Resource Management in Heterogeneous Parallel Computing Environments with Soft and Hard Deadlines

Bhavesh Khemka[1], Dylan Machovec[1], Christopher Blandin[1], Howard Jay Siegel[1,2], Salim Hariri[3], Ahmed Louri[3], Cihan Tunc[3], Farah Fargo[3], and Anthony A. Maciejewski[1]

[1] Department of Electrical and Computer Engineering
[2] Department of Computer Science
Colorado State University, Fort Collins, CO 80523, USA
Bhavesh.Khemka@colostate.edu, DJMachov@rams.colostate.edu, CBlandin@rams.colostate.edu, HJ@colostate.edu, AAM@colostate.edu

[3] Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721, USA
Hariri@ece.arizona.edu, Louri@ece.arizona.edu, CihanTunc@email.arizona.edu, FarahFarjo@email.arizona.edu

Abstract

Heuristic-based approaches are often used to perform the assignment and scheduling of parallel high-performance computing applications on machines (as it is an NP-hard problem). The time-varying importance of tasks has been represented using monotonically-decreasing value functions. The value function of a task gives the value that will be earned based on when the task completes execution and is a flexible method to incorporate soft and hard deadlines. To measure the amount of useful work accomplished in an oversubscribed heterogeneous parallel computing environment, we conduct a simulation study to evaluate the performance of schedulers based on the total value they earn from executing tasks. To optimize for such an objective (as opposed to traditional response time-based or fairness-based objectives), requires the design of novel parallel scheduling metaheuristics. We design new metaheuristics and create a new concept of "place-holder" tasks that gives our metaheuristics the advantages of reservations, but also allows future tasks to nullify those reservations. We examine the performance of our metaheuristics and other popular parallel scheduling techniques such as EASY Backfilling and Conservative Backfilling in a variety of environmental conditions. Our real workload trace-driven simulations show that our metaheuristics that use the concept of place-holders consistently outperform the other techniques, in terms of the total value earned, improving on average EASY Backfilling by over 50% and Conservative Backfilling by over 100%.

1 Introduction

High-performance computing (HPC) systems are increasingly used to solve a host of important problems that are implemented as large parallel applications. The assignment and scheduling of such parallel applications on machines for execution is an NP-hard problem. Therefore, heuristic-based approaches are often used to perform the resource management.

To establish a metric for comparing metaheuristics, value functions have been used to more accurately represent the time-varying nature of the importance of tasks (e.g., [8, 10, 12, 19]). In our environment, tasks dynamically arrive, and the value function of a task specifies the "value" that will be earned for completing the execution of the task at any time after the task has arrived. A task earns a certain starting value if it is finished by its soft deadline. After the soft deadline, the value of a task starts to decrease linearly until the hard deadline of the task, and after that the task earns no value. An oversubscribed environment is one where the workload of tasks arriving exceeds the compute capacity of the system, and as a result, not all tasks can complete by their soft deadlines. To measure the amount of useful work accomplished by a system in oversubscribed serial environments, the performance of a scheduler has been measured as the total value earned from completing tasks [8]. We apply this concept to the field of

This work was partially supported by the National Science Foundation (NSF) under grant numbers CCF-1302693, CNS-1318997, and CCF-1439142, and by AFOSR DDDAS award number FA95550-12-1-0241. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

parallel scheduling where typically scheduling metaheuristics have been designed for time-based objectives. Further, we model a heterogeneous computing environment where the compute system comprises clusters with different performance capabilities. Each of the clusters have a homogeneous set of compute nodes. In such a heterogeneous environment, cluster A's nodes may be faster than cluster B's nodes to execute a task, but cluster B's nodes may be faster for some other task.

Making reservations is important in parallel scheduling to prevent the starvation of tasks that need a large number of computing elements. Once such reservations are made, they can create idle slots in the schedule of the nodes. Therefore, after making reservations it is important to be able to backfill tasks into those idle slots to improve the utilization of the resources. Standard parallel scheduling heuristics employ time-based criteria such as response time and average bounded slowdown, and prioritize tasks based on the arrival time to ensure fairness in scheduling. After "prioritizing" tasks in such a manner, higher-priority tasks that cannot fit into currently available slots are reserved for a future time and other tasks may be backfilled in the newly created idle slots (e.g., EASY Backfilling [13] and Conservative Backfilling [16]).

In our oversubscribed environment, where tasks have value functions representing their time-varying importance, new metaheuristics are needed that can prioritize tasks based on these value functions. Once a reservation is made for a task, it guarantees the task will execute at that reserved time. We show that in an oversubscribed value function-based environment, this may be sub-optimal. This is because, by the reservation time, a task that can earn an even higher value may arrive into the system. In such a situation, the overall total value that can be earned may be higher if the newly arrived task is scheduled to execute in place of the reserved task. Reservations prohibit this flexibility. Motivated by this, we present the novel concept of place-holders that give the metaheuristics the benefits of making reservations, but provide the flexibility of executing other tasks in those "reserved" slots.

We use real parallel workload traces in our simulations to evaluate the performance of the heuristics in terms of the total value earned from completing tasks. We also perform experiments under a variety of environmental conditions. The main contributions of this work are: (1) studying the problem of maximizing the value earned by tasks in a parallel scheduling environment (as opposed to traditional time-based metrics), (2) designing new metaheuristics to solve the above problem – including the novel concept of place-holder tasks that have the benefits of reservations but not their drawbacks in a value-based oversubscribed environment, and (3) using simulation studies to test, compare, and analyze the performance of the new metaheuristics along with traditional parallel scheduling techniques in a wide variety of environments that differ in the level of oversubscription, heterogeneity, and correlation of the worth of a task with its average execution time.

2 Problem Description

Workload Model: We model a system where the tasks arrive dynamically. The workload comprises both serial and parallel tasks. Parallel tasks specify the number of compute cores that they concurrently need for their execution. As it is typical in most HPC environments, all of the submitted tasks are independent (i.e., they do not need to communicate with each other).

The importance and urgency of each submitted task is represented by its unique value function. The *value function* of a task is a monotonically-decreasing function that specifies the value earned when completing the task at different times. For a given type of application, the value function is designed by the user submitting the task in collaboration with the system owner and is defined by four parameters: a soft deadline, a hard deadline, a starting value, and a final value. Figure 1 shows a sample value function. Value functions begin at a *starting value* and remain at that value until their *soft deadline*. After the soft deadline has passed, the value of the task decreases linearly until it reaches the *final value* at the *hard deadline* of the task. After the hard deadline, the value drops to zero. The soft and hard deadlines are defined after the arrival time of the task. The value functions are similar in principle to the representation of different deadlines and importance levels of tasks in [8, 10, 12, 19]. The techniques and analyses that we develop are applicable to any arbitrarily-shaped monotonically-decreasing value functions.

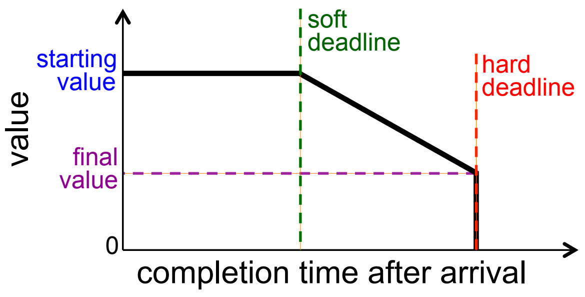


Figure 1: The form of a value function that is parameterized by four numbers: soft deadline, hard deadline, starting value, and final value.

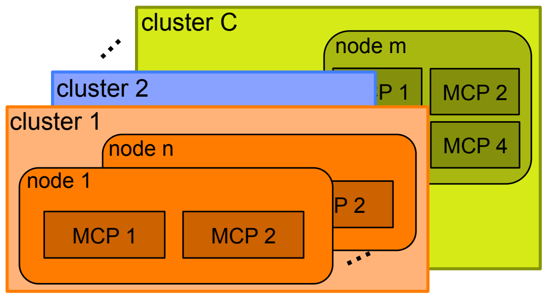


Figure 2: Compute system comprising heterogeneous clusters that are made up of homogeneous compute nodes that contain one or more multicore processors (MCPs).

Compute System Model: The compute environment that we model consists of a set of heterogeneous clusters. Each of the clusters consists of compute nodes that each have one or more multicore processors (see Figure 2). Each of the clusters themselves are homogeneous and the architecture of all of the compute nodes within a cluster is identical. Therefore, all nodes within a cluster have the same number of compute cores. Task assignments are made at the node-level such that no two tasks share any node. In addition, a task cannot be split across clusters, i.e., all of its allocated nodes must be on the same cluster. A task is allocated the minimum number of nodes that provides its requested number of cores. For example, if a task (t_1) that requests twelve cores is to be assigned on a cluster (c_1) that has eight cores per node, then, that task would be allocated two nodes for its execution. Even though this may result in four cores not being used, this is a common strategy in HPC systems as it prevents performance degradation from different tasks interfering with each other. Based on the number of cores a task t_i requests and the number of cores per node on a cluster c_j, we can compute the number of cores that will be allocated to t_i when it is executed on c_j, and we represent this as $NC(t_i, c_j)$. For the above example, $NC(t_1, c_1) = 16$. All of these characteristics and assumptions are typical of most HPC systems and clusters (e.g., [3, 4]).

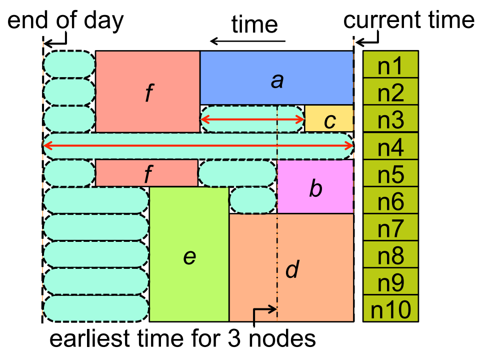
In many HPC centers such as those for satellite image processing, weather prediction, seismic analyses, etc., the tasks that arrive belong to a known set. In such environments, through historical data or experiments, one can obtain the execution time information for the different tasks on each of the compute clusters. This is a common assumption in heterogeneous resource management literature (e.g., [5, 7, 9]). For task t_i's execution on cluster c_j, we assume we are given the *E*stimated *T*ime to *C*ompute ($ETC(t_i, c_j)$). We define the amount of resources used by a task t_i on a cluster c_j ($R(t_i, c_j)$) as the product of the execution time of that allocation and the number of cores allocated for that execution. Therefore, $R(t_i, c_j) = ETC(t_i, c_j) \times NC(t_i, c_j)$.

Problem Statement: Recall that we model an oversubscribed heterogeneous parallel computing environment where tasks arrive dynamically for execution. The arrival time and the value function of each task is not known *a priori*. Depending on when tasks complete execution (based on how they are scheduled), we can compute the value earned from them. The goal of the metaheuristics is to maximize the total value earned from executing tasks that arrive into the system.

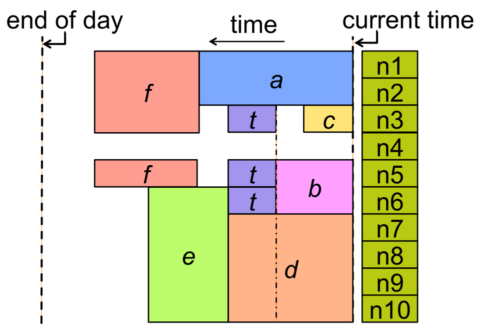
3 Resource Management

3.1 Mapping Events

A *mapping event* is an instant at which scheduling decisions are made. In our environment, mapping events are triggered either when a new task arrives into the system or when a node becomes available after completing the execution of a task. At a mapping event, all tasks that have not started execution or have not been reserved for execution are considered *mappable*. First, a check is performed to remove any tasks that will only be able to earn zero value even if they were allowed to immediately start execution



(a) State of system before selecting nodes.



(b) Selected nodes for task *t*.

Figure 3: Showing an example state of ten nodes in a cluster. The rectangles represent task executions or reservations and the bubbles represent the idle slots in the schedule up to the end of day. The earliest possible time when three nodes are concurrently available is shown in (a). As shown in (b), for task *t* (that requires three nodes), node n4 would create the most fragmentation and is therefore not chosen.

on their best execution time cluster. Then, a metaheuristic is called to make resource allocation decisions for the remaining mappable tasks. We do not allow preemption in our environment, i.e., once a task starts executing, it executes until completion.

3.2 Picking Earliest Slot for a Task

Metaheuristics will need to know the earliest available time at which a given task can start execution on a given cluster. On a cluster, multiple nodes may tie in terms of the earliest available time for a task. In such situations, we use a heuristic technique to break ties with the goal of reducing fragmentation and increasing the flexibility for later assignments. All of the metaheuristic techniques use our new approach.

Idle slots are the time intervals where the nodes are not scheduled for any execution. Figure 3(a) shows an example cluster with ten nodes, the tasks assigned to it, and the idle slots in its schedule. Note that idle slots can extend up to the end of the day. As we do not allow preemption, any new task will be assigned in some space occupied by the idle slots. Consider the scenario when slots need to be determined for executing task *t* that needs three nodes for its execution in the cluster shown in Figure 3(a). The earliest available time when three nodes are concurrently available is shown. At that time, there are four nodes available in all. The first condition we use to break ties is picking the node that has the smallest change in the number of idle slots after making the assignment. Comparing Figures 3(a) and 3(b), we observe that assigning task *t* to node n6 changes its number of idle slots by -1. Assigning *t* to node n5 changes its number of idle slots by 0 (i.e., no change). For nodes n3 and n4, task *t* splits an idle slot into two, and therefore changes the number of idle slots for each of those nodes by $+1$. Based on this, we would pick nodes n6 and n5, because the new assignment is immediately after and/or immediately before an existing task. The other two nodes (n3 and n4) tie in terms of the change in the number of idle slots. We compare the size of the idle slots into which the assignment is being made and select the one with the smallest size. The red arrows in Figure 3(a) represent the duration of the idle slots for the tied nodes. As node n3 has the shorter idle slot, it is selected for task *t*. Having a longer idle slot gives more flexibility in selecting a task to assign to that slot.

3.3 Metaheuristics

EASY Backfilling: Extensible Argonne Scheduling sYstem (EASY) Backfilling [13] is one of the most popular scheduling metaheuristics for parallel tasks. The characteristic feature about EASY is that it only maintains one reservation at any time. EASY considers tasks in the order of their arrival. When a mapping event is triggered, EASY works through the queue and schedules tasks on the currently available nodes without delaying the reserved task (if one exists). When it encounters a task that cannot start execution on the currently available nodes it either makes a reservation for this task at the earliest possible start time (if there is no reserved task already) or it skips consideration of this task (if there already is a reserved task). Once a task reservation is in place, the tasks in the queue are only considered for backfilling. Tasks are backfilled if they satisfy two conditions: (1) they cannot delay the start time of the reserved task and (2) they must start execution immediately on some currently available node(s).

Conservative Backfilling: Conservative Backfilling [16] was designed to enforce fairness more strictly than EASY by making sure that no later arriving tasks can delay the start of earlier arriving tasks. EASY Backfilling aggressively backfills tasks as long as they do not delay the start of the one reserved task. In EASY, a task that is backfilled may delay an earlier arriving task that was unassigned because it could not start immediately and another task already had a reservation. To overcome this, Conservative Backfilling considers tasks in the order of their arrival and makes reservations for all tasks that cannot fit into the currently available nodes. The only condition when scheduling or reserving tasks is that the assignment should not delay the start of any other reservation.

Conservative Multiple Queues: This metaheuristic separates tasks based on their resource usage into three different queues: small, medium, and large. The goal of this metaheuristic is to emulate how some systems such as the CSU ISTeC Cray [3] perform scheduling. The idea behind this metaheuristic is that earlier arriving tasks and smaller-sized tasks get a higher priority of being scheduled. Let C be the number of clusters in the environment. We use the average of the resources used by a task t_i across the clusters $\widetilde{R}(t_i) = (\sum_{j=1}^{C} R(t_i, c_j))/C$ to represent its average resource usage. The maximum value of $\widetilde{R}(t_i)$ over all tasks t_i is represented as R_{max}. To partition the tasks into the three different queues, we compare the task's $\widetilde{R}(t_i)$ with R_{max}. If $\widetilde{R}(t_i) \leq 0.3R_{max}$, the task is inserted into the small queue, if $0.3R_{max} < \widetilde{R}(t_i) \leq 0.6R_{max}$ the task is inserted into the medium queue, and if $\widetilde{R}(t_i) > 0.6R_{max}$ the task is inserted into the large queue. Within each of the queues, the tasks are ordered based on their arrival time. The number of queues and the strategy for dividing tasks among them can be set by the system owners. Once the incoming tasks have been partitioned into the three queues, the metaheuristic cycles through the different queues in a round-robin fashion considering more tasks from the queues with smaller-sized tasks. For example, in our implementation, the metaheuristic selects one, four, and eight tasks from the large, medium, and small queues, respectively, before repeating. The metaheuristic makes assignments for each task it considers in a Conservative Backfilling style, i.e., starts the execution of the task or makes a reservation for it at the earliest available slot that does not delay any reservations.

These first three metaheuristics (EASY Backfilling, Conservative Backfilling, and Conservative Multiple Queues) are implemented for comparison purposes. These techniques do not use the heterogeneity of the environment nor the value functions of the tasks.

Maximum Value: This metaheuristic is designed based on the Min-Min technique [2, 6, 14]. The metaheuristic independently finds for each mappable task, the allocation choice that: (1) does not violate any reservations and (2) maximizes the value earned for that task. If a task has multiple choices that tie for the same maximum value, then, the one with the earliest completion time is selected. The assignment over all mappable tasks that earns the maximum value is chosen and a reservation for it is made (or its execution is started). The metaheuristic updates the system state information based on the newly made assignment. It then repeats until all mappable tasks have either started execution or been reserved.

Maximum Value-Per-Resource (Maximum VPR): In an oversubscribed heterogeneous environment, it may help to pick choices that provide the most value while using the least amount of resources. This metaheuristic is similar to Maximum Value but instead of maximizing value, it picks the choice that maximizes: "value earned / amount of resources allocated." The amount of resources allocated for an execution is computed using the technique defined in Section 2.

Maximum Value and Maximum VPR with Place-holders: In Maximum Value, reservations ensure that tasks that earn lower value are only scheduled as long they do not affect the start of the higher-value tasks. The disadvantage of reservations in an oversubscribed environment like ours can be explained by considering the situation that a task that can earn a higher value than a task with a reservation arrives into the system at a later time. This can result in reduced performance. To prevent this, we introduce the concept of place-holder tasks to be used instead of reservations. *Place-holders* are like reservations that are only active during the mapping event, i.e., when making resource allocation decisions. Once the mapping event is completed, the place-holders are removed from the schedule and those place-holder slots are just treated as any idle slots. During the mapping event, it draws on the benefits of reservations by preventing the assignment of lower value tasks that may disturb the execution of higher value tasks. After the mapping event finishes, those place-holder slots are freed and allow any newly arrived task of higher value to be executed in those slots. The tasks that were assigned place-holder slots during a mapping event will be present in the mappable set when the metaheuristic is called in subsequent mapping events, and if no other higher value task becomes available, they may be executed as initially planned by the place-holder. In this way, the place-holder technique does not lose the benefits that reservations provide and adds flexibility for the next mapping event.

It is worth noting that the concept of place-holders is not applicable to the other metaheuristics because they prioritize tasks based on arrival time. Therefore, a later arriving task would never be scheduled at a sooner time than an earlier reserved task to warrant executing it in the reserved slot.

4 Simulation Setup

We create 48 simulation scenarios (our simulator uses two 24 core nodes on the Colorado State University ISTeC Cray HPC cluster [3]). The scenarios differ in the workload of tasks (having different number of tasks, arrival times, value functions, and number of requested cores) and in the environment architecture (having different number of clusters, number of nodes within a cluster, and number of cores within the nodes). Each scenario also has different ETC values. This section describes our method to generate these values for each of 48 simulation scenarios.

We use the log of the Curie Supercomputer in France from Dror Feitelson's Parallel Workloads Archive [17] to model the arrival of tasks. From the "clean" version of the Curie trace, we picked 48 days for our simulations. We simulate an arrival of 28 hours using the last four hours from previous simulated day to bring the system up to steady state. All of our results are collected from the remaining 24-hour period to avoid the scenario where the machines start with empty queues. To keep the size of the simulations tractable, we removed tasks that requested more than 4,096 cores for their execution. Over the 48 days, 1,157 tasks were removed out of 76,523 tasks total ($\approx$1.51%).

The starting value for a task's value function was obtained by sampling a gamma distribution. For each task, the mean of the distribution was selected to be proportional to the average execution time and was generated by linearly interpolating between the range of 5 to 50, with a 1 second execution time corresponding to a value of 5 and the maximum run time from the entire trace (in seconds) corresponding to a value of 50. The Coefficient Of Variation (COV) was set to 2.5. The starting values were bounded to be within the range [1, 100]. This generates an environment where the starting value of a task is weakly correlated with its average execution time. Such a correlation has been modeled in the literature [19]. We use the notation *U(a,b)* to denote the sampling of a number uniformly at random from the range [a,b]. The final value at the hard deadline for a task's value function was obtained by multiplying the start value of the task (at the soft deadline) with the factor: U(0.01, 0.8). The soft deadline time for a task was set to U(0.9, 1.2) times the average execution time of the task. The hard deadline time was set to the soft deadline + U(0, 1.5) times the average execution time of the task. We directly used the run times provided for each task in the trace as the first column of the ETC matrix. To generate the other columns of the ETC matrix, we used the COV method [1] with a COV of 0.3.

To model an oversubscribed computing environment, we set the total number of cores for our system to be approximately 20% of the total number of cores in the Curie system resulting in 18,432 cores. Even after such a significant reduction in the size of the system, our best metaheuristics could successfully execute about 82% of the tasks. Using the mean of 18,432 cores and a COV of 0.05, we sampled gamma distributions to generate the total number of cores in the compute system for each scenario. The number of clusters in the different scenarios was picked from the set: 2, 3, 4. The number of cores per node for a cluster was picked from the set: 1, 2, 4, 8, 16, 24, 32. Finally, to split the total number of cores across the different clusters, we use the following randomized technique. The number of unclaimed cores is initialized to the total number of cores. For each cluster, its number of cores is picked as U(0.1, 0.5) times the number of unclaimed cores. Then, the number of unclaimed cores is updated and the process is repeated until the second-to-last cluster. The last cluster is given the number of unclaimed cores remaining. Then, the number of nodes per cluster are calculated.

5 Results

All of the results are averaged over the 48 simulation scenarios and the figures show 95% confidence interval error bars. The arrival patterns and the number of task arrivals for the different days from the trace varies significantly. Among the 48 days we studied, the least and the most number of task arrivals in a day was 897 and 2904 tasks, respectively. To more accurately compare the performance, we compute the percentage of a maximum value bound that each metaheuristic earns for each scenario and then take the average. The maximum value bound for a scenario is computed by summing the starting values of all tasks. All results are shown as a percentage of the maximum value bound earned by the metaheuristics. The number of tasks that the different metaheuristics could complete in the day were very similar to the trends for the percentage of maximum value earned.

We refer to the environment described in Section 4 as "intermediate oversubscription," and the results from it are shown in Figure 4. Conservative Backfilling performs the worst in our environment because once reservations are made for all tasks, it becomes harder to find slots to backfill new tasks. EASY Backfilling more easily finds slots to backfill any newly arrived tasks and services them quickly (earning meaningful value from them). Conservative Multiple Queues performs better than Conservative Backfilling because it partitions the tasks into queues of different sizes and uses round-robin through them as opposed to working through the whole set of tasks in a first-come-first-served manner.

In general, the value-based metaheuristics earn more value than the popular parallel scheduling techniques because they directly optimize for value and exploit the heterogeneity of the environment. This allows the metaheuristics to execute more tasks, and as a result earn even higher value.

The improvement by adding place-holders to Maximum Value and Maximum VPR is $\approx$28% and $\approx$34%, respectively. This highlights the importance of the increased flexibility provided by the place-holders. In addition, the number of executed tasks by the place-holder version of the metaheuristics was also higher than that by their non-place-holder counterparts. This is because when reservations are made for all tasks in a mapping event, it becomes harder to find slots to execute tasks that arrive later. By using place-holders and keeping those slots open, it is much easier to find allocations for tasks in subsequent mapping events. This improves the packing of task executions more efficiently, which results in more tasks being executed and that leads to higher value being earned overall.

A possible downside to using place-holders (instead of reservations) is the increased computation overhead. This is because place-holder tasks will continue to remain in the mappable set and the metaheuristic will have to perform allocation decisions for those tasks again. For the "intermediate oversubscription" environment, this was not a concern as the average execution time of each of the different techniques was sub-second. The non-place-holder metaheuristics took on average $\approx$0.006 seconds while the place-holder metaheuristics took on average $\approx$0.02 seconds for a mapping event.

We also studied the performance of the metaheuristics under a variety of conditions. We generated new environments with lower and higher oversubscription by using 25% and 15%, respectively, of the total core count of the Curie system (the "lower" oversubscription environment is still oversubscribed). Results from this are shown in Figure 4. All of the metaheuristics performed better under lower oversubscription but performed worse when the amount of oversubscription was increased. The change in performance for the place-holder metaheuristics is less than that for the other metaheuristics. The relative performance among the metaheuristics is the same across the oversubscription levels.

Recall that in our environment, the starting value of a task is weakly correlated with its average execution time. We created two new environments to evaluate the performance of the metaheuristics: (1) where the starting value of a task was exactly correlated with its average execution time and (2) where the starting value of a task was completely uncorrelated with its average execution time (i.e., the starting value is U(1, 100) ignoring the average execution time). The results from these environments are shown in Figure 5(a). It is worth noting that in a perfectly correlated environment, in general, the

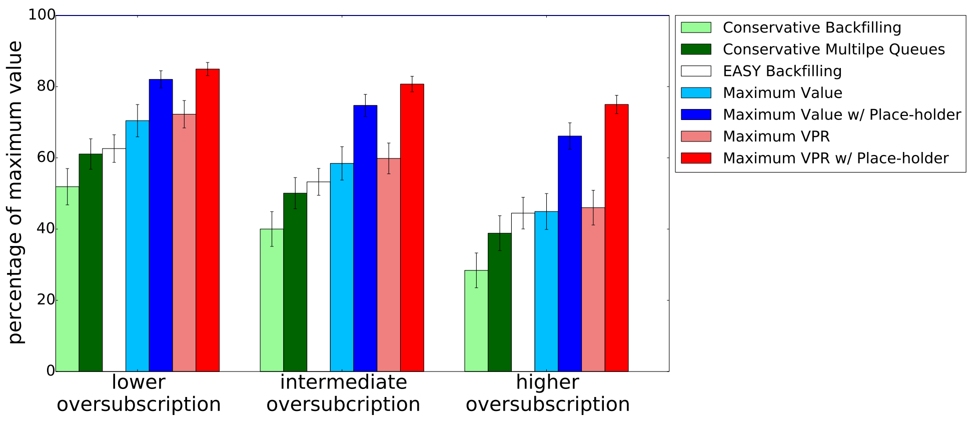


Figure 4: Percentage of the maximum value bound earned by the different metaheuristics in three different levels of oversubscription.

more the systems are utilized, the more value they earn. Therefore, metaheuristics are not penalized heavily for not considering value functions of tasks. Even in such an environment, the value-based place-holder metaheuristics perform better because they have the ability to better pack tasks onto the compute resources that execute them efficiently, thereby increasing the utilization of the system resources and the number of tasks they can complete, leading to higher value earned overall. In the uncorrelated environment, the insight provided by these best-performing metaheuristics is critical to making good decisions and maximizing the value earned.

Figure 5(b) shows the performance of the different techniques under environments with lower and higher heterogeneity than the current environment that we call "intermediate." The ETCs with lower and higher heterogeneity were created by using COV values of 0.01 and 1, respectively. Although, the value-based place-holder metaheuristics consistently outperform the other techniques.

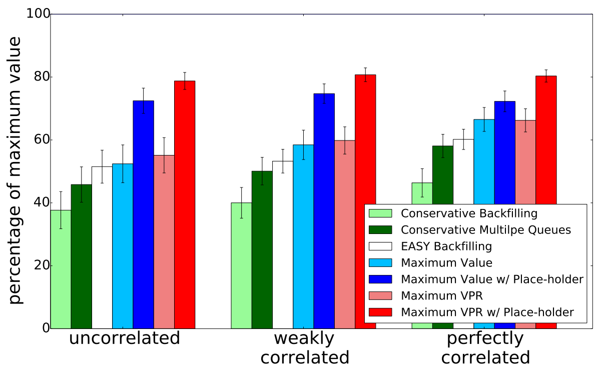
All of these experiments emphasize an important trait of the value-based place-holder metaheuristics, i.e., their performance is much more resilient to changes in the environmental conditions than any of the other techniques. Further, Maximum VPR (with place-holder) consistently outperforms all other metaheuristics including Maximum Value (with place-holder) in all cases.

6 Related Work

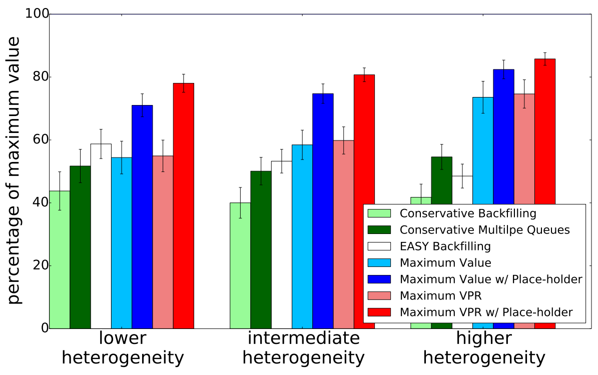
Parallel scheduling of tasks has been studied extensively, but most works have examined homogeneous computing environments that are not oversubscribed. Further, almost all parallel scheduling works are designed to optimize for time-based and fairness objectives (e.g., [11, 13, 15, 16, 20]). A common technique in parallel task scheduling is to modify/improve the EASY or Conservative Backfilling techniques. In [15], Conservative Backfilling is modified to create the shortest job first backfilling that improves the average turnaround time for tasks. Another work uses an iterative Tabu search algorithm to improve the fairness of the schedule created by Conservative Backfilling [11]. As mentioned before, our problem and the solution techniques for them are very different from such time-based or fairness-oriented scheduling.

Some works have used value-based approaches in parallel scheduling but they both do not consider a heterogeneous computing environment [12, 19]. In [12], the authors use a genetic algorithm to solve the problem of maximizing the total value earned. As we study a dynamic environment where allocation decisions need to be made on-line (and very quickly), such global-search heuristics would have prohibitive execution times. In [19], the authors use a reinforcement learning approach to scheduling. That work deals more with the data-movement costs and it does not consider the heterogeneity in the environment.

In [18], the authors consider the problem of scheduling parallel tasks in an environment comprising heterogeneous computing sites (similar to the clusters we model), but they optimize for time-based objectives and do not have the notion of value functions.



(a)



(b)

Figure 5: Percentage of the maximum value bound earned by the different metaheuristics in three different levels of (a) correlation between the starting value of a task and its average execution time and (b) heterogeneity.

7 Conclusions and Future Work

In this work, we quantified the performance of schedulers in terms of the total value they earn from completing tasks (as opposed to typical time-based or fairness-based objectives used in parallel scheduling). Through workload trace-driven simulations, we showed that in oversubscribed heterogeneous environments, our value-based metaheuristics always outperformed the popular scheduling techniques that do not consider value functions nor heterogeneity. Further, our novel concept of place-holders consistently improved the performance of the value-based metaheuristics. It did this by drawing on the benefits of reservations (to prevent lower-value tasks from delaying higher-value tasks) and by overcoming its drawback (of being locked down by previous decisions) by opening those "reserved" slots for future tasks that can earn higher value. We tested the performance of all techniques in a wide variety of environments and showed that Maximum VPR with Place-holders always outperformed all the other metaheuristics and its performance is less sensitive to environmental changes. On average, Maximum VPR with Place-holders earned over 50% higher value than EASY Backfilling and over 100% higher than Conservative Backfilling.

In the future, we would like to expand this work by: (1) modeling uncertainty in the execution time of the tasks, (2) optimizing additional objectives such as the power consumed, (3) considering moldable tasks (for which the scheduler decides how many cores to parallelize it across), and (4) designing metaheuristics that give higher priority to tasks whose value is going to reduce soon.

In summary, the novelty of our work is using value functions in an oversubscribed and heterogeneous computing environment and dynamically mapping parallel tasks to maximize the total value earned. Further, we introduce the novel concept of place-holder tasks in parallel scheduling.

References

[1] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and Sa. Ali. Representing task and machine heterogeneities for heterogeneous computing systems. *Tamkang Journal of Science and Engineering*, 3(3):195–207, Nov. 2000.

[2] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, Jun. 2001.

[3] Colorado State University ISTeC Cray High Performance Computing System. Available: http://istec.colostate.edu/activities/cray, Jan. 2011.

[4] Extreme Science and Engineering Discovery Environment (XSEDE). Available: https://www.xsede.org/, July 2011.

[5] A. Ghafoor and J. Yang. A distributed heterogeneous supercomputing management system. *IEEE Computer*, 26(6):78–86, June 1993.

[6] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on non-identical processors. *Journal of the ACM*, 24(2):280–289, Apr. 1977.

[7] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3):42–51, July 1998.

[8] B. Khemka, R. Friese, L. D. Briceño, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole. Utility functions and resource management in an oversubscribed heterogeneous computing environment. *IEEE Transactions on Computers*, accepted 2014, to appear.

[9] A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang. Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[10] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli. Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. *Journal of Parallel and Distributed Computing*, 67(2):154–169, Feb. 2007.

[11] D. Klusaccek and H. Rudova. Performance and fairness for users in parallel job scheduling. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 235–252. Springer Berlin Heidelberg, 2012.

[12] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *International Symposium on High Performance Distributed Computing (HPDC '07)*, pages 107–116, 2007.

[13] D. A. Lifka. The ANL/IBM SP scheduling systems. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg, 1995.

[14] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–121, Nov. 1999.

[15] A. Mishra, S. Mishra, and D. S. Kushwaha. An improved backfilling algorithm: SJF-B. *International Journal on Recent Trends in Engineering & Technology*, 5(1):78–81, Mar. 2011.

[16] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[17] Parallel Workloads Archive. Available: http://www.cs.huji.ac.il/labs/parallel/workload/, Dec. 2014.

[18] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 87–104. Springer Berlin Heidelberg, 2003.

[19] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos. Adaptive data-aware utility-based scheduling in resource-constrained systems. Technical Report 2007-164, Sun Microsystems, Inc., 2007.

[20] Y. Yuan, Y. Wu, W. Zheng, and K. Li. Guarantee strict fairness and utilize prediction better in parallel job scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):971–981, Apr. 2014.