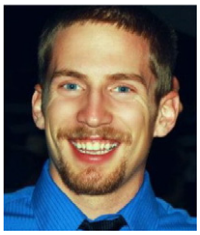
[26] K.M. Tarplee, R. Friese, A.A. Maciejewski, H.J. Siegel, Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques, IEEE Trans. Parallel Distrib. Syst. (2015) in press.

[27] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274. http://dx.doi.org/10.1109/71.993206.

[28] R.S. Witte, J.S. Witte, Statistics, tenth ed., Wiley, 2014.

[29] M.-Y. Wu, W. Shu, A high-performance mapping algorithm for heterogeneous computing systems, in: Proceedings 15th International Parallel and Distributed Processing Symposium, 2001, pp. 74–80. http://dx.doi.org/10.1109/IPDPS.2001.925020.

Kyle M. Tarplee is a Senior member of IEEE. He has worked for Numerica Corporation since 2005 developing multi-target tracking algorithms for the Department of Defense. He has been the principal investigator on several projects. Mr. Tarplee also is pursuing his Ph.D. from Colorado State University in Electrical Engineering. He has a BSEE and Masters from University of California at San Diego. Further details at: https://www.engr.colostate.edu/~ktarplee.



Ryan Friese received dual B.S. degrees in Computer Engineering and Computer Science from Colorado State University (CSU) in 2011. He received his M.S. degree in Electrical Engineering from CSU in the Summer of 2012. He is currently a Ph.D. student at CSU. He is a United States National Science Foundation graduate research fellow. His research interests include heterogeneous computing as well as energy-aware resource allocation. Further details at: http://RyanDFriese.com.

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

Howard Jay Siegel was appointed the Abell Endowed Chair Distinguished Professor of Electrical and Computer Engineering at Colorado State University in 2001, where he is also a Professor of Computer Science. From 1976 to 2001, he was a professor at Purdue University, West Lafayette. He is an IEEE Fellow and an ACM Fellow. He received B.S. degrees from MIT, and the Ph.D. degree from Princeton. A complete vita is available at: https://www.engr.colostate.edu/~hj.