

Performance Optimization of a De-centralized Task Allocation Protocol via Bandwidth and Buffer Management *

Bo Hong and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2562
{bohong, prasanna}@usc.edu

Abstract

Sharing the resources among various users and the lack of a centralized control are two key characteristics of many distributed heterogeneous computing systems. A critical challenge for designing applications in such systems is to coordinate the resources in a de-centralized fashion while adapting to the changes in the system. In this paper, we consider the computation of a large set of equal-sized independent tasks. This represents the computation paradigm for a variety of large scale applications such as SETI@home and Monte Carlo simulations. We focus on the performance optimization for a de-centralized adaptive task allocation protocol. We develop a bandwidth allocation strategy based on our de-centralized task allocation algorithm, and a simple task buffer management policy. Simulation results show that our task allocation protocol achieves close to the optimal system throughput.

1 Introduction

One of the key characteristics of many distributed heterogeneous systems is the lack of a centralized control [10]. This is especially the case when the distributed resources belong to different administrative domains. For example, a Grid may consist of compute resources from different universities, companies, and government organizations, each having its own administrative policy. A distributed system without a centralized control does not have complete knowledge of the system state and user requests, nor can it control the individual computers. A challenging problem for application design in such systems is to maximize system-wide resource utilization (e.g. system throughput,

task turnaround time) with de-centralized operations where each individual resource determines its own action based on locally available information. Additionally, since the resources are often shared among various users, from a single user's point of view, the system is dynamic and the capabilities of the available resources for a given application vary continuously. Such a dynamic nature makes efficient utilization of these distributed resources even more challenging. In order to maintain a sustained speed-up over stand-alone systems, the execution of applications must be *adaptive* to the changes in a dynamic heterogeneous system [1, 23].

We study the computation of a large set of equal-sized independent tasks on a dynamic heterogeneous computing system. This represents the computation paradigm for a wide range of research and commercial activities. In these activities, the computation of a complex problems is divided into many small independent work units that are processed in parallel. Examples of such activities include signal processing applications such as SETI@home [13], life science studies such as protein folding [15], computations in mathematical sciences such as the verification of Riemann's hypothesis [21], data encryption/decryption [6], etc. This computation paradigm is also exemplified by more tightly coupled computations such as Monte Carlo simulations.

In our previous study, we focused on the mathematical formulation of the above computation paradigm and showed that the throughput maximization problem can be efficiently solved in a distributed and adaptive fashion [9]. By modeling the computation at the compute nodes as a special type of data flow, we transformed the system throughput to the network flow in a corresponding graph. More importantly, for the network flow maximization problem, we developed a decentralized task allocation algorithm that can adapt to the changes in the system.

In this paper, we address the practical issues of imple-

*Supported by the National Science Foundation under award No. ACI-0204046 and in part by an equipment grant from Intel Corporation.

menting this task allocation algorithm as a distributed task allocation protocol. We consider the network interface of the compute nodes to be capable of performing concurrent communications with multiple neighbors. The network interfaces are therefore shared by the concurrent communications and we develop a strategy to allocate the bandwidth, which is based on our task allocation algorithm. Furthermore, since tasks are atomic and the real-valued task allocation generated by our task allocation algorithm cannot be directly applied, task buffers are used to convert the real-valued task allocation to integer-valued allocation. In this paper, we show that only a moderate amount of local storage is required by the task buffers. This fact leads to a simple task buffer management policy. Simulations show that such a policy, when combined with our bandwidth allocation strategy, ensures close to optimal system throughput.

The rest of this paper is organized as follows. Related works are reviewed in Section 2. In Section 3, we describe our system model and present a brief summary of our previous work - the Incremental Push-Relabel algorithm for throughput maximization. Experimental studies and performance optimizations of the implementation of the Incremental Push-Relabel is presented in Section 4. Concluding remarks are made in Section 5.

2 Related Work

Task scheduling for heterogeneous computing systems has received a lot of attention recently. Minimizing the overall computation time of all tasks (*makespan*) is known to be NP-complete [11] and hence heuristic based approaches have been extensively studied [2, 4, 19, 20, 22]. Instead of minimizing the makespan, some studies focus on the maximization of the system throughput. For example, the Condor project [18] develops a software infrastructure so that heterogeneous resources with distributed ownerships can be utilized to provide large amounts of processing capacity over long periods of time. The master-slave computation is widely used by many applications and has been exploited by various research efforts ([8, 17]) to maximize the throughput. The multi-level master-slave paradigm is studied in [3], where a bandwidth-centric approach was proposed to maximize the throughput of systems that are connected via a *tree topology*. Besides the design of optimal/approximation scheduling algorithms and the development of actual software infrastructures, a few other works have addressed the practical issues that may occur during the implementation of task allocation/scheduling algorithms. For example, for Internet based master-slave computations, the impact of task duplication and timeouts was studied in [12]. Assuming a tree-structured system topology, an autonomous task allocation protocol was proposed in [14].

Our study departs from the above research works in that we develop a *distributed* and *adaptive* task allocation protocol for systems with *general* topologies. Compared with centralized task allocation strategies, a distributed one can achieve a comparable system-wide performance while being scaled to very large systems, which is often the practical requirement of large scale applications such as SETI@home and peer to peer computation. Since many heterogeneous systems consist of resources that are geographically distributed and belong to different organizations, they do not exhibit any regular topologies (e.g. a tree or a ring), nor are they static. Our study provides a useful methodology toward task allocation in the context of large scale, distributed, and dynamic systems.

3 System Model and Distributed Adaptive Task Allocation Algorithm

The compute nodes are assumed to be connected via an *arbitrary* topology and the system is represented by a directed graph $G(V, E)$. Each node $V_i \in V$ in the graph represents a compute node. The weight of V_i is denoted by w_i . w_i represents the processing power of node V_i , i.e. V_i can perform one unit of computation in $1/w_i$ time. Each edge $E_{ij} \in E$ in the graph represents a network link from V_i to V_j . The weight of E_{ij} is denoted by l_{ij} . l_{ij} represents the communication bandwidth of link E_{ij} , i.e. Link E_{ij} can transfer one unit of data from V_i to V_j in $1/l_{ij}$ time. The links are uni-directional, so G is directed and $E_{ij} \neq E_{ji}$. In the rest of the paper, ‘edge’ and ‘link’ are interchangeably used. The successors of V_i in G is defined as $\sigma_i = \{V_k \in V | E_{ik} \in E\}$ and the predecessors of V_i in G is defined as $\psi_i = \{V_k \in V | E_{ki} \in E\}$.

The compute nodes are assumed to be equipped with full-duplex network interfaces that allow concurrent communications with multiple adjacent nodes. Due to the hardware limitations of the network interfaces or the specifications of network protocol, the rate with which a network interface sends and receives data cannot increase infinitely, even with concurrent communications. The capabilities of the network interfaces of each node V_i is described by two parameters: c_i^{in} and c_i^{out} : within one unit of time, at most c_i^{in} (c_i^{out}) units of data can flow into (out of) V_i . We further assume that the compute nodes can perform computation and communication concurrently.

Without loss of generality, we assume that each task consumes one unit of source data and requires one unit of computation. (If not so, we can normalize the values of w_i and l_{ij} .) The tasks are independent of each other and do not share the source data. A compute node can compute a task only after receiving the source data of the task. Initially, node V_0 holds the source data for all the tasks. Except V_0 , all other compute nodes in the system face the same ques-

tions: (1) where to get the tasks from? (2) how many tasks to compute locally? (3) where to send the remaining tasks?

Instead of minimizing the overall execution time of all tasks, we explored the maximization of the system throughput in [9]. The results are briefly summarized below.

Let $f(V_i, V_j)$ denote the number of tasks transferred from V_i to V_j in one unit of time. For notational convenience, if edge $E_{ij} \notin E$, we define $l_{ij} = 0$; if the actual data transfer is from V_i to V_j , we define $f(V_j, V_i) = -f(V_i, V_j)$. With these two definitions, if neither E_{ij} nor E_{ji} belongs to E , then $l_{ij} = l_{ji} = 0$, which implies that $f(V_i, V_j) = f(V_j, V_i) = 0$. In this way, we can define $f(V_i, V_j)$ over $V \times V$, rather than being restricted to E . $f(V_j, V_i) = -f(V_i, V_j)$ also allows us to compute the total number of tasks transferred to V_i as $\sum_{V_k \in \psi_i \cup \sigma_i} f(V_k, V_i)$, which equals $\sum_{V_k \in V} f(V_k, V_i)$ since $f(V_i, V_j) = f(V_j, V_i) = 0$ if $E_{ij} \notin E$ and $E_{ji} \notin E$. We first obtained the following linear programming problem formulation.

Base Problem: Given an undirected Graph $G(V, E)$, where node V_i has weight w_i and associated parameters c_i^{in} and c_i^{out} , and edge E_{ij} has weight l_{ij} . $w_i > 0$. $c_i^{in} > 0$. $c_i^{out} > 0$. $l_{ij} > 0$ if $E_{ij} \in E$ and $l_{ij} = 0$ otherwise.

Maximize: $\mathcal{W} = w_0 + \sum_{V_i \in V - \{V_0\}} (\sum_{V_k \in V} f(V_k, V_i))$

Subject to:

$$\begin{aligned} f(V_i, V_j) &\leq l_{ij} && \text{for } \forall V_i, V_j \in V && 1 \\ \sum_{V_k \in V} f(V_k, V_i) &\geq 0 && \text{for } \forall V_i \in V - \{V_0\} && 2 \\ \sum_{V_k \in V \& f(V_i, V_k) > 0} f(V_i, V_k) &\leq c_i^{out} && \text{for } \forall V_i \in V && 3 \\ \sum_{V_k \in V \& f(V_k, V_i) > 0} f(V_k, V_i) &\leq c_i^{in} && \text{for } \forall V_i \in V && 4 \\ \sum_{V_k \in V} f(V_k, V_i) &\leq w_i && \text{for } \forall V_i \in V - \{V_0\} && 5 \\ f(V_j, V_i) &= -f(V_i, V_j) && \text{for } \forall V_i, V_j \in V && 6 \end{aligned}$$

The Base Problem can be transformed to a network flow problem with the extended network flow (ENF) representation. Given an instance of the Base Problem, its ENF representation is obtained using the following procedure:

Procedure 1:

1. Create a node S with weight 0.
2. For each node V_i in the base problem, create three nodes: V_i^* , V_i' , and V_i'' . The weight of the three new nodes are 0. Add a directed edge of weight c_i^{in} from V_i' to V_i^* , a second directed edge of weight c_i^{out} from V_i^* to V_i'' , and another directed edge of weight w_i from V_i^* to S . V_0^* is the root node in the ENF representation.
3. For each edge E_{ij} in the base problem, create an edge E'_{ij} , where E'_{ij} is from V_i'' to V_j' and has weight l_{ij} .

We call the hypothetical node S the *sink* node of the ENF representation. Based on our ENF representation, we have the following flow maximization problem:

Problem 1: Given a directed graph $G(V, E)$, where edge E_{ij} has weight $l_{ij} > 0$, $l_{ij} = 0$ if $E_{ij} \notin E$, a root node V_0 , and a sink node S .

Maximize: $\mathcal{W} = \sum_{V_i \in V} f(V_0, V_i)$

Subject to:

1. $f(V_i, V_j) \leq l_{ij}$ for $\forall V_i, V_j \in V$
2. $f(V_j, V_i) = -f(V_i, V_j)$ for $\forall V_i, V_j \in V$
3. $\sum_{V_j \in V} f(V_i, V_j) = 0$ for $\forall V_i \in V - \{V_0, S\}$

If an instance of the Base Problem has G as the input graph and V_0 as the root node, we denote it as Base Problem (G, V_0) . If an instance of Problem 1 has G as the input graph, V_0 as the root node, and S as the sink node, we denote it as Problem 1 (G, V_0, S) . We use $\mathcal{W}_B(G, V_0)$ to represent the maximum throughput for Base problem (G, V_0) . We use $\mathcal{W}_1(G, V_0, S)$ to represent the maximum throughput for Problem 1 (G, V_0, S) .

The next proposition shows that the Base Problem is a special case of Problem 1 under the ENF representation.

Proposition 1: Suppose Base Problem (G, V_0) is converted to Problem 1 (G', V_0^*, S) using Procedure 1, then

$$\mathcal{W}_B(G, V_0) = \mathcal{W}_1(G', V_0^*, S)$$

Problem 1 is a standard network flow maximization problem. In [9], we developed a decentralized and adaptive algorithm for the network flow maximization problem (Problem 1). This algorithm is an augmentation to the Push-Relabel algorithm [5] and is denoted as the *Incremental Push-Relabel* algorithm. In the Incremental Push-Relabel algorithm, every node in the graph determines its own behavior based on the knowledge about itself and its neighbors. No central coordinator or global information about the system is needed. More importantly, unlike the Push-Relabel algorithm, no global synchronization is needed when the Incremental Push-Relabel algorithm adapts to the changes in the system.

For each node $V_i \in G$, $e(V_i)$ is defined as $e(V_i) = \sum_{V_k \in V} f(V_k, V_i)$, which is the total number of tasks V_i receives. An integer valued auxiliary function $h(V_i)$ is also defined for $V_i \in G$, which will be explained in the algorithm. The algorithm is listed as follows:

1. **Initialization:** $h(V_i)$, and $f(V_i, V_j)$ are initialized as follows:

$$\begin{aligned} h(V_i) &= 0 && \text{for } \forall V_i \in V \\ f(V_i, V_j) &= 0 && \text{for } \forall V_i, V_j \in E \\ h(V_0) &= |V| \\ f(V_0, V_k) &= l_{0k} && \text{for } \forall V_k \in V \\ f(V_k, V_0) &= -l_{0k} && \text{for } \forall V_k \in V \\ e(V_i) &= \sum_{V_k \in V} f(V_k, V_i) && \text{for } \forall V_i \in V \end{aligned}$$

2. **Search for the maximum flow:**

Each node $V_i \in V - \{V_0, S\}$ conducts one of the following three operations as long as $e(V_i) \neq 0$:

- (a) *Push* (V_i, V_k) : applies when $e(V_i) > 0$ and $\exists V_k$ s.t. $l_{ik} - f(V_i, V_k) > 0$ and $h(V_i) > h(V_k)$,

$$\begin{aligned}
d &= \min(e(V_i), l_{ik} - f(V_i, V_k)) \\
f(V_i, V_k) &= f(V_i, V_k) + d \\
f(V_k, V_i) &= -f(V_i, V_k) \\
e(V_i) &= e(V_i) - d \\
e(V_k) &= e(V_k) + d
\end{aligned}$$

- (b) *Relabel*(V_i): applies when $e(V_i) > 0$ and $h(V_i) \leq h(V_k)$ for $\forall V_k \in \{V_k | l_{ik} - f(V_i, V_k) > 0\}$,

$$h(V_i) = \min_{V_k \in \{V_k | l_{ik} - f(V_i, V_k) > 0\}} h(V_k) + 1$$

- (c) *Rebalance*(V_i): applies when $e(V_i) < 0$,

while $e(V_i) < 0$

pick a V_k s.t. $f(V_i, V_k) > 0$

$$d = \min(-e(V_i), f(V_i, V_k))$$

$$e(V_i) = e(V_i) + d$$

$$e(V_k) = e(V_k) - d$$

$$f(V_i, V_k) = f(V_i, V_k) - d$$

$$f(V_k, V_i) = -f(V_i, V_k)$$

3. *Adaptation to changes in the system*: For the flow maximization problem, the only possible change that can occur in the system is the increase or decrease of the weight of some edges. Suppose the value of l_{ij} changes to l'_{ij} , the following four scenarios are considered when performing the Adaptation (V_i, V_j) operation:

- (a) if $l'_{ij} > l_{ij}$ and $f(V_i, V_j) < l_{ij}$, do nothing.
(b) if $l'_{ij} > l_{ij}$ and $f(V_i, V_j) = l_{ij}$, then

$$\begin{aligned}
h(V_0) &= h(V_0) + 3|V| \\
f(V_0, V_k) &= l_{0k} && \text{for } \forall V_k \in V \\
f(V_k, V_0) &= -l_{0k} && \text{for } \forall V_k \in A_0 \\
e(V_k) &= \sum_{V_j \in V} f(V_j, V_k) && \text{for } \forall V_k \in V
\end{aligned}$$

- (c) if $l'_{ij} < l_{ij}$ and $f(V_i, V_j) \leq l'_{ij}$, do nothing.
(d) if $l'_{ij} < l_{ij}$ and $f(V_i, V_j) > l'_{ij}$, then

$$\begin{aligned}
h(V_0) &= h(V_0) + 3|V| \\
f(V_0, V_k) &= l_{0k} && \text{for } \forall V_k \in V \\
f(V_k, V_0) &= -l_{0k} && \text{for } \forall V_k \in V \\
e(V_k) &= \sum_{V_j \in V} f(V_j, V_k) && \text{for } \forall V_k \in V \\
f(V_i, V_j) &= l'_{ij} \\
f(V_j, V_i) &= -l'_{ij} \\
e(V_i) &= e(V_i) + (l_{ij} - l'_{ij}) \\
e(V_j) &= e(V_j) - (l_{ij} - l'_{ij})
\end{aligned}$$

In the above algorithm, the ‘Push’, ‘Relabel’, and ‘Rebalance’ operations are called the *basic operations*. The next theorem shows the optimality and complexity of the Incremental Push-Relabel algorithm.

Theorem 3.1. *The Incremental Push-Relabel algorithm finds the maximum flow with $O(n^2 \cdot |V|^2 \cdot |E|)$ basic operations, where n is the number of adaptation operation performed, $|V|$ is the number of nodes in the graph, and $|E|$ is the number of edges in the graph.*

In summary, the Base Problem can be transformed to a standard network flow maximization problem by using the ENF representation. The Incremental Push-Relabel algorithm was further developed to solve the network flow maximization problem in a distributed and adaptive fashion. More details about the algorithm are available in [9].

4 Performance Optimization of the Decentralized and Adaptive Protocol

In this section, we study the performance optimizations of a de-centralized and adaptive task allocation protocol, which is based on the Incremental Push-Relabel algorithm.

The Incremental Push-Relabel algorithm is executed by hypothetical nodes in the ENF representation. Yet we can easily transport such executions to the actual compute nodes: an actual compute node V_i simply needs to execute the ‘Push’, ‘Relabel’, ‘Rebalance’, and ‘Adaptation’ operations for all the ENF nodes that V_i maps to. For the discussion in the rest of this section, V_i is used to refer to either a compute node or an ENF node, which can be easily clarified given the context. When V_i and V_j are used to refer to compute nodes, $f(V_i, V_j)$ is used to refer to the net amount of flow from V_i to V_j .

The optimal task allocation generated by the Incremental Push-Relabel algorithm does not tell us how to allocate the tasks yet. It only contains information like ‘ V_3 needs to transfer 0.38 tasks to V_6 in one unit of time’, or ‘ V_4 needs to compute 0.43 tasks in one unit of time’. But the tasks are atomic and the actual system can only deal with integer number of tasks. Furthermore, before the Incremental Push-Relabel algorithm finds the optimal task allocation, a node, say V_i , may have a positive valued $e(V_i)$, which means V_i is accumulating tasks at that time instance. Yet tasks need to be allocated in a way that the compute nodes do not accumulate more tasks than they can compute.

These issues are addressed by maintaining a task buffer at each compute node, which contains the source data of the tasks. Initially, the task buffer at the root node contains all the source data and all other task buffers are empty. Let $b(V_i)$ denote the length of the buffer at V_i . At any time instance, each compute node $V_i \in V$ operates as follows:

1. Contact the adjacent compute node(s) and perform the ‘Push’, ‘Relabel’, ‘Rebalance’, and ‘Adaptation’ operations, if necessary.
2. If $b(V_i) > 0$ and V_i is not computing any task, then set $b(V_i) \leftarrow b(V_i) - 1$, remove one task from the task buffer, and compute the task.
3. While $b(V_i) > 0$ and V_i is computing a task, send message ‘request to send’ to $\forall V_k \in \sigma_i$ s.t. $f(V_i, V_k) > 0$. If ‘clean to send’ is received from V_k , then set $b(V_i) \leftarrow b(V_i) - 1$ and send a task to V_k .

4. Upon receiving ‘request to send’, V_i acknowledges ‘clean to send’ if $b(V_i) \leq U$. V_i acknowledges a denial if $b(V_i) > U$. Here U is a pre-set threshold that limits the maximum number of tasks a task buffer can hold.

However, several specifications need to be made before the above procedure can be used as a practical task allocation protocol. First of all, the value of U needs to be determined. Secondly, if node V_i has some spare tasks to transfer to its neighbors but more than one neighbor wants the tasks, there should be a mechanism to solve the conflict among these neighbors. The above procedure attempts to assemble the Incremental Push-Relabel algorithm while generating integer-numbered task allocation. Since rounding the optimal solution of a linear programming problem does not always lead to an optimal solution of the corresponding integer programming problem, it is interesting to know if such a procedure may degrade the performance of the system. In the rest of this section, performance optimization of the task allocation is studied through simulations.

We simulate scenarios where compute resources are connected via the Internet. Empirical studies [7] have shown that the Internet topologies exhibit power laws (e.g. out-degree v.s. rank, number of nodes v.s. outdegree, number of node pairs within a neighborhood v.s. neighborhood size in hops, etc). Recent work in [16] has pointed out two critical phenomena that generate the observed topological properties: incremental growth and preferred connectivity. In our simulations, the graph representations of the systems are generated using Brite, which is a tool developed in [16] that generates networks with power law characteristics. The values of l_{ij} , c_i^{in} , and c_i^{out} are uniformly distributed between 0 and 1. w_i is uniformly distributed between 0 and w_{max} . Note that $1/w_{max}$ represents the average computation/communication ratio of a task. We simulate the scenarios where $w_{max} = 0.1$ and $w_{max} = 0.05$, which represents computation/communication ratio of 10 and 20, respectively.

4.1 Bandwidth Allocation

In our system model, we have assumed that the network interface of the compute nodes can perform concurrent communications with multiple neighbors. However, the available incoming(outgoing) bandwidth $c_i^{in}(c_i^{out})$ of the network interfaces is shared by the concurrent communications. In order to allocate the tasks without any conflicts, we need a mechanism to allocate the bandwidth of c_i^{in} and c_i^{out} to the multiple neighbors that are sending or receiving tasks from the same compute node.

The allocation of bandwidth is usually performed by the flow control of the network protocols, which are often built into the kernel of the operating systems. Flow control aims at congestion avoidance while ensuring high utilization of

the bandwidth resources and fairness among different connections. By default, we can simply leave the allocation of bandwidth to the flow control provided by the network protocols, which, most probably, will evenly allocate the bandwidth to different connections.

A slightly different solution is to modify or override the default flow control and allocate the bandwidth by considering the weight of the connections. As for the weighting of the connections, we have two strategies to choose from. The first strategy weights the connections according to the bandwidth of the links. For example, suppose V_1 with $c_1^{out} = 10Mbps$ is transferring tasks concurrently to four neighbors V_2, V_3, V_4 , and V_5 . Links $(V_1 \rightarrow V_2)$ and $(V_1 \rightarrow V_3)$ have the same bandwidth $l_{12} = l_{13} = 5Mbps$. Links $(V_1 \rightarrow V_4)$ and $(V_1 \rightarrow V_5)$ have the same bandwidth $l_{14} = l_{15} = 10Mbps$. Although $l_{12} = 5Mbps$ is the maximum achievable bandwidth of link $(V_1 \rightarrow V_2)$, the actual communication bandwidth allocated to $(V_1 \rightarrow V_2)$ is only $10 \times \frac{5}{5+5+10+10} = 1.67Mbps$ in this scenario. This is actually a greedy choice since network links with higher bandwidth are given priority, with the hope that the tasks can be distributed over the system faster.

As shown in [9], when the Incremental Push-Relabel algorithm terminates, the resulting optimal task allocation specifies not only which link to use, but also the number of tasks $f(V_i, V_k)$ that should be transferred over each link $(V_i \rightarrow V_k)$ in one unit of time. Our second strategy for bandwidth allocation is based on the observation that $f(V_i, V_k)$ indicates the bandwidth requirement of link $V_i \rightarrow V_k$ in the optimal task allocation. We allocate the bandwidth of value $f(V_i, V_k)$ to link $V_i \rightarrow V_k$. Such a strategy is feasible since the Incremental Push-Relabel algorithm guarantees that $f(V_i, V_k)$ never exceeds the bandwidth constraints of l_{ik} and the capability constraints of c_i^{out} and c_k^{in} , even before the optimization is completed. In other words, this strategy never allocates more bandwidth than what is available.

In summary, we have three strategies for bandwidth allocation: (1) even allocation, (2) proportional to the bandwidth of the links, and (3) using the flow information provided by the Incremental Push-Relabel algorithm. Instead of exploring the implementation details of various bandwidth allocation strategies, which might involve programming efforts such as the modification of the kernels, we focus on the performance evaluation of these strategies via simulations. In the simulations, the graphs are generated as discussed before, we simulated $w_{max} = 0.05$ and $w_{max} = 0.1$, which represent communication/computation ratio of 20 and 10, respectively. U , whose impact will be discussed in detail in Section 4.2, is temporarily set to infinity.

We first compare the steady state performance of the three strategies. The parameters l_{ij} , w_i , c_i^{in} , and c_i^{out} were

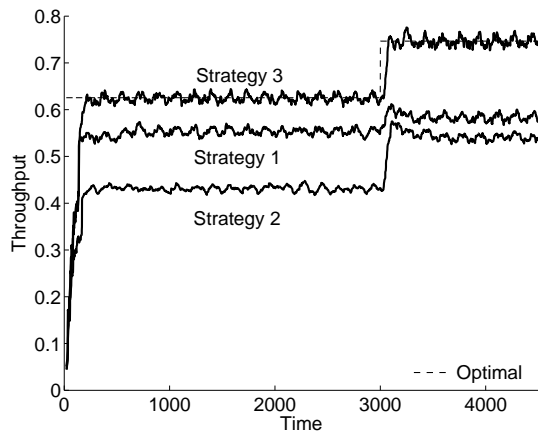


Figure 2. Adaptation to changes in the system

assumed to be constant for this set of simulations. 800 randomly generated systems were simulated, 200 with 20 nodes and $w_{max} = 0.05$, 200 with 20 nodes and $w_{max} = 0.1$, 200 with 80 nodes and $w_{max} = 0.05$, and 200 with 80 nodes and $w_{max} = 0.1$. Initially, there are 2500 tasks on node V_0 . The throughput of a system is calculated as $2500/t_{all}$, where t_{all} is the overall computation time of all tasks. The results in Figure 1 show the throughput of the three bandwidth allocation strategies. The reported throughput has been normalized with respect to the optimal throughput, which was calculated offline. The results are shown in the increasing order of the normalized throughput of Strategy 1. As can be seen from these results, task allocation using Strategy 3 achieves a close-to-optimal system throughput. In most of the experiments, Strategy 3 outperforms Strategy 1 and 2. Throughput improvement by a factor of up to 2.54 was observed. For those experiments that Strategy 2 or 3 outperforms Strategy 3, the performance of the three strategies were close to the optimal, and the difference between these three was negligible.

Not only do the three strategies differ in their steady state performance, they react to changes in the system differently. This is illustrated in Figure 2. The system consists of 20 nodes. The adjacency matrix was initialized as discussed above. $w_{max} = 0.1$. During the course of the simulation, the network condition and the effective compute power of the nodes were altered at time instance $t = 3000$ such that the compute power of a set of nodes was increased by 50% and the bandwidth of a selected set of links was increased by 20%. In Figure 2, the optimal throughput was calculated offline. The actual throughput for time instance t was calculated as $(N(t + 75) - N(t - 75))/150$, where $N(\tau)$ is the number of tasks computed by the system from time 0 to time τ . When $t < 75$, the actual throughput was calculated as $N(t)/t$.

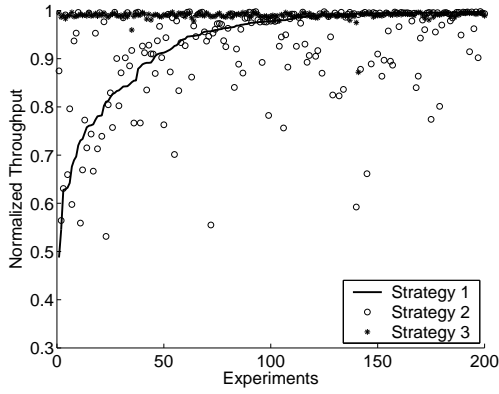
When changes in the system occurs at $t = 3000$, the adaptation procedure was activated and the Incremental Push-Relabel algorithm found a new optimal task allocation, which leads to a new optimal system throughput. All three strategies attempted to improve the system throughput by re-allocating the bandwidth of c_i^{out} 's and c_i^{in} 's according to the new optimal task allocation. As can be seen from Figure 2, while Strategy 3 achieved (close to) the new optimal system throughput, Strategies 1 and 2 failed to do so. Although Strategies 1 and 2 did improve the system throughput to some extent, the improvement was not proportional to the increase in the optimal throughput. In other words, Strategies 1 and 2 cannot guarantee the optimality of the system throughput under changes in the system, not even a guaranteed tight upper bound on the performance degradation from the optimal throughput.

These simulation results show that being bandwidth allocation based on fairness (Strategy 1) or greedy choices (Strategy 2) does not necessarily lead to an optimal throughput of the system. The compute nodes need to work in a coordinated manner, as indicated by Strategy 3, in order to obtain system-wide optimal resource utilization, as in our case, the throughput.

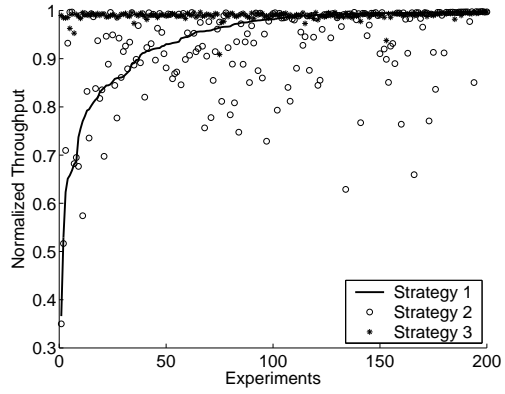
4.2 The Size of the Task Buffers

The task buffers are used to discretize the real-valued optimal task allocation generated by the Incremental Push-Relabel algorithm. In terms of storage requirement on the compute nodes, small buffers are preferred. However, larger buffer sizes enable a higher utilization of the network resources since task transfers do not have to be suspended while waiting for the receiver nodes to clear space in their task buffers. In this section, we study the impact of the buffer size on the performance of the system. Bandwidth Allocation Strategy 3, the best one as shown in Section 4.1, is used in this section. To simplify the discussions, we assume that all the compute nodes have task buffers of the same size U .

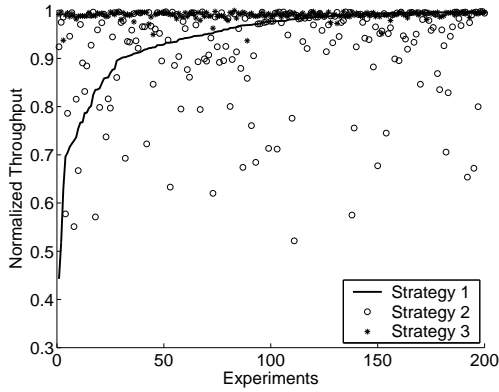
If the size of the task buffers is set to $U = \infty$, then a good indicator of storage requirement is the maximum length of task buffer that is actually consumed. This is illustrated through the simulations of 400 randomly generated systems with $w_{max} = 0.05$, 200 with 20 nodes and another 200 with 40 nodes. No changes occurred to the system during the simulations, hence no adaptations were activated. For each system, there were 2500 tasks on root node V_0 initially and all the nodes in the system have infinite task buffers. We monitored the maximum buffer consumed by each individual compute node and the results are summarized in Figure 3 in the form of histogram. The histogram in Figure 3 (a) is computed over all the 20-node systems (200 such systems in total), and the histogram in Figure 3 (b) is



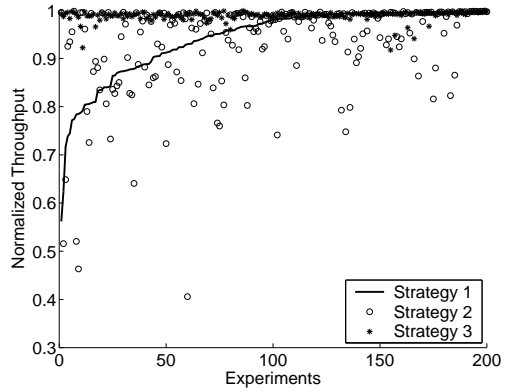
(a) 20 nodes, $w_{max} = 0.05$



(b) 20 nodes, $w_{max} = 0.1$



(c) 80 nodes, $w_{max} = 0.05$



(d) 80 nodes, $w_{max} = 0.1$

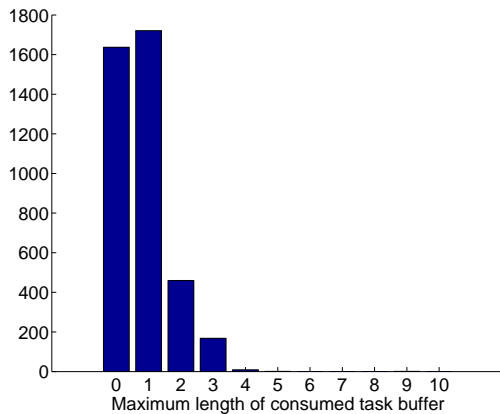
Figure 1. Performance comparison of three bandwidth allocation strategies. Details of the three strategies are discussed in Section 4.1. The X-axis represents the 200 experiments, for each comparison. The reported throughput has been normalized with respect to the optimal throughput calculated off-line.

computed over all the 40-node systems (200 such systems in total). The results in figure 3 show that task buffers of size 4 can cover more than 99% of the cases. More interestingly, task buffers of size 1 is big enough for more than 80% of the cases.

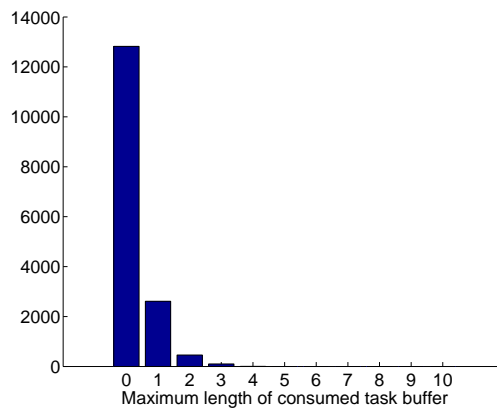
The following simulation results illustrate the impact on the system performance when the sizes of the task buffers are limited. We conducted simulations on the same 400 systems that were used to generate the results in Figure 3, but U was no longer set to infinity. For each system, we conducted two separated simulations, one with $U = 5$ and the other with $U = 1$. Again no changes to the systems occurred during the simulations. The results are shown in Figure 4, in the increasing order of the normalized throughput

of scenarios with $U = 1$. We can see that $U = 5$ leads to a higher, and closer to optimal, throughput than $U = 1$. We have observed in the simulations that further increasing the value of U did increase the system throughput. The benefit, however, becomes marginal as U gets larger, since the newly increased task buffers are rarely used. When the system was dynamic, similar results were observed, which shows that our task allocation does not need a large task buffer to adapt. Figure 5 shows one such example where the system consisted of 20 nodes and changes in the system occurred at time $t = 2000$ and $t = 6000$.

These simulation results show that our task allocation protocol is resource efficient as it requires only a moderate amount of local storages (task buffers) on the compute



(a) 20-node systems, 200 experiments



(b) 80-node systems, 200 experiments

Figure 3. Histogram of maximum length of consumed task buffer. The y -axis represents the total number of occurrence of a specific maximum task buffer length.

nodes. Consequently, the task buffers can be managed easily as they need not to be of the same size on all the compute nodes. Initially, each compute node simply allocates a task buffer that can contain a small number of, say 5, tasks. In most of the cases, this task buffer suffices the operation of our task allocation protocol. In the rare cases that more task buffer spaces are needed to receive a new task, allocate more local storage if this is allowed by the compute node, otherwise, simply ignore or reject the task transfer request. This reject-if-buffer-full policy is easy to implement and can achieve close to the optimal system throughput, as verified through the simulations.

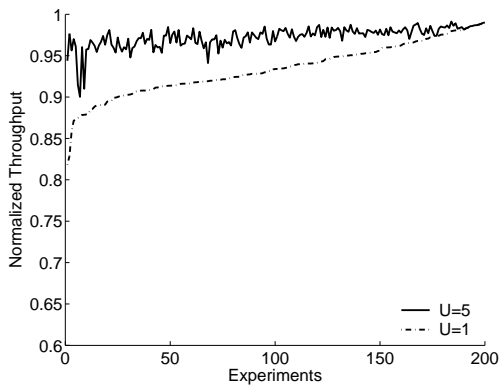
5 Conclusion

In this paper, we focused on the class of applications that compute a large set of equal-sized independent tasks in a distributed computing system. We studied the practical issues in the implementation of a distributed and adaptive task allocation protocol. This protocol is based on the Incremental Push-Relabel algorithm developed in our previous study. We designed a bandwidth allocation strategy to coordinate the concurrent communications that transfer data to/from the same node. This strategy, as shown by the simulations, outperforms the fairness- or greedy- based strategy by a factor of upto 2.54. Since tasks are atomic and the real-valued task allocation generated by the Incremental Push-Relabel cannot be directly applied, task buffers are used in our protocol to convert the real-valued task allocation to integer-valued allocation. We showed that only a moderate amount of local storage is required by the task buffers. This leads

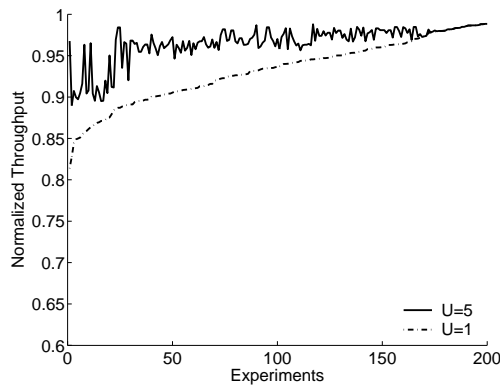
to a simple task buffer management policy that grows the task buffer if allowed and rejects task transfers when the task buffer overflows. Simulations show that such a policy, when combined with our bandwidth allocation strategy, ensures close to optimal system throughput.

References

- [1] A. Alhusaini, C. S. Raghavendra, and V. K. Prasanna. Run-Time Adaptation for Grid Environments. *Heterogeneous Computing Workshop, 15th International Parallel and Distributed Processing Symposium (IPDPS '01)*, 2001.
- [2] M. Arora, S. K. Das, and R. Biswas. A De-Centralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments. *International Conference on Parallel Processing Workshops (ICPPW'02)*, 2002.
- [3] O. Beaumont, A. Legrand, Y. Robert, L. Carter, and J. Ferrante. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.
- [4] T. D. Braun, H. J. Siegel, and N. Beck. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.
- [5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.
- [6] Distributed.net. <http://www.distributed.net>.
- [7] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM*, pages 251–262, 1999.
- [8] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations. *7th International*

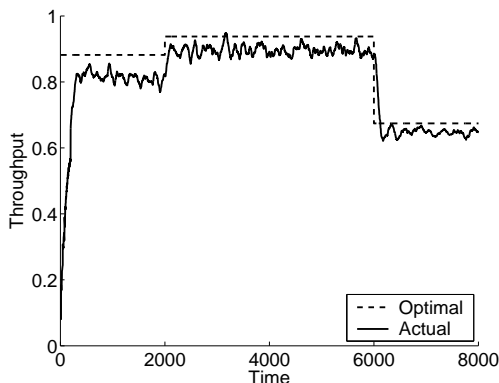


(a) 20-node systems, 200 experiments

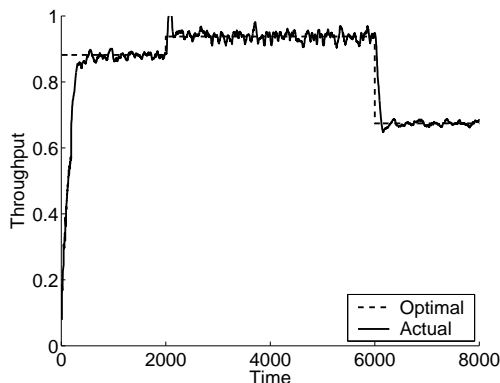


(b) 80-node systems, 200 experiments

Figure 4. Impact of buffer size on the throughput of the system



(a) $U = 1$



(b) $U = 5$

Figure 5. Impact of buffer size on the adaptation of the system

Conference on High Performance Computing (HiPC 2000), December 2000.

- [9] B. Hong and V. K. Prasanna. Distributed Adaptive Task Allocation in Heterogeneous Computing Environments to Maximize Throughput. To appear in the proceedings of IPDPS, April 2004. Also available as Technical Report CENG-2003-02, Department of Electrical Engineering, University of Southern California, http://www.usc.edu/~bohong/report_oct03.ps, October 2003.
- [10] I. Foster and C. Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
- [11] O. Ibarra and C. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM*, 24(2):280–289, 1977.
- [12] D. Kondo, H. Casanova, and F. Berman. Models and Scheduling Mechanisms for Global Computing Applications. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.
- [13] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home-Massively Distributed Computing for SETI. *Computing in Science and Engineering*, January 2001.
- [14] B. Kreaseck, L. Carter, H. Casanova, , and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent Task Applications. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
- [15] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology, Computational Genomics*, Richard Grant, editor. Horizon Press, 2002.
- [16] A. Medina, I. Matta, and J. Byers. On the Origin of Power Laws in Internet Topologies. *ACM Computer Communica-*

tion Review, 30(2):18–28, April 2000.

- [17] G. Shao, F. Berman, and R. Wolski. Master/Slave Computing on the Grid. *9th Heterogeneous Computing Workshop*, May 2000.
- [18] D. Thain, T. Tannenbaum, and M. Livny. *Condor and the Grid*, in F. Berman, A.J.G. Hey, G. Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.
- [19] H. Topocuoğlu, S. Hariri, and M. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.
- [20] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing, Special Issue on Parallel Evolutionary Computing*, 47(1):8–22, 1997.
- [21] S. Wedeniwski. ZetaGrid - Computations Connected with the Verification of the Riemann Hypothesis. *Foundations of Computational Mathematics conference, Minnesota, USA. Workshop on Computational Number Theory*, August 2002.
- [22] J. B. Weissman. Scheduling Multi-Component Applications in Heterogeneous Wide-area Networks. *Heterogeneous Computing Workshop, International Parallel and Distributed Processing Symposium IPDPS*, May 2000.
- [23] H. Zhu, M. Parashar, J. Yang, Y. Zhang, S. Rao, and S. Hariri. Self-Adapting, Self-Optimizing Runtime Management of Grid Applications Using PRAGMA. *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2003.