Scheduling High-Level Tasks Among Cooperative Agents

Bradley J. Clement
University of Michigan
Artificial Intelligence Laboratory
Ann Arbor, MI 48109
bradc@umich.edu

Edmund H. Durfee
University of Michigan
Artificial Intelligence Laboratory
Ann Arbor, MI 48109
durfee@umich.edu

Abstract

Scheduling tasks among cooperative agents requires tradeoffs between various factors including task priorities and context-dependent execution times. We have specifically been investigating the space of functions for evaluating alternative distributed task schedules for multi-operator applications. In this paper, we describe some candidate functions and converge on intuitively appealing functions, which we show to lead to equivalent preferences over distributed schedules. We then look at the computational complexity of finding schedules that (approximately) optimize this function. When context-switching costs are thrown into the mix, moreover, the complexity becomes even more daunting. To address these problems, this paper summarizes our work on forging correspondences between our problems and those studied in operations research. Moreover, we have developed a new hill-climbing strategy for solving these problems, and in this paper we show that these strategies yield very good performance within the range of parameter settings that are representative of our application domain.

1. Introduction

The problems of allocating and scheduling tasks among distributed agents has been a central concern in the multi-agent systems literature for many years. Often, making good allocation and scheduling decisions requires a careful analysis of task needs and the relationships between tasks [5,13]. Moreover, in many domains the priorities of tasks factor into the decisions, as do the costs of switching between dissimilar tasks by an agent.

In particular, some tasks may have different degrees of urgency and/or importance and are to be accomplished by human operators, such as customer phone service operators, operators of nuclear reactors, or personnel in a submarine control room. Specifically, we are looking at the latter application domain; TAIPE (Tactical Assistants for Interaction Planning and Execution) is a system designed for the Navy to help automate ship operation [6]. In TAIPE interaction plans determine the functional components that must be used to manage a display dialogue for operators to process tasks. A crew of operators needs to apply their abilities to process the most important tasks quickly while considering the inherent difficulty of jumping between dissimilar tasks.

In this paper, we focus on strategies for automatically managing the task assignments and schedules among the human operators. Operator task management in TAIPE has, as one of its responsibilities, the need to balance the task processing loads among the operators such that urgent tasks are processed quickly, operators are all utilized, each operator's time is used efficiently, and operator stress is minimized. These different demands can at times be contradictory. For example, using operators efficiently can mean that each operator specializes in a particular subclass of tasks so as to minimize context switching overhead. But this can lead to underutilization (some operators might be idle when there are no tasks of their subclasses to do), poor response to urgent tasks (which might be concentrated at a few operators), and high stress for the operators that have the urgent tasks.

Systems like TAIPE have an unusual combination of properties that set them apart from the kinds of distributed and real-time systems more often studied in computer science. The tasks are of coarse granularity; the message passing costs are negligible relative to the task processing time; the processing elements are complex (humans); the context switching between tasks can require substantial effort; the tasks can have different levels of importance; the tasks should not be preempted (due to context-switching costs); and hard deadline information is not available since tasks are "best effort."

The goal of this paper is to begin to lay out the design space for alternative load balancing strategies; specifically, we are concerned with the design of metrics for evaluating the utility of a particular distributed task schedule for operators, given the priorities and execution times of the tasks. With such metrics in hand, we then need to develop algorithms for forming (approximately) optimal distributed schedules, where we can characterize the complexity of such algorithms. The contributions of this paper, therefore, are in describing a process for articulating the task management objective function in a decentralized application involving sophisticated agents and for designing algorithms that are cost-effective in this complex domain.

In Section 2, we relate how this work fits in with previous research by providing background on related work in distributed computing, operations research, and artificial intelligence. Section 3 proposes a strategy of incorporating priorities and time needs into utility functions and singles out one that is intuitively appealing and flexibly meets the needs of systems like TAIPE. After reviewing the complexity of different variations and simplifications of the scheduling problem associated with the performance evaluation function, Section 4 presents simulation results revealing interesting aspects of the performance based on the search space topology. Section 5 talks about the complexity of adding context-switching into the problem, introduces a heuristic-based hill-climbing algorithm for tractably searching for good solutions in the more complex domain using the proposed performance functions, and describes how it performs in various contexts.

2. Related work

In distributed and real-time systems literature, priorities are assigned to tasks to resolve system resource contention. These resources could be processors, memory, input, etc. Priorities in TAIPE are different in that the execution time of tasks also plays a part in the scheduling of tasks (which resolves processor contention). TAIPE's task priorities represent urgency, but there is no hard deadline information to consider. These priority values are not assigned based on ordering relationships among the tasks but are intrinsic values possibly representing the importance or urgency of tasks, the rewards for processing them, or the costs of not processing them.

Section 3 examines the selection of a function to evaluate the performance of task schedules based on this notion of priority. Recently, the operations research community has produced various approximation algorithms for scheduling that give performance guarantees for different performance evaluation functions. Some of these algorithms guarantee to produce a schedule that has a performance no worse than a constant factor of the optimal schedule's performance [2,4,9]. These kinds of guarantees are helpful in building real-time systems that have hard deadlines. For systems like TAIPE, knowing the expected performance would also be a valuable, so we give results of simulations aimed at finding that information in Section 4.3.

Although these methods for performance evaluation shape strategies for load balancing, there are also issues concerning the actual load balancing algorithms and the architecture of the distributed system. Many studies of load balancing algorithms have centered on particular hardware architectures [1,10,12,14,16]. They usually seek greater performance by reducing communication overhead, which is greatly influenced by the way the processors are connected. These architectures and associated algorithms offer insight into designing scaleable systems of distributed agents for task management for applications like TAIPE.

On a software level, load-balancing algorithms generally adopt one of two approaches. One approach is to optimally balance the load given the current task set; this approach works well if the optimization problem can be solved quickly relative to the evolution of the task set. The other approach is to use suboptimal mechanisms that locally adapt schedules to rapidly-changing task sets [9,21]. Our work tries to move the first approach toward the second. The characteristics of TAIPE, where the arrival rate of tasks is relatively low and the processing time of tasks is relatively large, motivate making (near) optimal distributed schedules, yet there is enough potential task turnover to suggest the use of faster, anytime algorithms.

Context switches between tasks can also be viewed as a hindering relationship between tasks. Task relationships, such as hinders, enables, facilitates, and precedence are computationally described in the TÆMS (Task Analysis, Environment Modeling, and Simulation) framework [5]. TÆMS provides a method for modeling complex computational task environments. A TÆMS model aims to analyze, explain, or predict the performance of a system or some part of it. TÆMS may be useful for representing complex agent and task interrelationships for TAIPE when more information about possible task relationships are available.

3. Load balancing strategies

To design algorithms for forming distributed schedules, we first must ask what defines a good schedule. A good schedule maximizes the performance of the distributed system, but we then must be clear about what "performance" means. The performance of schedules have conventionally been measured in many ways:

- average wait
- throughput

- operator utilization
- turnaround time
- average response time
- schedule completion time (makespan)
- deadlines met
- availability
- reliability

In a system like TAIPE, the response time of tasks is the most important criterion because at critical times the system must react quickly to its environment. In other applications, such as a telephone customer service, average wait may be more important than response time. We expect that issues dealing with performance evaluation in terms of average wait to be similar to those involving response time. Because some tasks are more urgent than others, we would rather minimize the response time of higher priority tasks. Dealing with both of these variables in characterizing the performance of the system is the subject of this section.

3.1. Initial strategy; priority = load

Currently in TAIPE, not only does the priority of a task determine its order in an operator's queue, but it is also used to represent the task's load. If we assume that the priority represents the importance of executing a task, then load balancing actually tries to balance cumulative responsibility or *stress* instead of actual workload in terms of process time requirements. That is, it is possible for priority to be balanced among operators and yet one operator could have substantially more work to do because of the execution times of his/her tasks.

Even ignoring this workload imbalance for the moment, balancing based on priorities does not ensure that the highest priority tasks are addressed first since balancing the cumulative priorities does not ensure that the highest priority tasks are at different operators. If we balance load as the sum of priorities of tasks for each operator (Figure 1a), we see that not only are the highest priority tasks not necessarily processed soonest, but also workload (in terms of processing time) may not be distributed evenly. In Figure 1a, there are two operators, and the boxes are tasks labeled with priorities and having uniform process time requirements. Tasks at the bottom of the queue are executed first, and we assume that priority values are represented as non-negative real numbers.

The priorities of such tasks can be exaggerated to improve the distribution of high priority tasks; this is what happens in the current TAIPE, which actually balances the sum of the *squares* of the priorities. Even so, cases like the one in Figure 1b illustrate that this strategy can also fail both to ensure that the highest priority tasks are addressed soonest as well as to distribute workload.

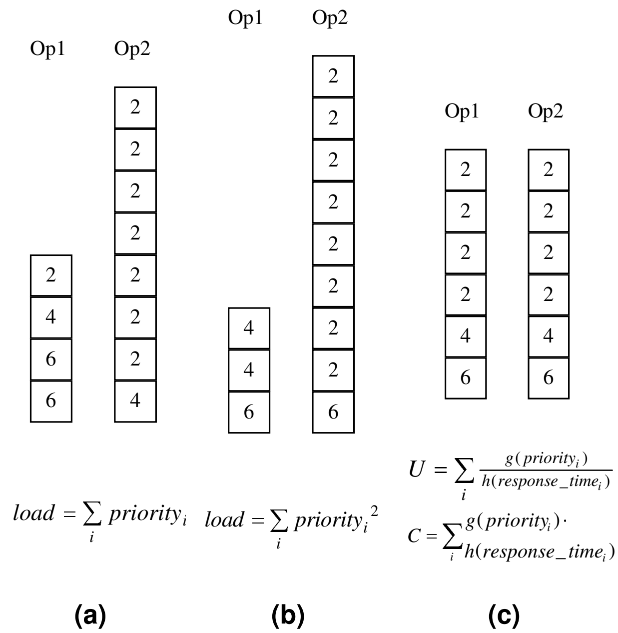


Figure 1. Strategies involving execution time

3.2. Strategies involving execution time

If we wish to minimize response time or average wait on higher priority tasks, we should maximize a utility function that gives greater value to higher priority tasks with smaller response times. The response time for a task is the time it takes to complete its execution from the time it enters the system; it is the sum of its execution time and the execution times of all tasks before it in the queue when it enters the system. For increasing functions g and h, the class of general utility and cost functions given in Figure 1c balance both the stress and work load. Some simple examples we will look at in some detail follow.

$$U = \sum_i \frac{priority_i}{response\_time_i} \quad (1) \qquad U = \sum_i \frac{priority_i^2}{response\_time_i} \quad (2)$$

$$C = \sum_i priority_i \cdot response\_time_i \quad (3)$$

Using any of these utility or cost functions achieves the same purpose of minimizing response time for higher priority tasks, but the meaning of relative priority values is different for each equation and variations of them. This is illustrated by comparing the utility and cost functions described above for the following alternative orderings of tasks in a queue in Figure 2. The first number in each box is the task's priority, the second is the task's execution time.

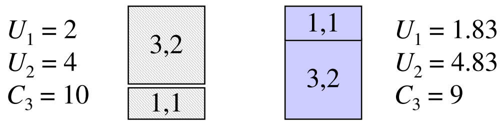


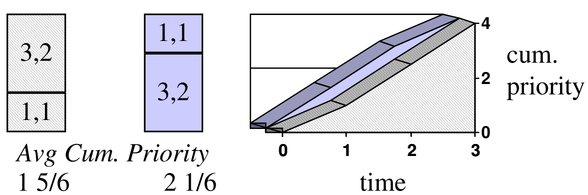
Figure 2. Performance function differences

We see that the left schedule is better for the utility function, Function 1, and the right is better for the other utility function, Function 2, and the cost function, Function 3. This shows that priority assignments have different meanings based on the nature of the function measuring performance. Moreover, if we were to square the priorities, Function 1 will prefer a different schedule. This illustrates the need to codesign the function for schedule evaluation and the function for computing priorities.

3.3. The cumulative priority strategy

<< What is the space of perf. eval. fcns.? Can it be traversed? >>

What TAIPE needs is to process higher priority tasks as soon as possible. We can think of this as trying to accumulate as much task priority value as possible as soon as possible. If we order tasks according to *priority* / *execution_time* ratios, operators will first perform tasks that get the most "bang" (priority value) for the "buck" (time invested).



$$U = \frac{\int cum\_priority(t)dt}{time\_period} \quad (4)$$

Figure 3. Average cumulative priority

In general, we can find the optimal ordering based on this "bang for the buck" notion by maximizing the time-averaged cumulative priority. Figure 3 shows how ordering the tasks for a single operator in decreasing *priority* / *execution_time* ratios maximizes the area under the cumulative priority curve and, thus, the average cumulative priority. Note that this shows a gradual accumulation of priority value over the execution of tasks. The curve could have been plotted as a step function that added the priority value of tasks at their completion. From the summation formula given in Section 3.4 for this function, it is simple to see that the step function has the same properties as the function that gradually accumulates priority value.

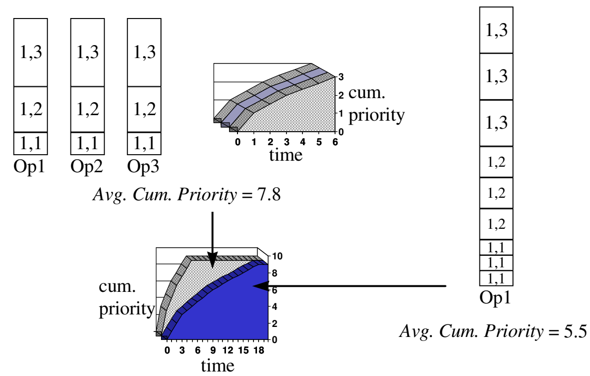


Figure 4. Cumulative priority in a multi-operator schedule

We can also use this measure to optimize a multi-operator schedule. The cumulative priority function now represents the total accumulated priority of the system which is the sum of the cumulative priority functions of all of the operators, and the utility function is simply the time-averaged cumulative priority. Figure 4 shows how this utility function gives preference to distributing load evenly among operators. The left side of Figure 4 shows the situation where nine tasks are spread evenly among three operators. We want to maximize the area under the cumulative priority curve while minimizing the total execution time. The chart at the bottom shows how the cumulative priority curve is constructed from those of the three operators. On the right, we have the same tasks all allocated to a single operator, and its cumulative priority function is also shown in the bottom chart. As we should expect, we see that processing tasks in parallel produces a greater cumulative priority curve. Other properties of this function are explored in the next section.

3.4. Comparing performance functions

We have described different functions which can be used to evaluate the performance of a schedule and improve strategies previously used by TAIPE. The different functions represent different views of the appropriate relations among priority, execution time, and the performance of a schedule. However, although the time-averaged cumulative priority, or "bang for the buck," function (Function 4) and the cost function summing the products of priorities and response times (Function 3) represent two such perspectives, they are, in fact, equivalent. For any two schedules for a set of tasks, they will always agree on which is better. A proof follows:

The time-averaged cumulative priority of a schedule, S, is simply computed as follows:

$$U(S) = \frac{\sum_i \frac{1}{2} p_i {t_{x_i}}^2 + p_i (t_{\max} - t_{e_i})}{t_{\max}}$$

where p_i is the priority, t_{xi} is the execution time, and t_{ei} is the end or finish time (response time for this static case) of i_{th} task. t_{max} is some constant greater than or equal to the sum of the execution times of all of the tasks. We need this constant as the time period over which the cumulative priority is averaged so that schedules can be compared fairly.

So, if we have two schedules, S_1 and S_2, and $U(S_1) > U(S_2)$, then

$$\frac{\sum_i \frac{1}{2} p_i {t_{x_i}}^2 + p_i (t_{\max} - t_{1e_i})}{t_{\max}} > \frac{\sum_i \frac{1}{2} p_i {t_{x_i}}^2 + p_i (t_{\max} - t_{2e_i})}{t_{\max}}$$

$$\sum_i p_i t_{1e_i} < \sum_i p_i t_{2e_i}$$

But these terms in the above inequality are just sums of products of priorities and response times of the two schedules, which is the cost function, Function 3. So, we have $C(S_1) < C(S_2)$. Thus, if the "bang for the buck" function prefers one schedule to another, the cost function will prefer it, too--the functions are equivalent. ■

3.5. Schedule evaluation function summary

In summary, by considering the relevant factors for schedule evaluation in TAIPE, we have been able to explore the space of alternative evaluation functions. Clearly, the TAIPE starting point is inappropriate. Moreover, we can enumerate some intuitively-appealing functions, and show that the choice between them is moot, in that they represent consistent preferences over schedules.

4. Complexity of schedule optimization

Now that we have identified a good measure for comparing alternative distributed schedules, we have to determine whether we can use this measure fruitfully within our application. In particular, we have to assess the computational complexity of finding schedules that satisfy the preferences.

Unfortunately, in general computing an optimal distributed schedule based on this measure is very costly. For example, the cost function is known in operations research as "weighted average completion time" or "total weighted completion time." The corresponding decision problem, Scheduling to Minimize Weighted Completion Time, is strongly NP-complete for an arbitrary number of machines and is pseudopolynomial for a fixed number of machines [7]. What this means is that we need to consider techniques for simplifying or approximating the calculations.

4.1. Simplifying strategies

Several simplifications of the problem ignore factors that introduce complexity. These factors include the distribution across multiple operators, the non-uniform priorities, and the non-uniform execution times. If we can relax one of these, we can generate simpler strategies for generating optimal schedules.

If we assume there is no distribution, the problem becomes very easy. For the single machine case, schedules are optimized by applying Smith's Rule [15], which orders tasks according to *priority / execution_time* ratios.

If we assume uniform priorities, computing a distributed schedule is similarly easy. Since the utility function and the cost function are equivalent, when two schedules are compared, the priority variable, p_i drops out, and only the sums of response times are compared. Thus, the best schedule minimizes the average response time. The shortest-job-first (SJF) strategy is known to optimally minimize average response time, so the "bang for the buck" strategy becomes SJF with this simplification. The optimal policy is to order all tasks according to ascending processing times and then to deal them out to the operators such that the next task is assigned to the operator assigned the shortest total processing time. [3].

If we assume uniform execution times, the best schedule never has a higher priority task begin after a lower priority task. This means that tasks can simply be sorted by their priority and dealt out to the next available operator to achieve the best performance; we will refer to this as the greatest priority first (GPF) rule. To prove this, if we assume that this property does not hold, then there must be an optimal schedule with tasks j and k that have priorities p_j and p_k and response times t_{ej} and t_{ek} such that $p_j > p_k$ and $t_{ej} > t_{ek}$. The contribution of these two tasks to the cost function would be $p_j t_{ej} + p_k t_{ek}$. If we swap the positions of the tasks in the schedule, the contributions of any other tasks in the schedule to the cost will not the change because the time slots are all the same, and the contribution of tasks j and k would be $p_j t_{ek} + p_k t_{ej}$. Since the former schedule is optimal, its cost must be smaller than the latter's, which means

$$p_j t_{ej} + p_k t_{ek} < p_j t_{ek} + p_k t_{ej}$$

$$p_j(t_{ej} - t_{ek}) < p_k(t_{ej} - t_{ek})$$
$$p_j < p_k$$

But this contradicts the assumption, so it is true that if the execution times are fixed, the tasks will execute in order of priority across all queues. ■

4.2. Approximation techniques

Clearly, one approximation would be to try to use simple algorithms like TAIPE currently uses, where we can just order based on a value that can be measured for each task. As opposed to original TAIPE's use of *priority*^{2}, we want to include response time. Unfortunately, this will not work for the *priority / response_time* and *priority*^{2} */ response_time* utility functions (Functions 1 and 2); Smith's rule does not apply to them. In Figure 2, we see an exception for Function 1, and a simple example for Function 2 can be seen in Figure 5 below. Although the queue on the right is ordered in decreasing *priority / execution_time* ratios, the left ordering is optimal for Function 2.

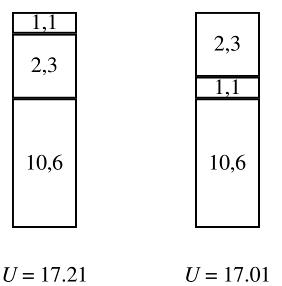


Figure 5. Function 2 and Smith's rule

Alternatively, we can try to apply Smith's Rule across operator task queues for the time-averaged cumulative priority function. That is, we could try to construct optimal schedules by ordering all of the tasks in decreasing *priority / execution_time* ratios and deal them out to the most underloaded operator. However, it is easy to find cases where this approximation fails to match an optimal allocation; an exception is shown in Figure 6:

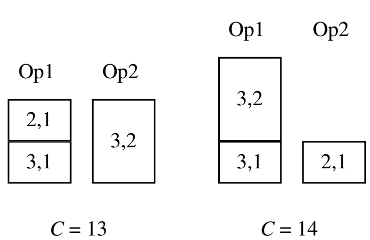


Figure 6. Smith's rule in the multi-operator case

The right side gives an allocation that ensures that the task with the highest *priority / execution_time* ratio is always being processed; however, the optimal allocation starts a task with a ratio of 3/2 before another with a ratio of 2. Notice that the optimal configuration increases operator utilization while the other puts a higher priority task behind a lower priority task.

Fortunately, recent work in the literature has been devoted to developing approximation strategies for variations of this problem [2,4,9]. An α-approximation algorithm guarantees that the performance will be a factor α of the optimal solution. In Table 1, we summarize the space of algorithms for solving variations of the distributed scheduling problem. The $\alpha|\beta|\gamma$ representations of the scheduling problems come from operations research. α describes the number of machines, and P corresponds to m identical machines. β is a set of constraints and/or relaxations such as precedence, preemption, or release dates. γ is the type of optimization, which in this case is minimizing weighted completion time.

	Performance Evaluation Function	*m*-Machs.	Single Machine	No Priorities
1	$U = \sum_i \frac{priority_i}{response\_time_i}$	?	?	SJF O($n$log$n$) [3]
2	$U = \sum_i \frac{priority_i^2}{response\_time_i}$	?	?	SJF O($n$log$n$) [3]
3	$C = \sum_i pri_i \cdot resp\_time_i$ $U = \frac{\int cum\_priority(t)dt}{time\_period}$ $P\vert\vert\sum_j w_j C_j$	NPC [7] $\alpha = 1.2$ [11]	$1\vert\vert\sum_j w_j C_j$ Smith's Rule O($n$log$n$) [15]	$P\vert\vert\sum_j C_j$ SJF O($n$log$n$) [3]
	Performance Evaluation Function	No Exec. Times	Precedence Constrs.	Context Switch
1	$U = \sum_i \frac{priority_i}{response\_time_i}$	GPF O($n$log$n$)	?	NP-hard [2]
2	$U = \sum_i \frac{priority_i^2}{response\_time_i}$	GPF O($n$log$n$)	?	NP-hard [2]
3	$C = \sum_i pri_i \cdot resp\_time_i$ $U = \frac{\int cum\_priority(t)dt}{time\_period}$ $P\vert\vert\sum_j w_j C_j$	$P\vert p_j = 1\vert\sum_j w_j C_j$ GPF O($n$log$n$)	$P\vert prec\vert\sum_j w_j C_j$ NPC [9] $\alpha = 1.58$ [4]	NP-hard [2]

Table 1. Complexity of scheduling problems

<< Check to see if table and explanation is clear >>

In addition, recent results for approximation algorithms are given. An α-approximation algorithm guarantees that the performance will be a factor α of the optimal solution. In recent years, many constant factor approximations for schedule optimization (such as those in the table) have been discovered [2,4,9]. We also wish to assert that introducing context switching adds another level of intractability to the problem. In Section 5, we discuss the added complexity of context switching.

The cumulative priority strategy appears to be much better studied than the other candidates we have investigated. However, the corresponding problem is yet still NP-complete, so suboptimal algorithms are expected be more cost-effective for real-time systems. The approximation algorithm referenced in Table 1 ensures a solution within a factor of $(1+\sqrt{2})/2$ or 1.2 of the optimal cost by ordering the entire set of tasks according to Smith's Rule and then applying a list scheduling algorithm. However, getting a better sense of the topology of the search space could tell us if better performance can be expected from informed search methods such as hill-climbing. We next present simulation results that give insight on the expected performance best-first search.

4.3. A hill-climbing strategy

<< Is this section clear? >>

So, we know we can efficiently minimize the weighted completion time cost function (Function 3) to within twenty percent of the optimal using the approximation algorithm of the preceding section. Can we expect to do much better than this? Best-first search methods, such as hill-climbing, offer simple mechanisms to quickly improve on solutions and can be flexibly used in conjunction with other algorithms. The experimental results given here reveal the expected performance of using such methods.

The simulation results given below were collected over a randomly generated sets of tasks to which hill-climbing was performed on all equally likely m^{n-1} assignments to operators to completely explore the terrain of the search space. The hill-climbing operations, pair-wise task swaps and single-task migration, were performed on a random allocation of tasks to operators, and the tasks were ordered within operators' queues according to Smith's Rule. Several parameters were varied (within bounds characteristic of TAIPE and computationally feasible) to see their effects on the performance:

- number of operators 2 . . 5
- number of tasks 4 . . 11
- range of priority values 1-3; 1-10; 1-1000
- range of processing times 1-3; 1-10; 1-1000

For each permutation of these parameter values, ten schedule instances were generated and simulated.

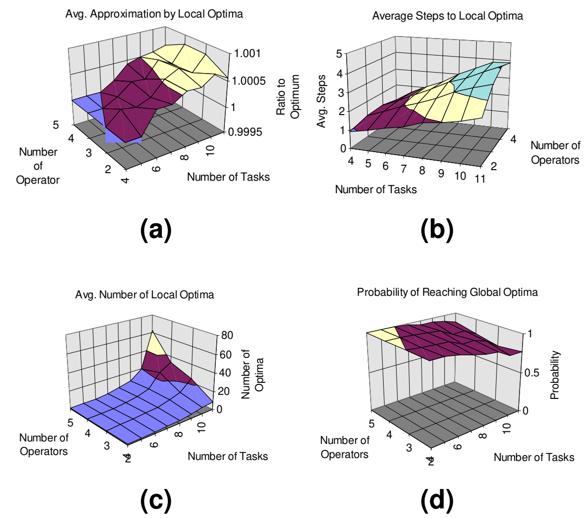


Figure 7. Performance based on number of operators and tasks

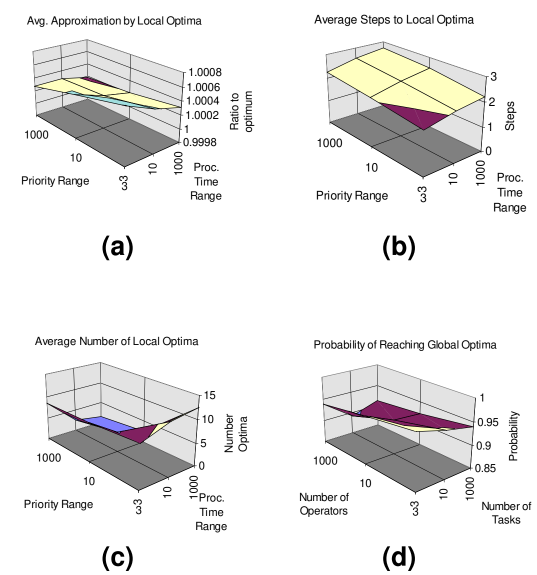


Figure 8. Performance based on the range of priority values and processing times

<< Can we make graphs more understandable by changing angle of view? >>

Figure 7 shows how the performance varies with the number of operators and tasks within the system, and Figure 8 varies the range of priority values and processing times. The average approximation by the local optima in Figures 7a and 8a reveal that hill-climbing performs within one percent of the optimal for our test set; however, the approximation seems to slowly diverge from the optimal for greater parameter values. It is interesting to note that performance is better for fewer operators and larger ranges of priorities and processing times. Figures 7b and 8b show how the number of hill-climbing steps increase with respect to the parameters. We noted that the worst case number of steps was 12. Figures 7c and 8c show the number of wells (or peaks) in the search space. In correlation with Figures 7a and 8a, the complexity is greater for larger numbers of tasks and operators and smaller ranges of priorities and processing times. Figure 7d shows how more operators makes load balancing easier, and Figure 8d seems to indicate that there is a better chance of finding a global optimum when priority and processing time ranges are small even though the expected performance (in Figure 8a) is worse.

This experiment suggests that other suboptimal search methods, such as simulated annealing or tabu search, can be expected to perform very well within bounded ranges of the parameters given here. We also know that we can bound this divergence if we use these search methods in conjunction with the 1.2-approximation algorithm. We can expect that good local optima can be found within a few hill-climbing steps, and hill-climbing can easily be adapted to use dynamic information, such as new task arrivals or changing priority and processing time values. This bodes well for systems like TAIPE that can use the "bang for the buck" performance measure where urgent situations require quick adaptation.

5. Using clustering algorithms for minimizing context switching

In Sections 3 and 4, we identified the time-averaged cumulative priority function as a fitting utility function for evaluating the performance of a schedule. Smith's Rule allows the search algorithm to concentrate on the allocation of tasks and ignore the ordering of tasks within a queue when context switching and other task dependencies are insignificant. Here we reintroduce context switching into the scheduling problem. We will give a brief analysis of the complexity incurred by context switching and relate how search algorithms can take advantage of dividing the search space to deal with the complexity added by context switches between tasks. We follow with results from a simulation to show how well one search algorithm can perform within the more complex system and comment on strategies multi-agent systems can apply to adapt to different environment conditions.

5.1. Reducing context switches through clustering

In Section 1, we discussed how context switches can affect the performance of a system. Context switches between tasks depend on both the order of execution of tasks within a queue and the allocation of tasks among the queues of multiple operators. An optimal algorithm must find the best allocation based on the best ordering for each queue. A sub-optimal algorithm, however, can deal with these two operations separately.

There may be much simpler, faster algorithms for handling static situations where the arrival of new tasks is ignored. It is thought that in short unusual bursts of task arrivals, the queues will be unusually long, and static algorithms could achieve better efficiency. In addition, the coarse granularity of the tasks allow more time for the algorithm to produce a solution before system changes make the information obsolete. For these reasons, we feel that static algorithms are appropriate for this system.

We assume that the context switch time between two tasks can be estimated by comparing their attributes. Tasks that have similar or matching attributes will have a shorter context switch time than tasks with very different properties. Thus, if similar tasks are allocated together, we should expect smaller average context switch times. This heuristic can take advantage of clustering algorithms whose purpose is to group similar things.

A clustering algorithm may have a measurement that either computes the similarity or dissimilarity of a group of tasks. For two tasks, this may be represented as a weighted sum of matches (or mismatches) of the values of the attributes. For a cluster of tasks, we can sum these matches over pairs of tasks and normalize them by dividing by the number of pairs, $n(n-1)/2$. We assert that the context switch time for two tasks is directly proportional to their dissimilarity, or incoherence. The incoherence of a cluster is equivalent to the average incoherence of all pairs of tasks, which is calculated as a weighted average percentage of mismatched attribute values. Because we only look at whether attribute values match or mismatch in the above computation, we assume that any pair of different values for an attribute will require the same amount of context switching.

Minimizing the cost function (Function 3) for the cumulative priority strategy for the case where context switches are computed in proportion to the number of mismatched attribute values is related to the NP-hard *minimum latency problem* (MLT), also referred to as the *delivery man problem* and/or the *traveling repairman problem* [2]. This is the problem of finding a tour that minimizes the sum of the *latencies* (distances from the starting point) of all points. There is a constant factor 29-approximation for MLT in the case of a distance matrix adhering to the triangle inequality[2]. The context switching problem is the same in the relaxed case of a single operator and equal processing times for tasks. Context switches between tasks are distances between points that adhere to the triangle inequality—no two tasks can have more mismatches than the sum of the number of mismatches they each have with any third task. In considering one very simplified aspect of this scheduling problem, we see intractability, so we can easily imagine that the scheduling problem without these relaxations must be at least as hard. Now we look at a clustering strategy that must deal with this anticipated complexity.

5.2. Heuristic-based hill climbing

This algorithm has been implemented in TAIPE to support two different load balancing schemes. One of these is for single operators to locally search for a good cluster of tasks that meet a load requirement and are to be shed to another operator. This method allows task management to be handled decentrally. The other scheme is centralized, and our heuristic-based hill-climbing mechanism searches through the space of task reallocations among two or more operators in order to improve the coherence of clusters at each operator and to balance load. The algorithm relies on a distance function, D_f below, for improving clusters and addressing load. For the multi-operator scheme, we could substitute the cost function (Function 3) to implement a cumulative priority strategy. The basic idea is to restart hill-climbing at places in the search space that could lead to the optimal solution. For this algorithm, these places are initial clusters of tasks constructed for each attribute as the largest set of tasks sharing the same value. From these initial clusters, tasks are added, swapped, and removed until a local optimum is found. The best of the local optima found is chosen as the solution. Here is a more detailed description of the algorithm:

```
•  compute the distribution of values for each
   attribute for all tasks in list
•  find the value(s) for each attribute that
   has the greatest distribution of tasks
•  construct candidate initial clusters of
   tasks for each value with greatest
   distribution for attribute
•  remove duplicate candidate clusters
•  BestLocalOptimum = {}
•  for each cluster := candidate_cluster
   •  done := false
   •  while not done
      •  if Df(modified_cluster) ≥ Df(cluster)
         •  LocalOptFound := true
      •  cluster := modified_cluster
      •  best := cluster
      •  for each task ∈ TaskList – cluster
         •  // Try adding task
         •  if Df(cluster + task) > Df(best)
            •  best := cluster + task
         •  // Try swap——swap original
            candidate cluster tasks iff to
            aspirate at local optimum
         •  if LocalOptFound swap_set =
            candidate_cluster
         •  else swap_set = cluster –
            candidate_cluster
         •  for each c-task ∈ swap_set
            •  if Df(cluster + task – c-task)
               > Df(best)
               •  best := cluster + task –
                  c-task
      •  // Try remove——remove original
         candidate cluster tasks iff to
         aspirate at local optimum
      •  if LocalOptFound remove_set =
         candidate_cluster
      •  else remove_set = cluster –
         candidate_cluster
      •  for each c-task ∈ remove_set
         •  if Df(cluster – c-task) > Df(best)
            •  best := cluster – c-task
      •  modified_cluster = best
      •  if LocalOptFound AND
         Df(modified_cluster) ≥ Df(cluster)
         •  done = true
   •  if Df(BestLocalOptimum) > Df(cluster)
      •  BestLocalOptimum := cluster
•  solution = BestLocalOptimum
```

The algorithm does not allow any changes to the original tasks in the initial candidate cluster until a local optimum is found and then "aspirates" a better solution if one exists by removing or swapping the original tasks. The idea stems from Tabu Search where tabu moves may be taken if they exceed an aspiration level[8].

5.3 Performance with Context Switching

The described algorithm was simulated on data having the same characteristics as those of the experiment outlined in Section 4.3 but with the following additional varied parameters:

- number of task attributes 3, 5, 9
- number of values per attribute 3, 10
- max context switch 0.1, 0.5 (factor of proc. time)

As expected, the performance of the hill-climbing algorithm grows worse with greater numbers of operators and tasks when faced with context switching (Figure 9). In contrast with the experiment in Section 4, the number of operators does not make the problem simpler.

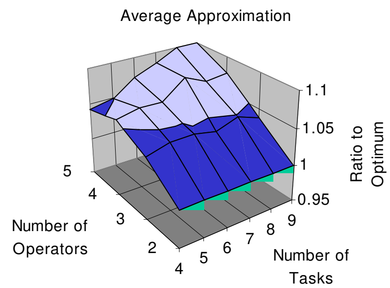
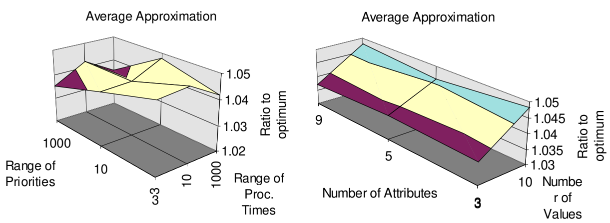


Figure 9. Performance based on number of operators and tasks



(a) **(b)**

Figure 10. Peformance based on number of attributes and attribute values

Figure 10a shows a slight increase in performance for larger ranges of priorities and processing times. Figure 10b reveals that the number of values per attribute greatly affects the performance while the number of attributes

6. Conclusion

In this report, we have looked at the problem of task scheduling and load balancing for a particular multi-agent system. The main contribution here is a function suitable for evaluating the performance of a schedule for such a system. In addition, a link has been made to the complexity analysis of proposed alternative algorithms in the operations research community. Furthermore, suggestions have been made for choosing search algorithms that can efficiently find solutions that are valued by this performance evaluation function. We have analyzed the computational needs of this problem, and identified simplifications and approximations that show promise. In particular, besides making new connections to the operations research literature, we have also examined the use of heuristic hill-climbing methods. For the parameter space manifested in TAIPE, our results indicate that using these methods provides a promising avenue for managing the task loads and schedules in the distributed system.

Further work is needed to extend the development of the strategy for evaluating allocation and scheduling decisions to accommodate time-dependent priority functions and the incorporation of hard deadlines. More work should also be directed towards formulating and evaluating centralized and decentralized architectures for performing the task management functions, and incorporating them into systems like TAIPE. The final objective of this research is to use information about how the dynamics of the system fluctuate in order to optimize performance by switching between centralized and decentralized coordination techniques and different load balancing algorithms to best handle the fluctuation of system dynamics.

References

[1] Ishfaq Ahmad, Arif Ghafoor. SemiDistributed Load Balancing for Massively Parallel Multicomputer Systems. *IEEE Transactions on Software Engineering*, 17(10):987-1004, October 1991.

[2] Avrim Blum, Prasad Chalasani, Don Coppersmith, Bill Pulleyblank, Prabhakar Raghavan, Madhu Sudan. The Minimum Latency Problem. *Proceedings of the 26th Symposium on the Theory of Computing*, pp. 163-171, 1994.

[3] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag, Berlin, 1995.

[4] C. Chekuri, R. Motwani, B. Natarajan, C. Stein. Approximation Techniques for Average Completion Time Scheduling. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms.* pp. 609-618, 1997.

[5] Keith Decker, Victor Lesser. Quantitative Modeling of Complex Environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4), 1994.

[6] Edmund Durfee, Jaeho Lee, Marcus Huber, Michael Kurnow. TAIPE: Tactical Assistants for Interaction Planning and Execution. *Proceedings of the International Conference on Autonomous Agents*, pp.443-450, 1997.

[7] Michael R. Garey, David S. Johnson. *A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[8] Fred Glover. Tabu Search—Part I, *ORSA Journal of Computing*, 1(3):190-206, 1989.

[9] Leslie A. Hall, Andreas S. Schultz, David B. Shmoys, Joel Wein. Scheduling to Minimize Average Completion Time: Off-line and On-Line Approximation Algorithms, *http://brutus.mts.jhu.edu/~leslie/#papers*. revised March 1997.

[10] Reinhard Hanxleden, L. Ridgway Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, 13:312-324, 1991.

[11] Tsuyoshi Kawaguchi, Seiki Kyan. Worst Case Bound of an (LRF) Schedule for the Mean-Weighted Flow-Time Problem. *Siam Journal on Computing*. 15(4):1106-1118, 1986.

[12] J. Mohan Kumar, L. M. Patnaik, A. Das. Load balancing algorithms for an extended hypercube. *IEE Proceedings—Comput. Digit. Tech.*, 5:298-306, September 1994.

[13] Jyi-Shane Liu, Katia Sycara. Multiagent Coordination in Tightly Coupled Task Scheduling. *Proceedings - Second International Conference on Multi-Agent Systems*, pp. 181-188, 1996.

[14] Ashwani Ramani, Pradip Chande, Pramod Sharma. A General Model for Performance Investigations of Priority Based Multiprocessor System. *IEEE Transactions on Computers*, 41(6):747-754, June 1992.

[15] W. Smith. Various Optimizers for Single-Stage Production. *Naval Res. Logist. Quart.* 3:59-66, 1956.

[16] Marc Willebeek-LeMair, Anthony Reeves. Strategies for Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979-993, September 1993.

[17] Hua Wu, David Chang, William Oldham. Dynamic Task Allocation Strategies. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1301-1306, December 1995.