Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications

Barbara Kreaseck[1] Larry Carter[1] Henri Casanova [1,2] Jeanne Ferrante[1]

[1] Department of Computer Science and Engineering [2] San Diego Supercomputer Center
University of California at San Diego
{kreaseck, carter, casanova, ferrante}@cs.ucsd.edu

Abstract

In this paper we investigate protocols for scheduling applications that consist of large numbers of identical, independent tasks on large-scale computing platforms. By imposing a tree structure on an overlay network of computing nodes, our previous work showed that it is possible to compute the schedule which leads to the optimal steady-state task completion rate. However, implementing this optimal schedule in practice, without prohibitive global coordination of all the computing nodes or unlimited buffers, remained an open question. To address this question, in this paper we develop autonomous *scheduling protocols, i.e. distributed scheduling algorithms by which each node makes scheduling decisions based solely on locally available information. Our protocols have two variants: with non-interruptible and with interruptible communications. Further, we evaluate both protocols using simulations on randomly generated trees. We show that the non-interruptible communication version may need a prohibitive number of buffers at each node. However, our autonomous protocol with interruptible communication and only 3 buffers per node reaches the optimal steady-state performance in over 99.5% of our simulations. The autonomous scheduling approach is inherently scalable and adaptable, and thus ideally suited to currently emerging computing platforms. In particular this work has direct impact on the deployment of large applications on Grid, and peer-to-peer computing platforms.*

1. Introduction

Advances in network and middleware technologies have brought computing with many widely-distributed and heterogeneous resources to the forefront, both in the context of *Grid Computing* [14, 15] and of *Internet Computing* [32, 12, 37]. These large distributed platforms allow scientists to solve problems at unprecedented scales and/or at greatly reduced cost.

Application domains that can readily benefit from such platforms are many; they include computational neuroscience [34], factoring large numbers [11], genomics [38], volume rendering [31], protein docking [24], or even searching for extra-terrestrial life [32]. Indeed, these applications are characterized by large numbers of independent tasks, which makes it possible to deploy them on distributed platforms despite high network latencies. More specifically, in this paper we assume that all application data initially resides in a single repository, and that the time required to transfer that data is a significant factor. Efficiently managing the resulting computation is a difficult and challenging problem, given the heterogeneous attributes of the underlying components. This problem is well recognized [3, 18, 21, 16, 13, 26, 36, 23, 7, 2], and there is a large body of applied research, see for instance [5, 10, 1, 17, 33], providing various practical approaches toward a solution. An added complexity is that resources in these environments exhibit dynamic performance characteristics and availability, and a number of the aforementioned works have taken first steps toward addressing this issue.

Our previous work [3] differs in that we employ a *hierarchical* model of the computing platform. The advantage of such a model is that *each* component can make *autonomous*, local scheduling decisions, without costly global communication or synchronization. The processor controlling the data repository decides how many tasks to execute and how many to send to other processors; these processors in turn decide whether to execute tasks or to send them down the hierarchy, and so on. This approach allows for adaptivity and scalability, since decisions can be made locally. It is particularly effective for scheduling in environments that are heterogeneous and dynamic, such as global and peer-to-peer computing platforms consisting mostly of home PC's, as well as emerging Grid platforms based on

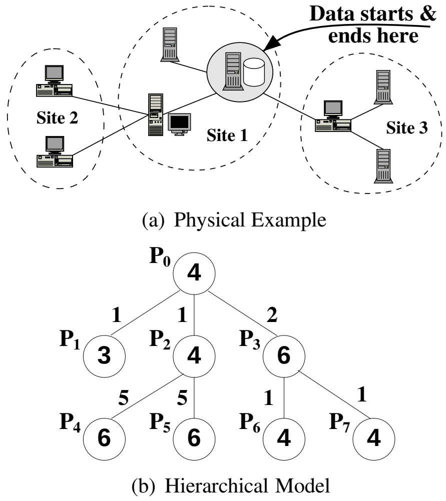


Figure 1. Underlying grid and tree overlay.

distributed Web services.

We showed in [3] that there is a simple algorithm for determining the maximum steady-state task execution rate in our hierarchical model. While we established a provably optimal bound on steady-state execution under idealized assumptions, we did not specify how to achieve this maximum steady-state rate in practice. In this paper, we present two autonomous bandwidth-centric scheduling protocols that address the practical problem of attaining the maximum steady-state rate after some startup and maintaining that rate until wind-down. In simulation, one of these protocols achieves the *provably optimal* steady-state throughput in over 99.5% of our test cases, and thus near-optimal overall execution times.

In Section 2 we review the scheduling principle that forms the basis for our protocols. In Section 3 we discuss the details of our protocols. We present our experimental results in Section 4 and related work in Section 5. Finally, in Section 6 we discuss future work and conclude.

2. Background

2.1. The Bandwidth-centric Principle

In all that follows we target applications that consist of large numbers of *independent* and *identical* tasks. That is, application tasks have no ordering or synchronization, and the computational cost and amount of input/output data required for each task is identical. We model a heterogeneous platform as a tree in which the nodes correspond to compute resources and edges to network connections (see Figure 1). We use weights for nodes and edges to denote computation and communication costs, respectively. The work in [3] also considered several other models for compute node capabilities. In this work we use only the "base model", i.e. a node can simultaneously receive data from its parent, send data to one of its children, and compute a task. A number of relevant models were shown to reduce to the base model. Finally, each node has a number of local buffers that can be used to hold data for one application task.

Somewhat to our surprise, since almost all scheduling problems are NP-complete, we were able to show in [3] that there is a simple algorithm for determining the maximum steady-state task execution rate. Consequently, we can create a schedule that can process a fixed number of tasks within an additive constant of the optimal schedule.

Assume each node has a finite number of buffers to hold tasks that it receives from its parent in the tree. Here is an intuitive description of the scheduling algorithm in [3]:

> Each parent node prioritizes its children according to the time it takes the node to communicate a task to the child. Each parent delegates the next task in its buffers to the highest-priority child that has an empty buffer to receive it.

We call a scheduling algorithm that follows this principle *bandwidth-centric* because the priorities do not depend on the descendent's' computation capabilities, only on their communication capabilities. (Of course, the computation time will affect how frequently a descendent's buffers become available.) We showed in [3] that if each node in the tree has *sufficient* buffers, then bandwidth-centric scheduling results in the optimal steady-state (i.e. asymptotic) execution rate.

Figure 1(a) illustrates a model of a small heterogeneous system spanning resources at three sites. The root node is the source and sink of all application data and tasks. It is connected to other nodes at site 1, one of which is connected to nodes in site 2. The root node also has a connection to a node at site 3, which is connected to two other nodes at that site. In effect, this is a tree *overlay network* that is built on top of the physical network topology among these hosts. It is possible to configure the tree differently, and some configurations are certainly better than others. We leave the question of which tree is the most effective for future work.

The formal tree model is a node-weighted, edge-weighted tree $T = (V, E, w, c)$, as shown in Figure 1(b). Since we assume that the target application is a collection of independent fixed-size tasks, we can let the node weight w_i denote the computation time of a single task at node i. Similarly, the edge weight c_i denotes the sum of the time to send a task's data to node i from its parent and to return the results. Larger values of w_i indicate slower nodes and larger values of c_i indicate slower communication. We use a single communication weight since for steady-state execution, it does not matter what fraction of time is spent sending the task's input data and what fraction of time is spent returning the output. We will return to this model later.

Often, the complete schedule for an application consists of a *startup* interval where some nodes are not yet running at "full speed", then a periodic *steady-state* interval where b tasks are executed every t time units, and finally a *wind-down* interval where some but not all nodes are finished. We can determine the maximum *steady-state* execution rate $R = \frac{b}{t}$ that can be sustained on the tree. Equivalently, we can minimize $w_{tree} = \frac{t}{b}$, the computational weight of the tree. If enough bandwidth is available, then the optimal steady-state schedule is to keep all nodes busy. If bandwidth is limited, then the *bandwidth-centric* scheduling strategy, where tasks are allocated only to the children which have sufficiently small communication times, is optimal.

We now formally state the result on which [3] is based. Consider a single-level *fork graph*, consisting of a node P_0 and children $P_1...P_k$, with time c_i needed to communicate a task to child P_i, and time c_0 for P_0 to receive a task from its parent. Then:

Theorem 1 *With the above notation, the minimal value for w_{tree} is obtained as follows:*

1. *Sort the k children by increasing communication times. Renumber them so that $c_1 \leq c_2 \leq ... \leq c_k$.*
2. *Let p be the largest index so that $\sum_{i=1}^{p} \frac{c_i}{w_i} \leq 1$. If $p < k$ let $\epsilon = 1 - \sum_{i=1}^{p} \frac{c_i}{w_i}$, otherwise let $\epsilon = 0$.*
3. *Then*

$$w_{tree} = \max(c_0, \frac{1}{\frac{1}{w_0} + \sum_{i=1}^{p} \frac{1}{w_i} + \frac{\epsilon}{c_{p+1}}})$$

Intuitively, this theorem says that for maximum steady state, a node cannot consume more tasks than it receives from its parent, hence the first term, c_0, of the maximum. For the second term, the first p children can be fed fast enough by the parent to be kept completely busy, but when $k > p$, those children with slower communication rates will either partially or totally starve, independent of their execution speeds.

This theorem allows us to generate the optimal steady-state scheduling policy for all nodes in a heterogeneous tree. A bottom-up traversal of the tree determines the rate (or equivalently, the computational weight w_{tree}) at which tasks can be processed by each subtree of the full tree.

2.2. Practical Limitations

There are two major practical problems with using Theorem 1:

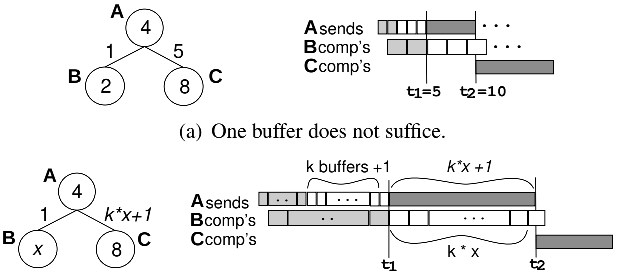
1. The proof of the theorem assumes that each node has enough buffers to queue the tasks that it has received but not yet processed. The number of buffers can be bounded by the least common multiple of all the node and edge weights of the entire tree. However, this bound is very large in practice and can lead to prohibitive startup and wind-down times.
2. It is necessary to know all the c_i's and w_i's in the entire tree to derive the schedule. Although it might be possible to make complete information available in some Grid platforms, doing so for dynamic and decentralized platforms such as global and peer-to-peer computing systems [25, 12] introduces significant complications of collecting and distributing the information, and maintaining consistency in a dynamically changing system.

The autonomous scheduling protocols we present in the next section address both of these limitations and achieve near optimal behavior in our experiments.

3. Autonomous Protocols

Our goal is to design a distributed scheduling algorithm that achieves the maximum task execution rate (according to the bandwidth-centric theorem) while using only "local information". In other words, scheduling decisions at each node are based only on information that is directly measurable at that node. Each node can measure the time it takes to communicate a task to each of its children, the time it takes to compute a task by itself, and the time it takes for each child node to have an empty buffer. In the bandwidth-centric scheduling protocol we present in this section, each node determines whether a received task will be computed by the node itself and if not, which child's subtree will compute the task. In our protocols, a child requests a task from a node when the child has an empty buffer. Each node can then schedule each received task using only directly measurable information. We term such a distributed scheduling protocol *autonomous*.

There are three main advantages to autonomous scheduling protocols. First, they can adapt naturally to *dynamically changing* distributed computing environments. Each node can change its scheduling to adapt to a change in its view of the world (e.g. a slower communication time to one child than to another). Second, they should be able to adapt without costly global communication and synchronization. Third, they promote scalability. Indeed, since all nodes use the same scheduling with no dependence on global information, it is very straightforward to add subtrees of nodes below any currently connected node. In other words, the tree overlay network can grow and reconfigure itself dynamically, which is a highly desirable property in emerging computing platforms.



(a) One buffer does not suffice.

(b) k buffers do not suffice.

Figure 2. Case studies to illustrate the buffer growth problem.

3.1. Non-interruptible communication

Our first approach for designing our autonomous protocol was to use what we call *non-interruptible communication*: once a parent node starts communicating a task to a child, it will finish that communication without any interruptions or pauses regardless of any arriving requests from other children. This is a traditional assumption in most previous work. We found that with non-interruptible communication:

1. One buffer per node does not suffice. In Figure 2(a), the highest priority node, B, should be kept busy processing. B takes 2 time units to compute a task and would need to have at least 3 buffered tasks in order to maintain its rate while node A is sending to node C for 5 time units.

2. For every positive integer *k*, there is a tree such that there is at least one node needing more than *k* buffers to reach optimal steady-state throughput. Consider a more general example in Figure 2(b), where node B takes x time units to compute a task and node A takes $k * x + 1$ time units to send a task to node C. Assuming $x > 1$, node B would need at least $k + 1$ buffered tasks to maintain its rate while node A is sending to node C.

3. For every node in a tree, there is a maximum number of buffers, m_i, needed in order to reach optimal steady-state throughput. The maximum number of buffers needed by a tree is m, where $m = \text{MAX}(m_i)$. In Figure 2(a), *m* is 3, whereas in Figure 2(b) *m* is $k+1$. Since *m* is not known *a priori* or may change over time for a dynamic platform, a correct protocol must allow for buffer growth and, optimally, buffer decay.

4. With unlimited buffers, a flawed protocol may not reach optimal steady-state throughput because it requests *too many* tasks from its parent. For example, if a child node is able to grow more buffers than it minimally needs, that child may be sent more tasks than necessary to maintain optimal steady state, thus robbing its siblings and possibly greatly extending the overall execution time.

Clearly, with non-interruptible communication, a bandwidth-centric protocol using a fixed number of buffers will not reach optimal steady- state throughput in all trees.

Consequently, in the case of non-interruptible communication, our autonomous protocol provides for a variable number of buffers per node. Thus, our protocol makes no assumption about the value of m_i and buffer growth is as follows. Initially, each node has a single, empty buffer. At the start of the application, each node makes a request to its parent. A node will additionally send a request to its parent when one of its buffers *becomes* empty: either the node has started the computation of a task or the node has started to communicate a task to one of its children. A node is allowed to 'grow' another buffer in the following situations:

1. when the node's buffers *all become* empty and there is an outstanding request from one of its children,

2. when the node completes the communication of a task to one of its children and there is an outstanding request from one of its children but the node's buffers are all empty,

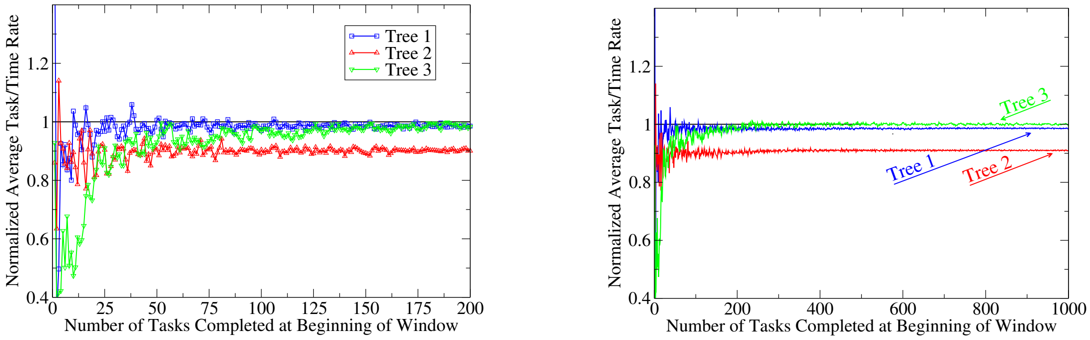
3. and when the node completes the computation of a task, and its buffers are all empty.

We considered many buffer growth events for our protocol. We found that the combination of the events listed above worked well for our random trees (see Section 4 for more details). They allowed almost every node to grow its necessary buffers, while discouraging over-growth.

In practice, our bandwidth-centric non-interruptible communication protocol has some undesirable characteristics. The maximum number of buffers needed for some trees can be prohibitively large. Also, the over-growth of buffers by some nodes may inhibit maximum steady-state throughput rate in some trees. In the next section we describe a protocol which assumes interruptible communications.

3.2. Interruptible communication

With interruptible communication, a request from a higher priority child may interrupt a communication to a lower priority child. The interrupted communication is shelved while a communication to the higher priority child is started. When the lower priority child comes again to the front of the priority queue, the interrupted communication is resumed from where it left off. A resumed communication can also be interrupted by a request from a higher priority child. And there may be more than one interrupted



(a) Startup.

(b) Entire run.

Figure 3. Throughput rates (tasks per time) over a sliding growing window normalized to the maximum steady state per tree. Each point on the x-axis designates a window from x to 2x tasks completed.

communication at a time, should a request come in from an even higher priority child. A high priority node like node B in Figure 2(a) will not need to stockpile tasks to sustain its maximum steady-state rate while its parent services a lower priority node.

Using interruptible communications allows us to follow the spirit of bandwidth-centric scheduling more closely: send tasks to the faster communicating nodes before sending tasks to the slower communicating nodes. With interruptible communication the fastest communicating nodes will never have to wait for another task so long as there is a task available for it to receive. Because it never waits on lower priority nodes, the highest priority node will need fewer buffers to sustain optimal steady-state throughput. All nodes will require fewer buffers, because no node waits on lower priority nodes. Therefore, interruptible communications alleviate the undesirable characteristics found in Section 3.1.

Similar to the non-interruptible case, a node will send a request to its parent when one of its buffers *becomes* empty. Because nodes receive tasks based upon their bandwidth priorities and their ability to receive tasks, no extra buffers are needed. The main advantage to using interruptible communication is that the number of buffers needed per node is not large. In Section 4 we use at most 3 buffers per node, in addition to having one buffer per child to hold the partially-completed transmissions. Finally, note that interruptible communications are easy to implement in practice and should cause little overhead (e.g. with multi-threading).

4. Simulation Results

4.1. Methodology

To validate our protocols, we simulate the execution of an application on a large number of random trees. Each tree is described by five parameters: m, n, b, d, x. Each tree has a random number of nodes between m and n. After creating the desired number of nodes, edges are chosen one by one to connect two randomly-chosen nodes, provided that adding the edge doesn't create a cycle. Each link has a random task communication time between b and d timesteps. Each node has a random task computation time between $x/100$ and x timesteps. All random distributions are uniform. Unless otherwise noted, the parameters we used for the simulations reported in this paper were $m = 10$, $n = 500$, $b = 1$, $d = 100$ and $x = 10,000$. The trees generated with these parameters had an average of 245 nodes, and ranged in depth from 2 to 82. (The distribution in depths can be seen in figure 6(b).)

We do not claim that these trees necessarily correspond to actual networks; our goal was simply to try a wide variety of trees to help us find flawed protocols. Indeed, the simulations helped us search the design space as described in the previous section. We implemented a simulator using the Simgrid toolkit [8]. Simgrid provides core functionalities for simulating distributed applications in heterogeneous distributed environments for the purpose of evaluating scheduling algorithms.

Determining when the execution of the application has reached steady state, maximum or otherwise, is difficult. The bandwidth-centric theorem gives the maximum steady-state rate for each node as well as the entire tree. But the period for each rate is unknown, and its only known bound is impractically large. To compensate for this, we computed the average execution rate in a *sliding window*. The window was made to increase in size as the simulation proceeded, so that eventually it would exclude the startup phase but would include a full period.

To illustrate our methodology we show an example in Figure 3(a). The figure plots the throughput rates over time, normalized to their respective maximum steady-state rates, for three selected trees. In the graphs, the y-axis value at point x on the x-axis represents the average rate between the time t_x when task x is completed and time t_{2x} when task $2x$ is completed. Thus, it is $(2x - x)/(t_{2x} - t_x)$. These

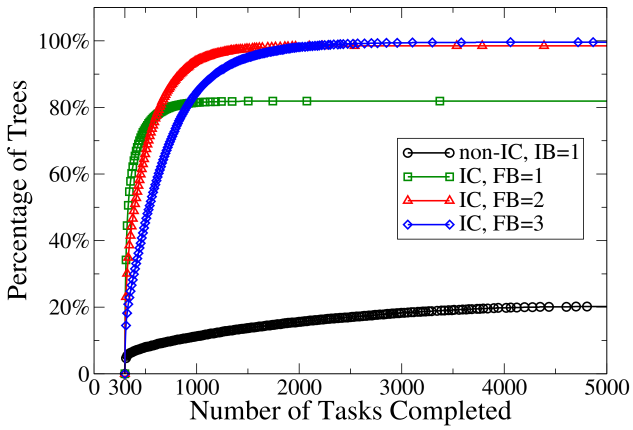


Figure 4. Achieving Maximal Steady State. For each point (x,y) on the graph, after x tasks have been completed we detect that y% of the trees have reached the onset of their maximum steady-state execution rate.

three particular trees were chosen to illustrate the difficulty in determining the onset of steady- state behavior.

All three trees eventually reach a steady state, which is clear from looking at the same data over a period of 1000 windows in Figure 3(b). Notice that the graphs of average execution time wiggle slightly, due to the discrete nature of task completion. However, the wiggles center around horizontal lines that are the steady-state execution rates (normalized to the maximum possible rate). Because of the wiggling, the curves sometimes exceed and sometimes are less than the steady-state rate.

Only tree 3 reaches the maximum steady-state rate, tree 1 comes very close to optimal, and tree 2 is well below optimal. It would be tempting to assume that the maximum theoretical steady-state rate is achieved if the average in sliding window ever reaches 1. However, Figure 3(a) reveals that tree 1 exceeds the optimal rate at several initial points, before it eventually settles to a near-optimal steady state. In contrast, tree 2 has less variation at the beginning. Tree 3 has low rates for a longer period than the other two, but the normalized rate steadily climbs toward optimal.

Based on our simulations with 1,000 random trees with the parameters as described earlier, we observed the following. If a tree did not reach maximum steady state before 1000 tasks, it had at most one point above the maximum steady-state rate beyond window 300. Also, we found that if a tree reached the maximum steady-state rate, then it had more than one point *above* the maximum steady-state rate beyond window 300. Based on this, we decide whether a tree has reached steady state or not as follows. We arbitrarily say that the tree has reached maximum steady state if its rate goes over the maximum steady-state rate *twice* after window 300. We say that the onset of maximum steady state occurs when the rate goes over the maximum steady-state rate for the second time after window 300. This heuristic is purely empirical, but works well in practice. We leave the development of more theoretically-justified decision criteria for future work.

Protocol	Number of Buffers					
	1	2	3	10	20	100
non-IC	0.0%	0.0	0.0	0.2	0.8	5.1
IC	81.9%	98.5	99.6	-	-	-

Table 1. Percentage of trees that reached the maximum steady-state rate using at most n buffers, for selected n between 1 and 100.

4.2. Results

We evaluate our protocols using a number of criteria. The most important criterion is how often they achieve the maximum steady-state rate. Another criterion is the number of buffers used or needed to reach maximum steady state. We present our evaluations in Sections 4.2.1 and 4.2.2, followed by a brief study of the adaptability of our protocols.

4.2.1. Reaching maximum steady state. We simulated an independent-task application of 10,000 tasks on 25,000 randomly assembled trees. With the non-interruptible communication protocol (*non-IC*), we give each node one initial buffer (*IB*), then allowed the pool of buffers to grow as dictated by the protocol. With the interruptible communication protocol (*IC*), we ran simulations for one, two, and three fixed buffers (*FB*). Figure 4 shows probability distribution functions for these four protocol variations. A point (x,y) on the graph indicates that y% of the trees reached the maximum steady-state rate within x tasks.

In Figure 4, the highest performer is interruptible communication with three fixed buffers, reaching the maximum rate in 99.57% of the trees. Interruptible communication with two fixed buffers is very close, reaching the maximum rate in 98.51% of the trees. The lowest interruptible performer has one fixed buffer, reaching the maximum steady-state rate in just less than 82% of the trees. This agrees with our findings on non-interruptible communication (see Section 3.1): one buffer is not sufficient to sustain the maximum steady-state rate in all trees.

Non-interruptible communication, starting with one initial buffer, reached the maximum rate in only 20.18% of the trees. As we will see in a moment, this protocol often required a large number of buffers. We also see much longer startup phases with non-interruptible communication: over half of the trees that reach the maximum rate with the non-IC protocol are still in the startup phase at 1000 tasks.

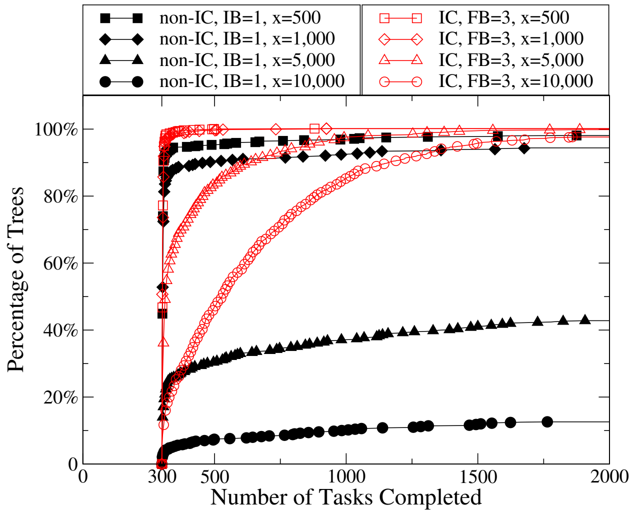


Figure 5. Impact of computation-to-communication ratios. Achieving maximum steady state across classes of trees with various computation-to-communication ratios.

The performances of the three IC protocols display a range of desirable characteristics. With FB=1, we see shorter startup phases but it reaches the maximum steady-state rate in fewer trees. With FB=3, we see longer startup phases but it reaches the maximum steady- state rate in the highest number of trees. FB=2 more closely matches the shorter startup phases of FB=1 and the higher number of trees of FB=3.

Figure 5 displays the impact of computation-to-communication ratios on our protocols. We randomly assembled 1000 trees with number of node parameters ($m = 10, n = 500$) and communication parameters ($b = 1, d = 100$). We created four classes of these trees by varying the value of the computation parameter x. For each class, x is one of $[500, 1000, 5000, 10000]$. Thus each tree class has a different range of computation-to-communication ratios $[x/10000$ through $x/1]$.

For two protocols, non-IC with IB=1 and IC with FB=3, Figure 5 shows the percentage of trees within each class that reach the maximum steady-state rate on an application of 4000 tasks and a window 300 threshold. We have two direct observations. First, IC with FB=3 performs well for all four classes of trees. Second, non-IC with IB=1 suffers greatly with the rise in computation-to-communication ratio. From other simulations not displayed here, we observe that for all protocols the startup time increases as the computation-to-communication ratio increases. Furthermore, while IC with FB=1 is inadequate with applications of any size, the choice of using the IC protocol with 2 or 3 fixed buffers may be based upon application size.

x	Median Buffers per #tasks completed 100	1000	4000	Maximum Buffers
500	3	3	3	165
1,000	4	5	5	472
5,000	150	212	218	1535
10,000	551	560	561	1951

Table 2. Median and maximum number of buffers used by non-IC, across various tree classes and number of tasks completed.

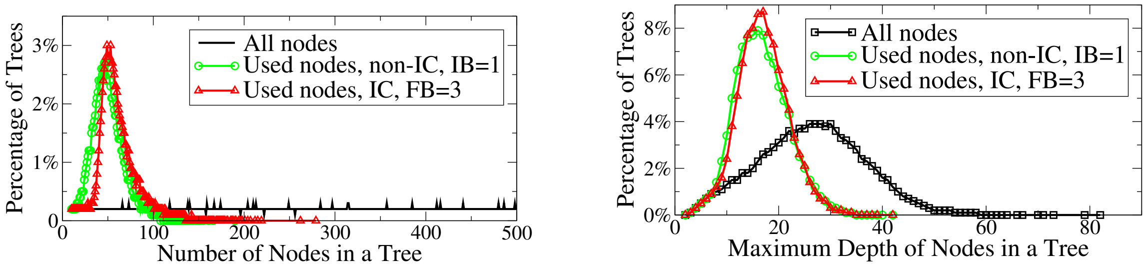
Figure 6 shows that, with the higher computation-to-communication ratios associated with the parameters chosen for our simulations, significant subtrees actually received tasks in the simulations. Figure 6(a) shows that it was usually more than 50 nodes and Figure 6(b) shows the typical sub-tree depth was around 18. There is a slight difference between the two protocols shown; the non-IC protocol occasionally used a larger or deeper tree than the IC protocol with three buffers.

4.2.2. Buffer usage. Table 1 shows the relationship between the maximum number of buffers used and reaching the maximum steady-state rate. The IC protocol required only two buffers for all but 1.5% of the 25,000 trees, and every 7 in 10 of those remaining 1.5% trees required only three buffers. (We note that the IC protocol also requires one additional buffer per child to hold the partially-transmitted data.)

However, the non-IC protocol required far more buffers. For instance, Table 1 shows that even when the number of buffers was limited to 100, non-IC achieved the maximum steady-state rate in only 5.1% of the trees. Table 2 displays median and maximum numbers of buffers used by non-IC with IB=1 across the four classes of trees with various computation-to-communication ratios. It shows that rampant buffer growth is a problem for non-IC. With the highest ratio tree class, ($x = 10,000$), the maximum number of buffers used was 1951. The lowest ratio tree class, ($x = 500$), is the best situation of the four for non-IC. Even though the worst-case increased to 165, the median number of buffers used was 3.

SubSubSectionAdapting to Changing Information

We contend that autonomous, bandwidth-centric scheduling can adapt to changes in the underlying network, since decisions are made locally. To illustrate this potential we re-visit our small model in Figure 1. We first simulate an application consisting of 1000 fixed-size, independent tasks using our non-interruptible protocol with two fixed buffers. We focus our attention on node P_1 with communication time c_1=1 and processor work time w_1=3. To simulate an increase in communication contention, we change c_1 to 3



(a) Tree Size PDF

(b) Tree Depth PDF

Figure 6. Tree characteristic comparisons between the entire tree with all nodes and the subtree composed only of *used* nodes (nodes that computed tasks during protocol simulation).

after 200 tasks are completed. Separately, to simulate a drop in processor contention, after 200 tasks we change w_1 to 1. Figure 7 shows these results.

Overall in Figure 7(a), we see that our protocol adjusts its steady-state performance with each change in the underlying network. The details in Figure 7(b), show the optimal steady-state performance for each instance of the network using dashed lines (original, change in c1, change in w1) along with the results of the simulation. Notice that for each change, the protocol performance adapts to closely approximate the optimal steady-state performance. This example shows the potential of our protocols for coping with underlying changes in the computing platform. Of course, there are many questions about speed of adapting and stability that must be addressed in future work.

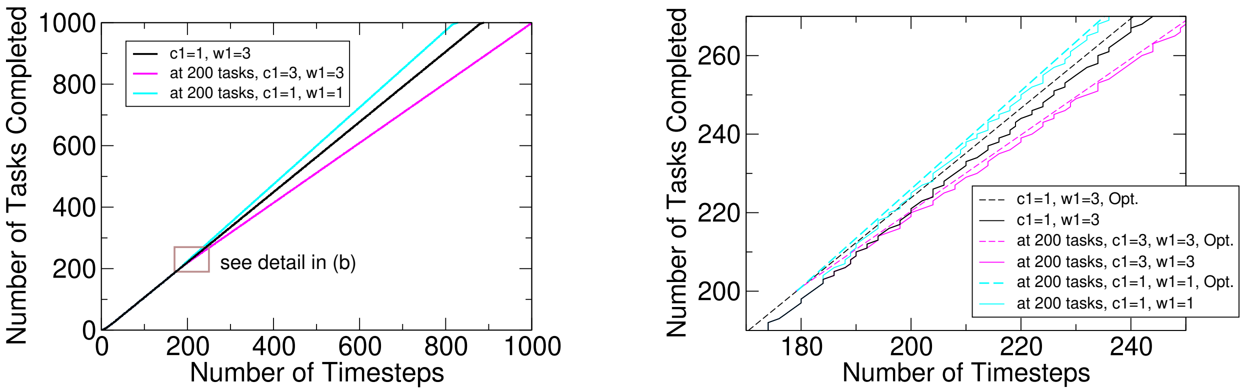
5. Related Work

The question of scheduling independent tasks onto heterogeneous sets of resources is a well known problem [18, 21, 16, 13, 26, 36, 23, 7, 2, 6, 9, 28, 4, 39], which has been studied with various sets of assumptions concerning both the application and the computing platform. Our work departs from previous approaches in that we develop a distributed, *autonomous* scheduling strategy. The major advantage of our approach is that it accommodates large-scale computing platforms on which centralized control is not feasible. The notion of decentralized scheduling for parallel computing has been explored by a few authors. For instance, the work in [22] presents a simple scheduling heuristic based on a K-nearest neighboring algorithm; the works in [30] and [19] study hierarchical job scheduling on metacomputing systems. These works show that decentralized scheduling compares favorably to centralized scheduling with the added benefit of scalability. By contrast with our work, these and other previous approaches consider only two-level scheduling schemes. The need for decentralized decision making and scheduling has perhaps been felt the most in the peer-to-peer community and interesting work has been done in the area of distributed storage applications [29, 27, 35]. Similar accomplishments have yet to be achieved for applications that involve significant communications, and our work provides a fundamental first step.

Our work is related to a number of endeavors that provide usable software for deploying applications on large-scale platforms [10, 1, 32, 12, 25]. At the moment, most of these projects employ some flavor of centralized scheduling. The works in [33, 17, 20] allow for the well-known hierarchical master/worker scheme, which is strongly related to our work. However, hierarchical master/worker has rarely been deployed in practice due to the lack of a decentralized, autonomous scheduling methodology. Our work provides such a methodology.

6. Conclusion and Future Work

In this paper we have investigated protocols for scheduling applications that consist of large numbers of identical, independent tasks on large-scale computing platforms. By imposing a tree structure on an overlay network of computing nodes, we had seen in our previous work that it is possible to compute the schedule which leads to the optimal asymptotic performance (i.e. optimal steady-state task completion rate). However, implementing this optimal schedule in practice, without prohibitive global coordination of all the computing nodes, remained an open question. To address this question we developed a number of *autonomous* scheduling protocols, i.e. distributed scheduling algorithms by which each node makes scheduling decisions based solely on locally available information. We have discussed such protocols in two different contexts: with non-interruptible and with interruptible communications. We have pointed at limitations of the non-interruptible communication case in terms of prohibitive number of buffers needed at each node and it's low performance on trees with higher communication-to-computation ratios. In order to evaluate our different protocols we performed extensive simulations. Our main finding was that an autonomous protocol requiring only 3 buffers per nodes reaches the optimal steady-state performance in over 99.5% of our simu-



(a) Overall

(b) Detail

Figure 7. Adaptability: Bandwidth-centric's potential to adapt to communication contention (changes in c_1) and processor contention (changes in w_1).

lations for the interruptible communication scenario. The autonomous scheduling approach is inherently scalable and adaptable, and thus ideally suited to currently emerging computing platforms. In particular this work has direct impact on the deployment of large applications on Grid and peer-to-peer computing platforms.

One question we have not addressed is that of the tree overlay network. Some trees are bound to be more effective than others. In future work we will perform analysis, simulations, and real-world experiments to understand on what basis the overlay network should be constructed. We will also conduct simulations and experiments to assess the resilience of our scheduling approach to changes in resource conditions and to dynamically evolving pools of resources. Finally, we will implement prototype software that uses interruptible communication and autonomous scheduling to deploy real applications on real computing platforms.

7. Acknowledgments

This work has been supported in part by NSF Research Infrastructure Grant 9802219, NSF Award ACI-0086092, and La Sierra University.

References

[1] D. Abramson, J. Giddy, I. Foster, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid ? In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'00)*, May 2000.

[2] D. Andresen and T. McCune. Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *Proceedings of the Seventh International Symposium on High Performance Distributed Computing (HPDC-7)*, pages 301–308, July 1998.

[3] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric Allocation of Independent Task on Heterogeneous Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida*, April 2002.

[4] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'01), Newport Beach, California*, pages 419–426, October 2001.

[5] F. Berman. High-performance Schedulers. In *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[6] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE computer society press, 1996.

[7] R. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW'99)*, pages 15–29, Apr. 1999.

[8] H. Casanova. Simgrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 430–437, May 2001.

[9] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, May 2000.

[10] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000.

[11] J. Cowie, B. Dodson, R. Elkenbrach-Huizing, , A. Lenstra, P. Montgomery, and J. Zayer. A World Wide Number Field Sieve Factoring Record: On to 512 Bits . *Advances in Cryptology*, pages 382–394, 1996. Volume 1163 of LNCS.

[12] Entropia Inc. `http://www.entropia.com`, 2001.

[13] S. Flynn Hummel, J. Schmidt, R. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 318–328, Jun 1996.

[14] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1999.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3), 2001.

[16] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997.

[17] E. Heymann, M. Senar, E. Luqye, and M. Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *Proceedings of the First IEEE/ACM Internation Workhop on Grid Computing (GRID 2000), Bangalore, India*, December 2000.

[18] 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.

[19] H. James, K. Hawick, and P. Coddington. Scheduling Independent Tasks on Metacomputing Systems. In *Proceedings of Parallel and Distributed Computing Systems (PDCS'99), Fort Lauderdale, Florida*, August 1999.

[20] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive Parallelism under Equus. In *Proceedings of the Second International Workshop on Configurable Distributed Systems, Carnegie-Mellon University*, pages 172–182, March 1994.

[21] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11:1001–1016, 1984.

[22] C. Leangsuksun, J. Potter, and S. Scott. Dynamic Task Mapping Algorithms for a Distributed Heterogeneous Computing Environment. In *Proceedings of the Heterogeneous Computing Workshop (HCW'95)*, pages 30–34, April 1995.

[23] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, Apr. 1999.

[24] J. Mitchell, A. Phillips, J. Rosen, and L. Ten Eyk. Coupled Optimization in Protein Docking. In *Proceedings of RECOMB 1999*, 1999.

[25] Parabon Inc. `http://www.parabon.com/clients/frontierEnteprise.jsp`, 2001.

[26] C. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36:1425–1439, 1987.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM'01, San Diego, California*, August 2001.

[28] A. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'01), Newport Beach, California*, pages 124–131, October 2001.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany*, November 2001.

[30] J. Santoso, G. van Albada, B. Nazief, and P. Sloot. Hierarchical Job Scheduling for Clusters of Workstations. In *Proceedings of the sixth annual Conference of the Advanced School for Computing and Imaging, Delft, Netherlands*, pages 99–105, June 2000.

[31] Scalable Visualization Toolkits. `http://vis.sdsc.edu/research/orion.html`, 2001.

[32] SETI@home. `http://setiathome.ssl.berkeley.edu`, 2001.

[33] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, UCSD, June 2001.

[34] J. Stiles, T. Bartol, E. Salpeter, , and M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[35] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM'01, San Diego, California*, August 2001.

[36] T. Tzen and L. Ni. Dynamic loop scheduling for shared-memory multiprocessors. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages II247–II250, 1991.

[37] United Devices, Inc. `http://www.ud.com`, 2002.

[38] M. Waterman and T. Smith. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[39] Y. Yang and H. Casanova. UMR: A Multi-Round Algorithm for Scheduling Divisible Workloads. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France*, April 2003. to appear.