

Adaptive Allocation of Independent Tasks to Maximize Throughput

Bo Hong, *Member, IEEE*, and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—In this paper, we consider the task allocation problem for computing a large set of equal-sized independent tasks on a heterogeneous computing system where the tasks initially reside on a single computer (the root) in the system. This problem represents the computation paradigm for a wide range of applications such as SETI@home and Monte Carlo simulations. We consider the scenario where the systems have a general graph-structured topology and the computers are capable of concurrent communications and overlapping communications with computation. We show that the maximization of system throughput reduces to a standard network flow problem. We then develop a decentralized adaptive algorithm that solves a relaxed form of the standard network flow problem and maximizes the system throughput. This algorithm is then approximated by a simple decentralized protocol to coordinate the resources adaptively. Simulations are conducted to verify the effectiveness of the proposed approach. For both uniformly distributed and power law distributed systems, a close-to-optimal throughput is achieved, and improved performance over a bandwidth-centric heuristic is observed. The adaptivity of the proposed approach is also verified through simulations.

Index Terms—Task allocation, heterogeneous computing, network flow, decentralized algorithm, throughput.

1 INTRODUCTION

RECENTLY, distributed heterogeneous computing systems have become attractive platforms for high performance computing. In such systems, distributed resources such as workstations and supercomputers are connected through local and/or wide area networks. By utilizing these distributed resources in a coordinated manner, a heterogeneous computing system can meet the computational demands of complex applications [12].

In this paper, we consider the computation of a large set of equal-sized independent tasks in a heterogeneous computing system. In particular, we consider the scenario where each task is to process a fixed amount of source data. The source data of all the tasks initially resides on a single node in the system, which we call the root node. Other nodes in the system need to receive the source data of a task, either directly from the root node or indirectly through other computation nodes. This computation paradigm models a variety of research and commercial activities. Internet-based distributed computations are among the most well-known examples, which include SETI@home [18], Folding@home [21], data encryption/decryption [10], and so forth. The computation paradigm also models various applications that are typically executed on tightly coupled systems (for example, Monte Carlo simulations and Computational Phylogeny [24]).

The computation paradigm reduces to the general problem of allocating or scheduling independent tasks in

heterogeneous computing systems. One objective of performance optimization is to minimize the overall execution time (the *makespan*) of all the tasks. Although some special cases of this problem can be solved in polynomial time, the makespan minimization problem is known to be NP-complete in its general form. In this paper, we focus on an alternative optimization objective: maximizing the system throughput. When the application is to compute a large number of tasks, the system throughput is a suitable performance metric. When the application is of streaming style (such as SETI@home that virtually never ends), then the system throughput is possibly the only feasible performance metric when computation speed is the major concern.

Because computation and communication resources in distributed heterogeneous computing systems are typically shared among multiple users and applications, the network performance and the effective computation power of each node may vary at runtime. This is particularly true in the case of Internet-based computation, Peer-to-Peer Computation, and the Grid [15]. Optimizing the performance of the system based on a snapshot of the current system status may not lead to optimal performance. Consequently, the task allocation needs to be *adaptive* to the changes in the system [5], [27].

For the throughput maximization problem, we consider the system model in which 1) a computer can send and receive data to/from multiple neighboring computers concurrently and 2) computation and communication can be overlapped at the computers. Such a model represents the capabilities of typical modern computers. We show that the system throughput can be maximized in a distributed and adaptive fashion. We model the computation as a special type of data flow. This leads to our network flow representation for the throughput maximization problem. Based on this representation, we show that the system throughput can be transformed to the network flow in a

- B. Hong is with the Department of Electrical and Computer Engineering, Drexel University, Philadelphia, PA 19104. E-mail: bo.hong@drexel.edu.
- V.K. Prasanna is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: prasanna@usc.edu.

Manuscript received 22 Oct. 2004; revised 18 Sept. 2005; accepted 18 Aug. 2006; published online 9 Jan. 2007.

Recommended for acceptance by D. Trystram.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0260-1004. Digital Object Identifier no. 10.1109/TPDS.2007.1042.

corresponding graph. More importantly, for the network flow maximization problem, we develop a decentralized task allocation algorithm that adapts to the changes in the system. For the algorithm to be adaptive to parameter (computation power of the nodes and communication bandwidths of the links) changes in the system, we require the root node to be notified that a change has occurred when a change occurs, which is the only nonlocal message in the algorithm. This task allocation algorithm can be approximated as a decentralized task allocation protocol to coordinate the computation nodes. Simulations are conducted to verify the effectiveness of the proposed approach. A close-to-optimal throughput is achieved and improved performance over a bandwidth-centric protocol is observed. The adaptivity of the proposed approach (to parameter changes) is also verified through simulations.

Notice that we assume in this paper that computation can be overlapped with communication without affecting the performance of the other. For the scenario where the computation capability of a computer reduces if the computer is involved intensively in communications, we are currently unable to develop a distributed and adaptive task allocation algorithm. The impact of such an overlapping cost is further discussed in Section 8.

The rest of the paper is organized as follows: Related work is reviewed in Section 2. Section 3 describes our system model and formally states the optimization problem. The network flow representation is discussed in Section 4. Section 5 presents our decentralized adaptive task allocation algorithm that maximizes the system throughput. Based on this algorithm, a task allocation protocol is developed in Section 6. Experimental results are shown in Section 7. Section 8 discusses the cost of overlapping computation with communication and concludes the paper.

2 RELATED WORK

Task scheduling for heterogeneous computing systems has received a lot of attention recently. Various research efforts have addressed the minimization of the makespan. Because the makespan minimization problem is known to be NP-complete [16] in its general form, designing heuristics and evaluating their performance become the key issues. For example, static scheduling heuristics for a set of independent tasks are studied in [6], whereas the related dynamic scheduling problem has been studied in [22]. Other research efforts consider tasks with interdependencies. For example, a heuristic-based approach is studied in [31] to schedule multicomponent applications in heterogeneous wide area networks. In [7], a software-in-the-loop approach is proposed to design and implement task scheduling algorithms for heterogeneous systems. In [25], an augmentation to Java is proposed to develop divide-and-conquer applications in distributed environments and several scheduling heuristics are experimentally studied. Note that the optimization objective in this paper is different from these studies in that we focus on the maximization of the system throughput.

For computation in heterogeneous systems, maximizing the system throughput is not a new idea. A well-known example is the Condor project [30]. It develops a software infrastructure so that a distributed system with different

ownerships can be utilized in a uniform manner to provide high throughput computation. The throughput maximization problem has also been studied from various algorithmic aspects. The work in [29] considers heterogeneous computing systems that are connected via a general graph topology. The application is of streaming style: A task continuously receives data from certain preceding tasks, processes the received data, and sends the processed data to some other tasks. Finding the optimal mapping from tasks to computers (such that the throughput of data processing is maximized) is shown to be NP-complete, and a mapping heuristic is developed in [29]. A different scenario is considered in [3], where the system topology is also graph structured but the application consists of a set of independent identical problems and each problem in turn consists of a set of interdependent tasks. Their result shows that maximizing the throughput (defined as the number of problems executed in one unit of time) for this general scenario is also NP-complete.

Although these general scenarios of throughput maximization are NP-complete, better complexity results and algorithms have been obtained for some specific (and possibly more practical) scenarios of the application settings and system topologies. Throughput maximization for single-level master-slave computation in a Grid has been studied in [28], where the computation resources are considered to communicate with the root node only. When the application consists of a large set of equal-sized independent tasks, the throughput maximization in a general graph-structured system is studied in [1]. The problem is formulated as a linear programming problem, which is a well-studied problem with many available algorithms. However, these algorithms are centralized and are not suitable for distributed execution. In [4], a bandwidth-centric method has been obtained for the computation of equal-sized independent tasks when the computers are connected via a tree topology, which further leads to a localized autonomous task allocation strategy. When the nodes in the system are connected via a general graph topology, the problem of extracting a best spanning tree that has the highest throughput among all possible spanning trees is studied in [2]. The result in [2] shows that the achievable throughput of the optimal spanning tree, in the worst case, can be arbitrarily bad compared with that of the original graph-structured system.

Similar to some of the previous work, we also study throughput maximization for the computation of equal-sized independent tasks. Our study differs from the previous studies in that we develop a *distributed* and *adaptive* algorithm when the system has a *graph-structured* topology.

In this paper, we consider the system model in which 1) a node can send and receive data to/from multiple neighboring nodes concurrently and 2) computation and communication can be overlapped at the computers. This model, as well as several other more restrictive models, has been considered in previous studies, such as in [1]. The other models include, for example, a computer that may not send or receive at the same time, a computer that may not send/receive data to/from multiple computers, or computations

and communications that may not be overlapped. For all these models, the study in [4] develops a distributed allocation algorithm for tree-structured systems. For the model we consider, we develop a distributed and adaptive task allocation algorithm for general graph-structured systems.

3 SYSTEM MODEL

The nodes are assumed to be connected via an *arbitrary* topology, and the system is represented by a directed graph $G(V, E)$. Each node $u \in V$ in the graph represents a computation node. u has weight w_u , representing the processing power of u , i.e., u can perform one unit of computation in $1/w_u$ time. The edge $(u, v) \in E$ in the graph represents a network link from u to v . Edge (u, v) has capacity c_{uv} , representing the communication bandwidth of (u, v) , i.e., (u, v) can transfer one unit of data from u to v in $1/c_{uv}$ time. To model nonsymmetric communication links, the edges are unidirectional, so G is directed and $(u, v) \neq (v, u)$. In the rest of the paper, we use “edge” and “link” interchangeably. The successors of u in G are denoted as $\sigma_u = \{w \in V | (u, w) \in E\}$, and the predecessors of u in G are denoted as $\psi_u = \{w \in V | (w, v) \in E\}$.

Although the physical media in modern networking techniques may be simplex (such as most fiber optic communications that use one strand to send data in each direction) or half-duplex (such as unswitched Ethernet which allows at most one device to transmit at a time), full-duplex network interfaces have been widely implemented as the standard practice. Consequently, we only consider fully duplexed network interfaces, which means that the computation nodes can send and receive data concurrently.

We also assume that the network interfaces can communicate with multiple adjacent nodes concurrently. This represents the modern network techniques (for example, the packet-switching technique) that support concurrent communications. However, the rate with which a network interface sends and receives data cannot increase infinitely as the number of concurrent communications increases. The data transfer rate cannot exceed the hardware limitation of the network interface. Furthermore, there are typically a send and a receive buffer associated with each network interface. Implemented in either hardware or software, the buffers are used to control the data flow rate as specified by the network protocol. To reflect this limitation, for each node u , we introduce another two parameters: c_u^{in} and c_u^{out} . These two parameters indicate the capability of u to receive and send data: within one unit of time, at most c_u^{in} (c_u^{out}) units of data can flow into (out of) u .

We consider the scenario where the computation nodes can perform computation and communication concurrently. The overlapping of computation and communication is made possible by various hardware and software techniques. One such hardware technique is direct memory access (DMA) (see [9, Chapter 7.3]), where the peripheral (network card in our case) assumes control of the system bus to access memory directly. Another hardware technique, designed for high-end computers, is *dedicated message processing*, where a dedicated communication processor operates directly on the network interface and exchanges data with

the main processor via shared memory (see [9, Chapter 7.5] and the specifications of Cray XT3 [17]). Both techniques remove communication load from the CPU (at the cost of adding certain hardware resources). Examples of software techniques include message passing interface (MPI) [26], which supports the overlap via nonblocking communication primitives. Some researchers have also pointed out that a certain cost is associated with the overlapping of computation and communication [19], i.e., the computation capability of a computer may be reduced if the computer is involved intensively in communications. For the discussion in this paper, we do not consider the cost of overlapping. The impact of this cost will be discussed in Section 8.

Without loss of generality, we assume that each task is to perform one unit of computation on one unit of source data. The tasks are independent of each other and do not share the source data. A computation node can compute a task only after receiving the source data of the task. Initially, node s holds the source data for all the tasks. s is called the root node. Except s , all other computation nodes in the system need to answer the same questions: 1) Where do we get the tasks? 2) How many tasks do we compute locally? 3) Where do we send the remaining tasks?

The purpose of this study is to answer the three abovementioned questions for all nodes in the system such that the system throughput can be maximized. The throughput of a system is defined as the number of tasks computed by the system in one unit of time under steady state condition.

For convenience, we say that a task is transferred from u to v when the source data of a task is transferred from u to v . Let $f(u, v)$ denote the number of tasks transferred from u to v in one unit of time. We have the following formal problem statement:

Base Problem: Given a directed graph $G(V, E)$. Node $u \in V$ has weight $w_u > 0$, input constraint $c_u^{in} > 0$, and output constraint $c_u^{out} > 0$. Edge (u, v) has capacity constraint $c_{uv} > 0$. s is the distinguished root node.

Maximize:

$$w_s + \sum_{u \in V - \{s\}} \left(\sum_{v \in \psi_u} f(v, u) - \sum_{v \in \sigma_u} f(u, v) \right)$$

Subject to:

1. $0 \leq f(u, v) \leq c_{uv}$ for $(u, v) \in E$,
2. $\sum_{w \in \psi_u} f(w, u) \leq c_u^{in}$ for $u \in V$,
3. $\sum_{w \in \sigma_u} f(u, w) \leq c_u^{out}$ for $u \in V$,
4. $0 \leq \sum_{w \in \psi_u} f(w, u) - \sum_{w \in \sigma_u} f(u, w) \leq w_u$ for $u \in V - \{s\}$.

In the optimization objective,

$$\sum_{v \in \psi_u} f(v, u) - \sum_{v \in \sigma_u} f(u, v)$$

is the net number of tasks received (and processed) by node u in one unit of time. Because s does not need to receive tasks from other nodes, the computation capability of s is just an additive factor to the optimization objective. The optimization objective is therefore to maximize the total number of tasks processed by all the nodes in the system.

Condition 1 reflects the capacity constraints of the edges. In Condition 2, $\sum_{w \in \psi_u} f(w, u)$ is the total number of tasks

transferred to u . Condition 2 means that no node can receive tasks at a rate higher than what is allowed by its network interface. Similarly, Condition 3 limits the rate at which a node can send out tasks. In Condition 4, $\sum_{w \in \psi_u} f(w, u) - \sum_{w \in \sigma_u} f(u, w)$ is the net number of tasks that u has kept locally. Condition 4 means that any node (except s , which has the source data for all the tasks) cannot keep more tasks than it can compute; otherwise, the number of uncomputed tasks on this node will increase monotonically as time advances.

The Base Problem has a linear programming formulation. Because we are considering steady-state throughput, the Base Problem only needs to be solved in rational. We are not looking for integer-valued solutions. Various algorithms have been proposed to solve linear programming problems. These include the Simplex algorithm, which has excellent average case performance, and the interior point algorithms, which guarantee a polynomial execution time. However, these linear programming algorithms need to be executed by a central coordinator that knows all the parameters of the problem. Although parallel implementations of these algorithms do exist, the parallelism comes from the parallelization of the linear system solver, matrix inversion, matrix-vector multiplication, and so forth. The central coordinator still needs to know all the parameters of the problem before distributing the computations. The Centralized algorithm is not desirable for the distributed computing system we consider.

If an instance of the Base Problem has G as the input graph and s as the root node, we denote it as Base Problem (G, s) .

4 NETWORK FLOW REPRESENTATION FOR THE TASK ALLOCATION PROBLEM

Base Problem (G, s) can be transformed to a network flow representation using the following procedure. (In Section 5, a distributed and adaptive algorithm will be developed based on this representation.)

Graph Transformation:

1. Create an empty graph $G'(V', E')$.
2. Insert a node t into V' .
3. For each node u in Base Problem (G, s) , insert three nodes $u_1, u_2,$ and u_3 into V' . $u_1, u_2,$ and u_3 all have zero weight; insert edges (u_2, u_1) with capacity c_u^{in} and edges (u_1, u_3) with capacity c_u^{out} into E' . Insert edge (u_1, t) into E' and set its capacity to w_u .
4. For each edge (u, v) in Base Problem (G, s) , insert edge (u_3, v_2) into E' and set its capacity to c_{uv} .

The transformation results in a new graph G' . A hypothetical node t is first added to G' . Each node u in Base Problem (G, s) is then split into three nodes, $u_1, u_2,$ and u_3 , representing the processor, the input interface, and the output interface of u . Hypothetical edges from u_1 to t represent task executions at the processor of u . s_1 is the root node in G' . The transformation procedure is illustrated in Fig. 1. To simplify the figure, node weight and edge capacities are not marked.

After transforming the graph, we have the following flow maximization problem:

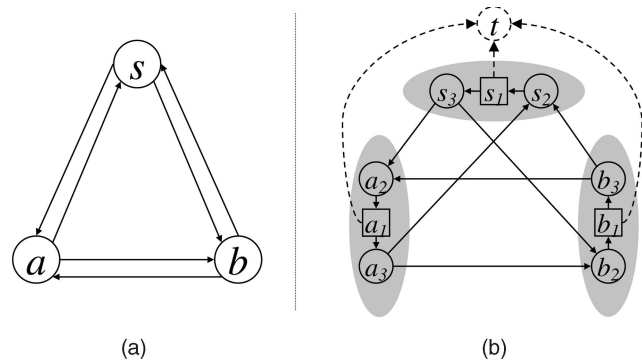


Fig. 1. Transform a base problem to a network flow representation. (a) A base problem with three nodes. (b) The corresponding network flow representation.

Problem 1: Given a directed graph $G(V, E)$ with root node s and sink node t . $s \neq t$. Edge (u, v) has capacity $c_{uv} > 0$.

Maximize:

$$\sum_{u \in \sigma_s} f(s, u) - \sum_{u \in \psi_s} f(u, s)$$

Subject to:

1. $0 \leq f(u, v) \leq c_{uv}$ for $(u, v) \in E$
2. $\sum_{v \in \sigma_u} f(u, v) = \sum_{v \in \psi_u} f(v, u)$ for $u \in V - \{s, t\}$

Problem 1 is the well-studied network flow problem. The objective is to maximize the amount of flow out of the root node without violating the edge capacity constraints. In the meantime, all the nodes except root s and sink t must have the same amount of incoming and outgoing flow.

Similarly, if an instance of Problem 1 has G as the input graph, s as the root node, and t as the sink node, we denote it as Problem 1 (G, s, t) . We further use $T_B(G, s)$ to denote the maximum throughput for Base Problem instance (G, s) and $T_1(G, s, t)$ to denote the maximum flow for Problem 1 instance (G, s, t) .

The next theorem shows that the Base Problem is a special case of Problem 1 after applying the graph transformation.

Theorem 1. Suppose that Base Problem (G, s) is converted to Problem 1 (G', s_1, t) by applying the abovementioned graph transformation. Then, $T_B(G, s) = T_1(G', s_1, t)$.

Proof. We use the notation in Problem 1 to denote the nodes and edges in G and the corresponding nodes and edges in G' .

Suppose that $f(u, v), (u, v) \in E$ is a feasible solution for Base Problem (G, s) . We map it to a feasible solution $f'(u', v'), (u', v') \in E'$ for Problem 1 (G', s_1, t) as follows:

1. $f'(u_3, v_2) \leftarrow f(u, v)$.
2. $f'(u_2, u_1) \leftarrow \sum_{w \in \psi_u} f(w, u)$.
3. $f'(u_1, u_3) \leftarrow \sum_{w \in \sigma_u} f(u, w)$.
4. $f'(u_1, t) \leftarrow f'(u_2, u_1) - f'(u_1, u_3)$.

It is easy to verify that such an f' is a feasible solution for Problem 1 (G', s_1, t) and that f' results in the same throughput as f .

Suppose that $f'(u', v'), (u', v') \in E'$ is a feasible solution for Problem 1 (G', s_1, t) . We map it to a feasible

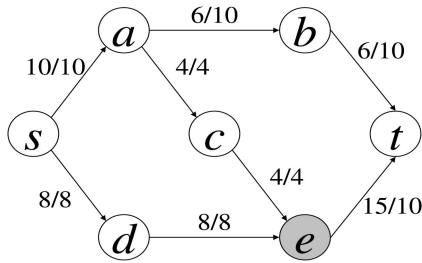


Fig. 2. An example of the relaxed flow maximization problem.

solution $f(u, v)$, $(u, v) \in E$ for Base Problem (G, s) using equation $f(u, v) = f'(u_3, v_2)$. It is also easy to verify that such an f is a feasible solution for Base Problem (G, s) and that it has the same throughput as f' . \square

Problem 1 is the well-studied network flow problem. Several algorithms [8] can be used to solve this problem, e.g., the Edmonds-Karp algorithm, which has $O(|V| \cdot |E|^2)$ complexity, the Push-Relabel algorithm, which has $O(|V|^2 \cdot |E|)$ complexity, and the Relabel-to-Front algorithm, which has $O(|V|^3)$ complexity.

However, in terms of decentralization and adaptivity, these well-known flow maximization algorithms are not suitable for distributed computing environments. Both the Edmonds-Karp and the Relabel-to-Front algorithms are centralized. The Push-Relabel algorithm has a decentralized implementation where every node only needs to exchange messages with its immediate neighbors and makes decisions locally. However, in order to be adaptive to the changes in the system, this algorithm has to be reinitialized and rerun from scratch each time when some parameters (edge capacities) of the flow maximization problem change. For each of the reruns, none of the nodes can start executing the push and relabel operations until all the nodes have finished the initialization process. In this case, there has to be a global controller that monitors the initialization of all the nodes and gives the nodes the “ok-to-start” signal. This again compromises the desired property of decentralization.

5 DECENTRALIZED ADAPTIVE TASK ALLOCATION ALGORITHM

In this section, we first show that the maximum flow remains the same even if Condition 2 in Problem 1 is relaxed. Then, we develop a distributed and adaptive algorithm for the relaxed problem.

5.1 Relaxed Flow Maximization Problem

Consider the example in Fig. 2. The flow and capacity of each edge is marked in the form of “flow/capacity.” Given the capacities of the edges, the maximum achievable flow is 18. In this example, node e has 12 units of incoming flow and 15 units of outgoing flow. Such a flow is not a feasible solution to Problem 1 since condition 2 is violated at node e . Suppose that the nodes form an actual system and 12 tasks have reached e , then e can send out no more than 12 tasks even if it is allowed to send out 15 tasks. The above observation can be generalized as follows: When a system is actually deployed, the number of tasks that a node can send

out is limited not only by the capacities of the edges, but also by the number of tasks that have been received. Intuitively, “allowing” a node to send out more tasks than what has been received will not affect the system throughput adversely.

This leads to the following relaxed network flow problem:

Relaxed Flow Problem: Given directed graph $G(V, E)$, source node $s \in V$, and sink node $t \in V$. Edge (u, v) has weight $c_{uv} \geq 0$.

Maximize:

$$\sum_{u \in V} f(s, u)$$

Subject to:

1. $f(u, v) \leq c_{uv}$ for $u \in V, v \in V$,
2. $f(u, v) = -f(v, u)$ for $u, v \in V$,
3. $\sum_{v \in V} f(v, u) \leq 0$ for $u \in V - \{s, t\}$.

In the problem statement, we have adopted the widely used notation for network flow problems: When the actual data transfer is from u to v , we define $f(v, u) = -f(u, v)$. Additionally, when edge $(u, v) \notin E$, we define $c_{uv} = 0$ and still enforce the capacity constraint on (u, v) . In this way, we can define $f(u, v)$ over $V \times V$ rather than being restricted to E . If neither (u, v) nor (v, u) belongs to E , then $c_{uv} = c_{vu} = 0$, which implies that $f(u, v) = f(v, u) = 0$ after enforcing edge capacity constraints. $f(v, u) = -f(u, v)$ also allows us to compute the total amount of flow into u as $\sum_{w \in V} f(w, u)$ (which is equal to $\sum_{w \in \psi_u \cup \sigma_u} f(w, u)$). Note that this expanded definition of $f(u, v)$ is for notational convenience only. It does not change the essence of the flow problem.

The Relaxed Flow Problem differs from Problem 1 in that the total flow out of a node can be equal to or larger than the total flow into the node (Condition 3). The objective is to maximize the total amount of flow out of root s . Notice that, by definition, a feasible solution to Problem 1 must also be a feasible solution to the Relaxed Flow Problem when the same input graph, source node, and sink node are given.

A feasible function f to the Relaxed Flow Problem is called a *relaxed flow* in graph G . We use $T_R(G, s, t)$ to denote the maximum throughput that flows out of node s in a relaxed flow problem with graph G , root s , and sink t . The following theorem shows the relation between the Relaxed Flow Problem and Problem 1:

Theorem 2. *Given graph $G(V, E)$, source s , and sink t , if f^* is an optimal solution to the Relaxed Flow Problem, then there exists an optimal solution f' to Problem 1 such that $0 \leq f'(u, v) \leq f^*(u, v)$ for each $f^*(u, v) > 0$. Additionally, $T_R(G, s, t) = T_1(G, s, t)$.*

Proof sketch. We first construct a new graph $G'(V, E')$ from $G(V, E)$ and f^* such that $(u, v) \in E'$ if and only if $f^*(u, v) > 0$, and we set the capacity of (u, v) in E' to $f^*(u, v)$ and the capacity of (v, u) to 0. Then, we execute any network flow maximization algorithm (for example, the Push-Relabel algorithm as in [8]), given G' as the input graph, s as the source node, and t as the sink node.

Suppose that the algorithm generates a solution f' . f' can be easily verified to be a valid solution to Problem 1.

We can show (details omitted here) that, for G' , dividing V into $\{s\}$ and $V - \{s\}$ is a minimum cut (refer to [8] for the definitions and properties of cut and minimum cut). According to the minimum-cut maximum-flow theorem ([8]), $\sum_{u \in V} f'(s, u)$ must be equal to the capacity of any minimum cut (all of which are equal). The capacity of the minimum cut $\{s\}$ and $V - \{s\}$ is $\sum_{u \in V} f^*(s, u)$, given the way G' is constructed. Therefore, we have $\sum_{u \in V} f'(s, u) = \sum_{u \in V} f^*(s, u)$.

Consequently, f' must be an optimal solution to Problem 1 because, otherwise, there must exist another solution f'' to Problem 1 such that $\sum_{u \in V} f''(s, u) > \sum_{u \in V} f'(s, u)$. By definition, f'' is also a relaxed flow. Remember that $\sum_{u \in V} f'(s, u) = \sum_{u \in V} f^*(s, u)$. This means that f'' has a larger flow out of the root node than f^* . This conflicts with the assumption that f^* is an optimal solution to the Relaxed Flow Problem.

Obviously, we have $0 \leq f'(u, v) \leq f^*(u, v)$ for each $f^*(u, v) > 0$ because f^* is the edge capacity for G' . \square

5.2 The Algorithm

The algorithm is an augmentation to the Push-Relabel algorithm and is denoted as the *Relaxed Incremental Push-Relabel* (RIPR) algorithm.

To explain the RIPR algorithm, we need two additional notations. For each node $u \in G$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the total amount of flow into node u . An integer-valued auxiliary function $h(u)$ is also defined for $u \in G$, which will be explained in the algorithm. The algorithm is described below:

1. Initialization phase: $h(u)$ and $f(u, v)$ are initialized as follows:

$$\begin{aligned} h(u) &\leftarrow 0 && \text{for } u \in V - \{s\}, \\ f(u, v) &\leftarrow 0 && \text{for } u \neq s \text{ and } v \neq s, \\ h(s) &\leftarrow |V|, \\ f(s, u) &\leftarrow c_{su} && \text{for } u \in V, \\ f(u, s) &\leftarrow -f(s, u) && \text{for } u \in V, \\ e(u) &\leftarrow \sum_{w \in V} f(w, u) && \text{for } u \in V. \end{aligned}$$

2. The initialization phase is executed only once (when the algorithm starts). After all the nodes complete the initialization phase, every node in the system except s and t execute the following two operations as long as $e(u) > 0$:

- a. *Push*(u, v): Applies when $e(u) > 0$ and $\exists v \in V$ such that $c_{uv} - f(u, v) > 0$ and $h(u) > h(v)$.

$$\begin{aligned} d &= \min(e(u), c_{uv} - f(u, v)), \\ f(u, v) &\leftarrow f(u, v) + d, \\ f(v, u) &\leftarrow -f(u, v), \\ e(u) &\leftarrow \sum_{w \in V} f(w, u), \\ e(v) &\leftarrow \sum_{w \in V} f(w, v). \end{aligned}$$

- b. *Relabel*(u): Applies when $e(u) > 0$ and $h(u) \leq h(v)$ for all $v \in \{v | c_{uv} - f(u, v) > 0\}$.

$$h(u) \leftarrow \min_{v \in \{v | c_{uv} - f(u, v) > 0\}} h(v) + 1.$$

3. Whenever the capacity of some edge changes, say, (\hat{u}, \hat{v}) changes from $c_{\hat{u}\hat{v}}$ to $c'_{\hat{u}\hat{v}}$, the following Adaptation (\hat{u}, \hat{v}) operation is executed:

- a. If $c'_{\hat{u}\hat{v}} > c_{\hat{u}\hat{v}}$ and $f(\hat{u}, \hat{v}) < c_{\hat{u}\hat{v}}$, do nothing.
- b. If $c'_{\hat{u}\hat{v}} > c_{\hat{u}\hat{v}}$ and $f(\hat{u}, \hat{v}) = c_{\hat{u}\hat{v}}$, then

$$\begin{aligned} h(s) &\leftarrow h(s) + 2|V|, \\ f(s, u) &\leftarrow c_{su} && \text{for } u \in V, \\ f(u, s) &\leftarrow -f(s, u) && \text{for } u \in V, \\ e(u) &\leftarrow \sum_{v \in V} f(v, u) && \text{for } u \in V. \end{aligned}$$

- c. If $c'_{\hat{u}\hat{v}} < c_{\hat{u}\hat{v}}$ and $f(\hat{u}, \hat{v}) \leq c'_{\hat{u}\hat{v}}$, do nothing.
- d. If $c'_{\hat{u}\hat{v}} < c_{\hat{u}\hat{v}}$ and $f(\hat{u}, \hat{v}) > c'_{\hat{u}\hat{v}}$, then

$$\begin{aligned} f(\hat{u}, \hat{v}) &\leftarrow c'_{\hat{u}\hat{v}}, \\ f(\hat{v}, \hat{u}) &\leftarrow -f(\hat{u}, \hat{v}), \\ h(s) &\leftarrow h(s) + 2|V|, \\ f(s, u) &\leftarrow c_{su} && \text{for } u \in V, \\ f(u, s) &\leftarrow -f(s, u) && \text{for } u \in V, \\ e(u) &\leftarrow \sum_{v \in V} f(v, u) && \text{for } u \in V. \end{aligned}$$

In the algorithm, $e(u)$ and $h(u)$ are the local variables maintained by u . Only u 's immediate neighbor nodes will query the value of $h(u)$ (to determine if a "push" or "relabel" needs to be executed). The "push" and "relabel" operations only change the local variables $e(u)$, $h(u)$, and $f(u, v)$ (where v is a neighbor of u). $f(u, v)$ is shared between u and v . The problem of maintaining a consistent image of a shared variable has already been addressed by many researches (e.g., distributed consistency protocols have been designed in [13] and [20]). In summary, both "push" and "relabel" operations can be executed in a localized fashion.

The adaptation operation changes $f(\hat{u}, \hat{v})$, which is local to \hat{u} and \hat{v} . The algorithm also increases $h(s)$ by $2|V|$ and sets the flow out of s to the edge capacities, regardless of the new capacity of (\hat{u}, \hat{v}) . Notifying s about the capacity change is indeed not a local operation. However, since all the tasks initially reside on the root node in our task allocation problem, it is reasonable to assume that every node can send a message to the root node. The RIPR algorithm assumes that s knows $|V|$, the total number of nodes in the system, which is the only global information that the RIPR algorithm needs to know.

We assume that the push and relabel operations are atomic. The adaptation (\hat{u}, \hat{v}) operation consists of two steps: 1) change $f(\hat{u}, \hat{v})$ to $c'_{\hat{u}\hat{v}}$ if $f(\hat{u}, \hat{v})$ was larger than $c'_{\hat{u}\hat{v}}$ and 2) increase $h(s)$ and $f(s, v)$ for all the neighbors of s . We assume that both steps are atomic, respectively. We also require that Step 1 is executed immediately after an edge capacity changes.

To simplify the analysis of the correctness and complexity of the RIPR algorithm, we assume that “push,” “relabel,” and “adaptation” are executed in serialized semantics, i.e., the operations are assumed to be executed one at a time. To implement the algorithm, however, the less restrictive concurrent execution model can be used. This can be achieved by using the alternator algorithm proposed in [13], which transforms a distributed algorithm with serial execution to one that assumes concurrent execution.

Specifically, the alternator algorithm in [13] satisfies three conditions. First, if a node has an enabled operation at some state, then no neighbor of that node has an enabled operation at the same state. This condition ensures that the concurrent execution is serializable. Second, along any concurrent execution, each operation is executed infinitely often. This condition ensures fairness among the nodes. Third, the alternator allows the maximal number of operations to be executed concurrently. The alternator algorithm is further improved in [20] to support the read/write atomicity, where a node can atomically read the state of its neighbor or write its own state but not both. By using an alternator algorithm, which prevents two neighbors from executing operations concurrently, atomicity of the operations is ensured. Because the operations are serialized by the alternator algorithm, possible deadlocks are eliminated.

Note that the execution of “push” and “relabel” operations starts after the initialization phase and continues at each node u as long as $e(u) > 0$. “Push” and “relabel” at one node may trigger “push” and “relabel” operations at other nodes. The “adaptation” operation increases the value of $h(s)$ and may increase some $e(u)$ to a positive value, which will also trigger new “push” and “relabel” operations to be executed. The algorithm executes as long as edge capacities keep changing. Assuming that the edge capacities eventually stop changing, the following theorem shows that eventually no push and relabel operations will need to be executed and the RIPR algorithm finds the maximum flow:

Theorem 3. *Given graph $G(V, E)$ with root s and sink t and assuming that no capacity changes occur after the n th adaptation operation, the number of “push” and “relabel” operations executed by the RIPR algorithm is bounded from above by $O(n^2 \cdot |V|^2 \cdot |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the graph. Additionally, the RIPR algorithm finds an optimal solution to the Relaxed Flow Problem when no “push” or “relabel” operation needs to be performed.*

The proof of the theorem is presented in the Appendix.

In the RIPR algorithm, edge capacities are allowed to change before an optimal solution is found, i.e., capacities can change while the nodes in the system are still performing “push” and “relabel” operations. The proof (in the Appendix) guarantees that, as long as the root node responds and the changes do not occur anymore, the RIPR algorithm will find an optimal solution with $O(n^2 \cdot |V|^2 \cdot |E|)$ “push” and “relabel” operations. When implementing the RIPR algorithm in an actual system, if multiple parameters change in the system simultaneously, Step 1 of the “adaptation” operation must be performed for each change immediately. However, the root node can perform

Step 2 of the adaptation operation (increases $h(s)$ by $2|V|$ and increases $f(s, u)$ to c_{su} for every neighbor node) only once after Step 1 of the adaptation has been performed for each of the changes. It can be proved that the RIPR algorithm still finds an optimal solution using this strategy. Lower numbers of “push” and “relabel” operations will be required even though, asymptotically, the number of operations remains the same.

The RIPR algorithm differs from the original Push-Relabel algorithm in two aspects. First, it solves the relaxed flow maximization problem instead of the standard flow maximization problem. Second, when some edge capacities have changed, the RIPR algorithm starts from the current values of $f(u, v)$ and searches for the new optimal solution. Such an *incremental* optimization means that the algorithm does not need to be reinitialized when adapting to edge capacity changes. Consequently, no global controller is needed to monitor the initialization process of all the nodes.

A similar network flow representation can be applied to problems in other areas. For example, when data gathering in sensor networks is subjected to energy constraints at the sensors, the maximum data-gathering problem can be formulated as a flow problem with capacity constraints at the nodes (in addition to edge capacity constraints, which represents the communication bandwidth). In [14], we show that the flow problem with node capacity constraints can be reduced to a standard network flow problem and, hence, a similar algorithm as mentioned above can be applied to data gathering problems in sensor networks.

6 ONLINE TASK ALLOCATION PROTOCOL

The RIPR algorithm can be approximated as a simple protocol to allocate the tasks. As discussed in Section 4, a computation node maps to three nodes in the network flow representation. Hence, in the protocol, each computation node needs to execute “push,” “relabel,” and “adaptation” for the three “hypothetical” nodes. For the discussion in the rest of this section, u is used to refer to either a computation node or a node in the network flow representation, which can be easily clarified given the context.

In this protocol, a task buffer is maintained by each computation node. By limiting the size of the task buffers, we can prevent any computation nodes from accumulating more tasks than they can compute. The buffers contain 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 $b(u)$ denote the length of the buffer at u . At any time instance, each computation node $u \in V$ operates as follows:

1. Contact the adjacent computation node(s) and perform the “push,” “relabel,” and “adaptation” operations, if necessary. By performing the operations, u can find the optimal rate $f(u, v)$ to transfer data to/from each neighbor v .
2. If $b(u) > 0$ and u is not computing any task, then remove one task from the task buffer, set $b(u) \leftarrow b(u) - 1$, and compute the task.
3. While $b(u) > 0$ and u is computing a task, send message “request to send” to each node v such that

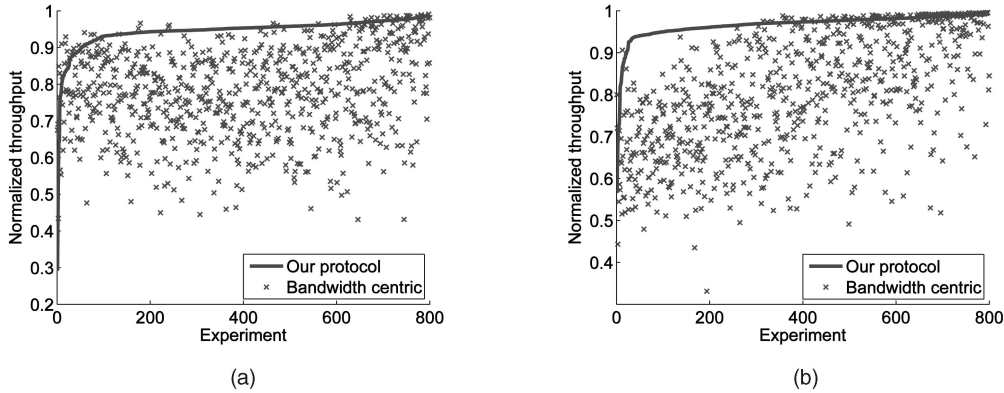


Fig. 3. Performance of the proposed task allocation protocol with uniformly distributed system topologies. The x -axis represents 800 experiments. (a) $w_{max} = 0.05$. (b) $w_{max} = 0.1$.

- $f(u, v) > 0$. If “clean to send” is received from v , then send a task to v at rate $f(u, v)$ and set $b(u) \leftarrow b(u) - 1$. To maximize the system throughput, u sends tasks at the rate determined by the RIPR algorithm, rather than utilizing the full capacity of its outgoing edges.
4. Upon receiving “request to send,” u acknowledges “clean to send” if $b(u) \leq U$. u acknowledges a denial if $b(u) > U$. Here, U is a preset threshold that limits the maximum number of tasks a task buffer can hold.

It should be pointed out that the flow rate control used in Step 3 of the abovementioned protocol (data is sent at a specific rate) is applicable. First of all, the RIPR algorithm guarantees that $f(u, v)$ never exceeds the bandwidth constraints of c_{uv} and the capability constraints of c_i^{out} and c_j^{in} , even before the optimization is completed. (In the case of a capacity change, $f(u, v)$ will be updated according to the new capacities as in the “adaptation” step of the RIPR algorithm.) Furthermore, rate control is supported by current network techniques and software development. (For example, many types of FTP software can impose the rate that data is transferred.)

7 EXPERIMENTAL RESULTS

Simulations were performed to evaluate the performance of the proposed task allocation protocol.

In the simulations, a graph is represented by its adjacency matrix A , where each nonzero entry a_{ij} represents the bandwidth of the corresponding link c_{uv} . Initially, all entries in A are set to 0. Then, a total number of $0.1 \times |V|^2$ randomly selected a_{ij} s are assigned values that are uniformly distributed between 0 and 1. If the graph is not connected with the $0.1 \times |V|^2$ edges just added, randomly chosen edges are then added to the graph, one edge at a time, until the graph is connected. c_i^{in} and c_i^{out} are also uniformly distributed between 0 and 1. w_u is uniformly distributed between 0 and w_{max} . Note that $1/w_{max}$ represents the average computation/communication ratio of a task. $w_{max} \geq 1$ represents a trivial scenario because the direct neighbors of the root node can consume, statistically, all the tasks flowing out of the root node and, hence, there is no need for other nodes to join the computation. The actual

value of w_{max} depends on the application. For example, in SETI@home, it takes about 5 minutes to receive a task through a modem and about 10 hours to compute a task on a current model home computer [18]. In our simulations, we used $w_{max} = 0.1$ and $w_{max} = 0.05$, which represent an average computation/communication ratio of 10 and 20, respectively. The network latencies were ignored when simulating the transfer of tasks.

In order to evaluate the performance of the proposed task allocation protocol, we compare it against the bandwidth-centric protocol proposed in [4]. In this protocol, when a node u can send tasks to multiple neighbors, it gives priority to the neighbor that has a higher network bandwidth, regardless of the computation power of the neighbors. It has been proved in [4] that this bandwidth-centric protocol assures maximum throughput for computing independent equal-sized tasks when the system is connected via a tree topology. This protocol can be adapted for any graph-structured system with the use of task buffers: Node u sends a task request to the neighbor with the highest network bandwidth when the task buffer at u becomes empty. When multiple requests are received simultaneously, the request from the neighbor with the highest network bandwidth is processed first.

For this set of simulations, the parameters c_{uv} , w_u , c_i^{in} , and c_i^{out} were set to constants; buffer size U , whose impact will be studied later, was temporarily set to 5. A total of 1,600 systems were simulated, 800 with $w_{max} = 0.05$ and 800 with $w_{max} = 0.1$, and there were 20 nodes in each of the systems. A node has an average of 2.3 neighbors. The average diameter of the systems is 4.1 hops. (We define the diameter of a system as the largest distance between any two nodes in the system, where the distance between two nodes is the length of the shortest path between the two nodes.) The average distance from the nodes to the root is 2.6 hops. Initially, there were 2,500 tasks on node s . The achieved throughput is calculated as $2,500/t_{all}$, where t_{all} is the overall computation time of all tasks (from the start of the computation until the last task finishes execution; note that the tasks are executed concurrently at the nodes). The results in Fig. 3 compare the performance of our task allocation protocol and the bandwidth-centric protocol. In Fig. 3, the throughput of the two protocols has been

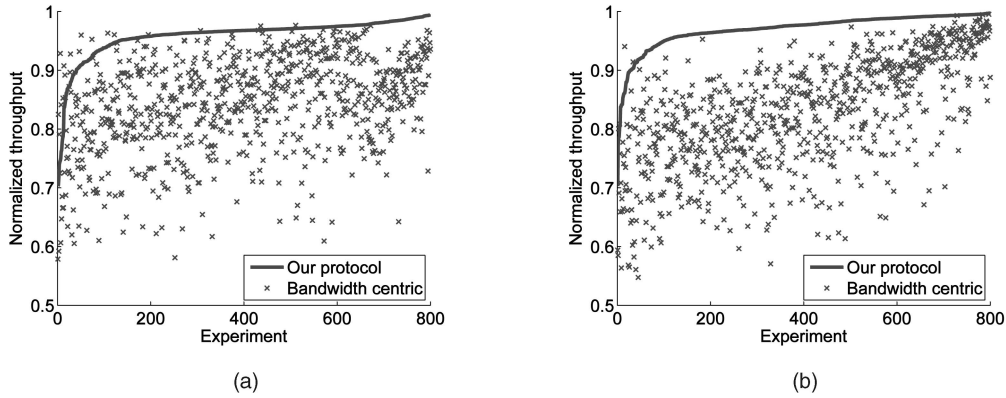


Fig. 4. Performance of the proposed task allocation protocol with power law distributed system topologies. The x -axis represents 800 experiments. (a) $w_{max} = 0.05$. (b) $w_{max} = 0.1$.

normalized to the optimal throughput (calculated offline). The results are shown in increasing order of the normalized throughput by using our task allocation protocol. When $W_{max} = 0.05$, our protocol achieves a normalized throughput of 0.945 with standard deviation 0.046; the bandwidth-centric protocol achieves a normalized throughput of 0.7815 with standard deviation 0.121. When $W_{max} = 0.1$, our protocol achieves a normalized throughput of 0.967 with standard deviation 0.034; the bandwidth-centric protocol achieves a normalized throughput of 0.807 with standard deviation 0.141. Compared with the bandwidth-centric protocol, our protocol achieves a close-to-optimal system throughput with a smaller standard deviation.

Fig. 3 demonstrates the performance of the proposed method when the system topology is represented by a uniformly distributed random matrix. We also simulate more practical scenarios where computation resources are connected via the Internet. Empirical studies [11] have shown that the Internet topologies exhibit power law distributions. In this set of simulations, the graph representations of the systems were generated using Brite, which is a tool developed in [23] that generates networks with power law characteristics. We used the following parameters for Brite: each system has 20 nodes, the topology was generated using the “Barabasi-Albert” model at the Autonomous System level, node placement was set to “random,” growth type was set to “incremental,” preferred connection was set to “none,” bandwidth of the edges was uniformly distributed between 0 and 1, and each new node connected to three existing nodes when added to the system. Brite does not generate c_{uv} , c_i^n , c_i^{out} , and w_u , so we generated their values separately. The values of c_{uv} , c_i^n , and c_i^{out} were uniformly distributed between 0 and 1. w_u was uniformly distributed between 0 and w_{max} . U was set to 5. s has 2,500 tasks initially. We simulated 900 systems with $w_{max} = 0.05$ and another 900 systems with $w_{max} = 0.1$. Fig. 4 compares the performance of our protocol against the bandwidth-centric protocol. When $W_{max} = 0.05$, our protocol achieves a normalized throughput of 0.959 with standard deviation 0.036; the bandwidth-centric protocol achieves a normalized throughput of 0.842 with standard deviation 0.08. When $W_{max} = 0.1$, our protocol achieves a

normalized throughput of 0.970 with standard deviation 0.031; the bandwidth-centric protocol achieves a normalized throughput of 0.827 with standard deviation 0.095. The result shows that our protocol achieves a close-to-optimal system throughput, regardless of the system topologies. The bandwidth-centric protocol achieves higher throughput for power law distributed systems than for uniformly distributed systems. However, it cannot ensure a close-to-optimal throughput.

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 computation 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 the second set of simulations, we study the impact of the buffer size on the performance of the system. To simplify the discussions, we assume that the task buffers at all the nodes have the same size U .

We simulated 120 systems, each having 40 nodes, and another 120 systems, each having 80 nodes. w_{max} was set to 0.05. U was set to 15. No changes occurred to the system during the simulations; hence, no adaptations were activated. For each system, there were 2,500 tasks on root node s initially, and all the nodes in the system have infinite task buffers. In the experiments, the 40-node systems achieved an average of 0.92 of the optimal throughput; the 80-node system achieved an average of 0.94 of the optimal throughput. We monitored the maximum buffer consumed by each individual computation node and the results are summarized in Fig. 5 in the form of a histogram. The histogram in Fig. 5a is computed over all the 40-node systems (200 such systems in total), and the histogram in Fig. 5b is computed over all the 80-node systems (200 such systems in total).

The results in Fig. 5 show that the 40-node systems rarely need buffers larger than 10 and the 80-node systems rarely need buffers larger than 6. Other values of w_{max} were tested, and a buffer requirement between 5 and 10 was observed.

Nodes in the 40-node systems had an average number of 3.9 neighbors; nodes in the 80-node systems had an average

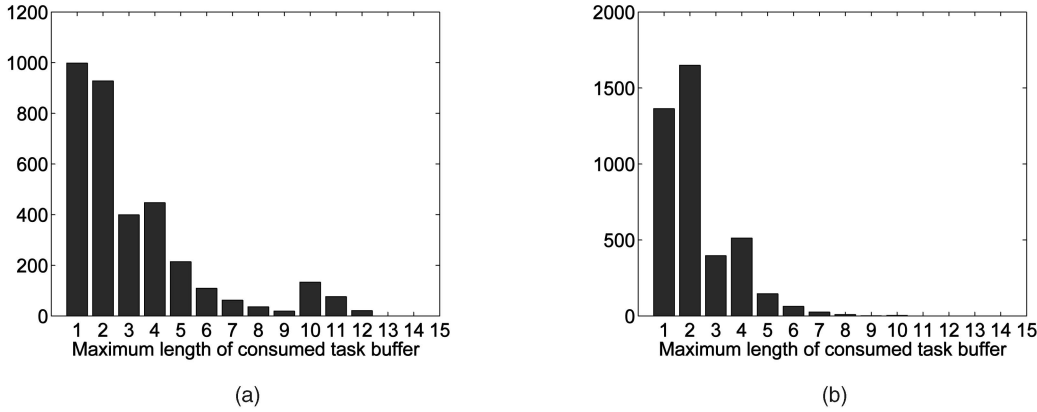


Fig. 5. Histogram of the maximum length of consumed task buffer. The y -axis represents the frequency. (a) 40-node systems, 200 experiments. (b) 80-node systems, 200 experiments.

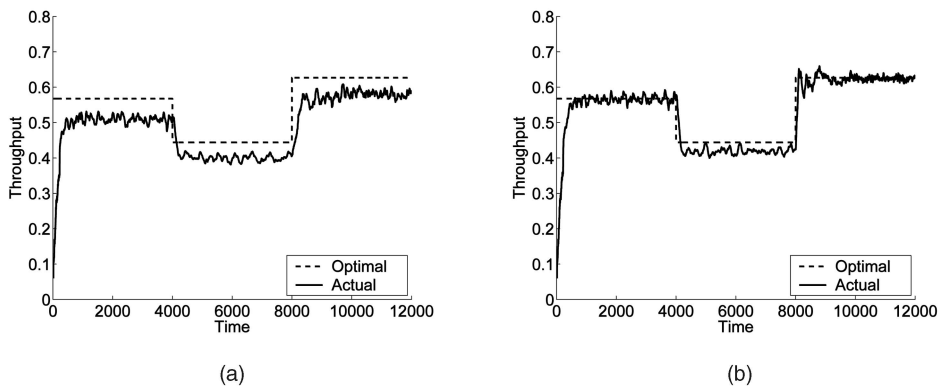


Fig. 6. Adaptation to changes in the first system simulation. (a) $U = 1$. (b) $U = 5$.

number of 7.9 neighbors. This shows that the maximum buffer usage does not seem to be directly related to the node degrees. Although we believe that the maximum buffer usage is determined by the system topology, we have not been able to determine a simple relation between the maximum buffer usage and the parameters (for example, node degrees) extracted from the system topology.

As claimed in Sections 5 and 6, adaptivity is an important property of our task allocation protocol. This is illustrated in the next two simulations.

In the first simulation, the system consists of 20 nodes. The adjacency matrix was initialized using uniform distribution. $w_{max} = 0.1$. However, during the course of the simulation, the network condition and the effective computation power of the nodes were altered at two time instances. At time instance $t = 4,000$, the computation power of a selected set of computation nodes was reduced by 80 percent. Then, at $t = 8,000$, these computation nodes recovered their computation power, while at the same time, the computation power of another set of nodes was increased by 40 percent and the bandwidth of a selected set of links was increased by 30 percent.

In Fig. 6, the optimal throughput was calculated off-line. It indicates the maximum throughput that can be achieved by the system. The instantaneous throughput actually achieved at time instance t was approximated 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 . The

size of this moving window, 150, is selected experimentally, as a trade-off between preserving the details and describing the trend. When $t < 75$, the instantaneous throughput was calculated as $N(t)/t$.

As illustrated in Fig. 6, our task allocation adapts to the changes in the system and approaches the optimal throughput during the course of the computation. We simulated $U = 1$ and $U = 5$ for the threshold U on the buffer size. Our task allocation exhibits similar adaptivity for both values of U . When the system parameters changed at $t = 4,000$ and $t = 8,000$, the adaptation procedure was activated and the task allocation was adapted. As can be seen, the system operates at (close to) the new optimal throughput after the adaptation was completed. In Fig. 6, at some time instances, the actual system throughput exceeds the optimum. This is because the size of the moving window is not wide enough. The impact of U is similar to the static scenario: $U = 5$ leads to a higher and closer to optimal throughput than $U = 1$. We have also observed that the benefit of further increasing the value of U , not surprisingly, becomes marginal as U gets larger.

In the second simulation, the system consists of 20 nodes. The adjacency matrix was also initialized using uniform distribution. $w_{max} = 0.1$. During the simulation, the bandwidth of a set of network links was each decreased by 40 percent at time instance $t = 4,000$. In Fig. 7, we illustrate the performance of our protocol when responding to the

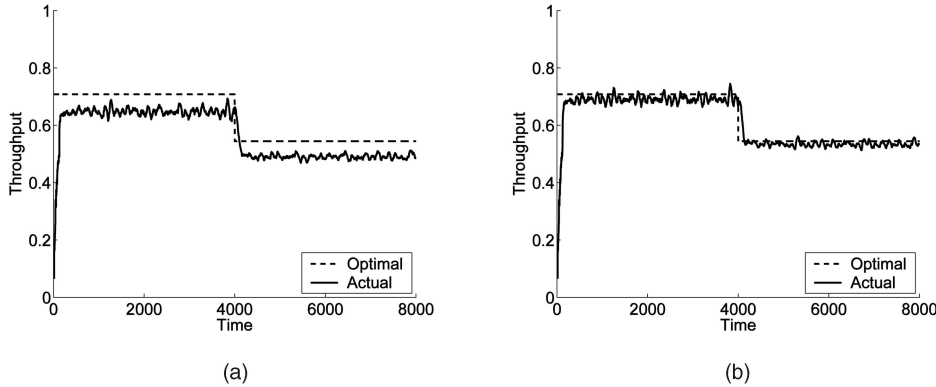


Fig. 7. Adaptation to changes in the second system simulation. (a) $U = 1$. (b) $U = 5$.

changes. The illustrated optimal throughput was calculated offline. The instantaneous throughput achieved by our protocol was calculated in the same way as in Fig. 6. A performance similar to Fig. 6 can be observed in Fig. 7.

8 DISCUSSION

In this paper, we studied the computation of a large set of independent tasks in a heterogeneous system. We modeled the computation at the nodes as a special type of data flow. We then transformed the throughput maximization problem to a network flow maximization problem. We further developed the RIPR algorithm for the relaxed form of the network flow maximization problem. Based on the algorithm, we developed a decentralized and adaptive protocol to allocate the tasks. Simulation results showed that the protocol achieves a close-to-optimal throughput. Furthermore, simulation results showed that the protocol can adapt to the changes in the system and maintain the throughput at close to optimal level.

The system model in this paper considers a simple scenario where the overhead of overlapping computation and communication is ignored. As pointed out in [19], the actual computation capability of a computer may reduce if the computer is involved intensively in performing communications. This study further shows that there is a (close-to) linear relation between the decrease in computation capability and the increase in communication activity. To reflect the impact of such an overhead, we need to add two more parameters, α_u and β_u , for each node u , representing the impact of receiving and sending on computation, respectively. We have the new problem formulation as follows:

Given: A directed graph $G(V, E)$. Node $u \in V$ has weight $w_u > 0$, input constraint $c_u^{in} > 0$, output constraint $c_u^{out} > 0$, input impact factor $\alpha_u \geq 0$, and output impact factor $\beta_u \geq 0$. Edge (u, v) has capacity constraint $c_{uv} > 0$. s is the distinguished root node.

Maximize:

$$w_s + \sum_{u \in V - \{s\}} \left(\sum_{v \in \psi_u} f(v, u) - \sum_{v \in \sigma_u} f(u, v) \right)$$

Subject to:

1. $0 \leq f(u, v) \leq c_{uv}$ for $(u, v) \in E$,
2. $\sum_{w \in \psi_u} f(w, u) \leq c_u^{in}$ for $u \in V$,
3. $\sum_{w \in \sigma_u} f(u, w) \leq c_u^{out}$ for $u \in V$,
4. $0 \leq \sum_{w \in \psi_u} f(w, u) - \sum_{w \in \sigma_u} f(u, w) \leq w_u - \alpha_u \sum_{w \in \psi_u} f(w, u) - \beta_u \sum_{w \in \sigma_u} f(u, w)$ for $u \in V - \{s\}$.

In Condition 4, the computation capability w_u of node u decreases as u is involved in sending and receiving data. The decrease in the computation capability is determined by factors α_u and β_u , as well as the amount of incoming and outgoing flow.

w_u can be regarded as the capacity of node u . w_u is shared among computing, sending, and receiving. However, computing, sending, and receiving consume the capacity w_u at different rates (1 for computing, α_u for receiving, and β_u for sending). This problem can be transformed to a network flow problem with node capacity constraints where the weighted sum of all the flows going through a node is limited by the capacity of the node. Further investigation is needed to solve this problem in a distributed environment.

APPENDIX A

In the following, we prove Theorem 3.

Before presenting the proof, we first briefly restate some notations widely accepted for network flow problems. Given a direct graph $G(V, E)$, function f is called a flow if it satisfies the three constraints in the statement of Problem 1. Given $G(V, E)$ and flow f , the residual capacity $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the residual network of G induced by f is $G_f(V, E_f)$, where $E_f = \{(u, v) | u \in V, v \in V, c_f(u, v) > 0\}$. If a *Push*(u, v) operation removes (u, v) from E_f (i.e., $c_f(u, v) = 0$ after the operation), it is a *saturated push*; otherwise, it is a *nonsaturated push*.

Given a direct graph $G(V, E)$, a feasible function f to the Relaxed Flow Problem is called a *relaxed flow* in graph G .

Lemma 1. *During the execution of the RIPR Algorithm, for any $u \in V$, $h(u)$ never decreases.*

Proof. $h(s)$ is only changed by the Adaptation operation, during which $h(s)$ is increased by $2|V|$. When $u \neq s$, $h(u)$ is only changed by $Relabel(u)$. $Relabel(u)$ is applied when $e(u) > 0$ and $h(u) \leq h(w)$ for $w \in \{w | c_{uw} - f(u, w) > 0\}$. In addition, $h(u) = \min_{w \in \{w | c_{uw} - f(u, w) > 0\}} h(w) + 1$ after $Relabel(u)$. Hence, $h(u)$ is increased at least by 1. \square

Lemma 2. *During the execution of the RIPR Algorithm, for each $u \in V$ such that $e(u) > 0$, there exists a simple path in E_f from u to a node v such that $e(v) < 0$.*

Proof. Suppose that $e(u) > 0$. Define $\hat{V} = \{w | w \in V \text{ and there exists a simple path from } u \text{ to } w\}$. Note that $u \in \hat{V}$. We also define $\bar{V} = V - \hat{V}$.

For the sake of contradiction, suppose that $e(w) \geq 0$ for any $w \in \hat{V}$.

We claim that, if $x \in \bar{V}$ and $y \in \hat{V}$, then $f(x, y) \leq 0$. Otherwise, if $f(x, y) > 0$, then

$$c_f(y, x) = c_{yx} - f(y, x) = c_{yx} + f(x, y) > 0,$$

which means x can be reached from y in E_f ; hence, there exists a path from u to x in E_f . However, this contradicts the choice that $x \in \bar{V}$.

It is fairly easy to show that

$$\sum_{w \in \hat{V}} e(w) = \sum_{x \in \bar{V}, y \in \hat{V}} f(x, y).$$

Hence, $\sum_{w \in \hat{V}} e(w) \leq 0$. However, this contradicts the assumption that $e(w) \geq 0$ for each $w \in \hat{V}$ and $e(u) > 0$. \square

Lemma 3. *During the execution of the RIPR algorithm, for $u \in V$, if $e(u) > 0$, then either $Relabel(u)$ can be applied or there exists a node $v \in V$ such that $Push(u, v)$ can be applied.*

Proof. $e(u) > 0$ means that node u has more incoming flow than out going flow, which further means that there exists at least one node w such that $f(w, u) > 0$. Obviously, for this node w , we have $c_{uw} - f(u, w) = c_{uw} + f(w, u) > 0$, which means that $\{w | c_{uw} - f(u, w) > 0\} \neq \emptyset$.

If $\exists v \in V$ such that $c_{uv} - f(u, v) > 0$ and $h(u) > h(v)$, then $Push(u, v)$ can be applied; otherwise, $h(u) \leq h(v)$ for each $v \in \{v | c_{uv} - f(u, v) > 0\}$, which means $Relabel(u)$ can be applied.

Of course, when there does not exist any node v such that $c_{uv} - f(u, v) > 0$ and $h(u) > h(v)$, then we must also have $c_{uv} - f(u, v) \leq 0$ for each v such that $h(u) > h(v)$. However, we are not interested in this scenario. \square

Lemma 4. *During the execution of the Incremental Push-Relabel Algorithm, after the initialization phase, and if no adaptation can be applied, then*

$$\begin{aligned} h(u) &\leq \max(h(v) + 1, h(s) - |V| - 1) \text{ for each } (u, v) \in E_f, \\ h(u) &\leq h(s) + |V| - 1 \text{ for each } u \in V. \end{aligned}$$

Proof. We prove by induction on the number of adaptation operations.

- **Base case.** Before any changes occur in the system, no adaptation operation can be applied. At this stage, the RIPR algorithm performs the exact operations as the Push-Relabel algorithm (this can be seen easily by comparing the details of the push and relabel operations executed by

the two algorithms). For the Push-Relabel algorithm, it has been proved that

$$\begin{aligned} h(u) &\leq h(v) + 1 \text{ for each } (u, v) \in E_f, \\ h(u) &\leq 2|V| - 1 \text{ for each } u \in V. \end{aligned}$$

At this stage, we have $h(s) = |V|$ for the RIPR algorithm. Consequently, we have

$$\begin{aligned} h(u) &\leq \max(h(v) + 1, h(s) - |V| - 1) \text{ for each } (u, v) \in E_f, \\ h(u) &\leq h(s) + |V| - 1 \text{ for each } u \in V, \end{aligned}$$

for the RIPR algorithm before any adaptation operation is applied.

- **Induction step.** Suppose that the adaptation has been applied $n - 1$ times, and we have

$$\begin{aligned} h(u) &\leq \max(h(v) + 1, h(s) - |V| - 1) \text{ for each } (u, v) \in E_f, \\ h(u) &\leq h(s) + |V| - 1 \text{ for each } u \in V. \end{aligned}$$

Then, a new capacity change occurs on edge (\hat{u}, \hat{v}) and the n th adaptation, $Adaptation(\hat{u}, \hat{v})$, is applied.

1. We first show that $h(u) \leq \max(h(v) + 1, h(s) - |V| - 1)$ for each $(u, v) \in E_f$ after $Adaptation(\hat{u}, \hat{v})$.

Considering $Adaptation(\hat{u}, \hat{v})$, if either Scenario a or c occurs, no residual edge is added or removed, and no node $w \in V$ has its $h(w)$ changed either. If Scenario d occurs, the change in the system removes (\hat{u}, \hat{v}) from E_f and, hence, the corresponding constraint on $h(\hat{u})$ and $h(\hat{v})$. If Scenario b occurs, (\hat{u}, \hat{v}) is added to the E_f . By the induction assumption, $h(\hat{u}) \leq h(s) + |V| - 1$ before the adaptation operation. Because $h(\hat{u})$ does not change and $h(s)$ increases by $2|V|$ after the operation, $h(\hat{u}) \leq h(s) - 2|V| - 1 < h(s) - |V| - 1$ after the adaptation operation. In summary, after the adaptation operation, $h(u) \leq \max(h(v) + 1, h(s) - |V| - 1)$ for each $(u, v) \in E_f$.

$Adaptation(\hat{u}, \hat{v})$ changes the values of some $e(w)$, allowing new push and relabel operations to be applied. Nevertheless, these operations preserve the property that $h(u) \leq \max(h(v) + 1, h(s) - |V| - 1)$ for each $(u, v) \in E_f$. This is shown below:

- a. Suppose that $Push(u, v)$ is applied. This may add edge (v, u) into E_f or remove edge (u, v) from E_f . In the former case, we have $h(v) < h(u)$ because, otherwise, the push will not be applied. In the latter case, the removal of (u, v) from E_f removes the corresponding constraint on $h(u)$ and $h(v)$. In both cases, we still have $h(u) \leq \max(h(v) + 1, h(s) - |V| - 1)$ for any $(u, v) \in E_f$.
- b. Suppose that $Relabel(u)$ is applied. For a residual edge (u, v) that leaves u , we have $h(u) = \min_{w \in \{w | (u, w) \in E_f\}} h(w) + 1$ after the Relabel operation, which means $h(u) \leq h(v) + 1$. For a residual edge (w, u)

that enters u , $h(w) \leq \max(h(u) + 1, h(s) - |V| - 1)$ before the relabel operation. According to Lemma 1, $h(w) \leq \max(h(u) + 1, h(s) - |V| - 1)$ after the relabel operation. Therefore, after a relabel operation, we have $h(u) \leq \max(h(v) + 1, h(s) - |V| - 1)$ for any $(u, v) \in E_f$.

2. Now, we need to show that $h(u) \leq h(s) + |V| - 1$ for each $u \in V$.

Let \hat{V} denote the set of $u \in V$ such that there exists a simple path from u to s in E_f . $\bar{V} = V - \hat{V}$.

- a. For any node $u \in \hat{V}$, suppose that the simple path to s in E_f is $\{u, u_1, \dots, u_k\}$, where q and $k \leq |V| - 1$. We have

$$\begin{aligned} h(u) &\leq \max(h(u_1) + 1, h(s) - |V| - 1), \\ h(u_1) &\leq \max(h(u_2) + 1, h(s) - |V| - 1), \\ &\dots \\ h(u_{k-1}) &\leq \max(h(u_k) + 1, h(s) - |V| - 1). \end{aligned}$$

Combining these inequalities, we have

$$\begin{aligned} h(u) &\leq \max(h(u_k) + k, h(s) - |V| - 1 + k - 1), \\ &= \max(h(s) + k, h(s) - |V| + k - 2), \\ &\leq \max(h(s) + |V| - 1, h(s) - 3), \\ &= h(s) + |V| - 1. \end{aligned}$$

- b. For any node $u \in \bar{V}$, according to Lemma 2, there exists a simple path in E_f from u to a node w such that $e(w) < 0$ and $w \neq s$. Suppose that the simple path is $\{u, u_1, \dots, u_k\}$, where $u_k = w$ and $k \leq |V| - 1$. We have

$$\begin{aligned} h(u) &\leq \max(h(u_1) + 1, h(s) - |V| - 1), \\ h(u_1) &\leq \max(h(u_2) + 1, h(s) - |V| - 1), \\ &\dots \\ h(u_{k-1}) &\leq \max(h(u_k) + 1, h(s) - |V| - 1). \end{aligned}$$

Note that $e(u) \geq 0$ immediately after the initialization for each $u \in V$. The only operation that can bring $e(u)$ below 0 is the adaptation operation (when Scenario d occurs). Suppose that the most recent time $e(w)$ switches from positive to negative is during the m th adaptation operation ($m \leq n$). The Relabel operation (which is the only operation that can increase the value of $h(w)$) is applied only if $e(w) > 0$. However, $e(w) < 0$ has been true since the m th adaptation, which means that $h(w)$ has not been increased after the m th and, hence, the n th adaptation operation. Therefore, $h(w) \leq h(s) + |V| - 1$ (already proved) before the n th adaptation means that $h(w) \leq h(s) - |V| - 1$ thereafter since $h(s)$ is increased by $2|V|$ after the adaptation.

Combining these inequalities, we have

$$\begin{aligned} h(u) &\leq \max(h(u_k) + k, h(s) - |V| - 1 + k - 1) \\ &= \max(h(w) + k, h(s) - |V| - 2 + k) \\ &\leq \max(h(s) - |V| - 1 + |V| - 1, h(s) - |V| \\ &\quad - 2 + |V| - 1) \\ &\leq h(s) + |V| - 1. \end{aligned}$$

Since $\hat{V} \cup \bar{V} = V$, we claim that $h(u) \leq h(s) + |V| - 1$ for any $u \in V$. \square

Corollary 1. *During the execution of the RIPR algorithm, after the initialization phase, and if no adaptation can be applied, then for any node $u \in V$, $e(u) < 0$ implies that $h(u) \leq h(s) - |V| - 1$.*

The proof of Corollary 1 is included in the proof of Lemma 4.

Lemma 5. *During the execution of the Incremental Push-Relabel algorithm, after the initialization phase, and if no adaptation can be applied, then there is no path from s to t in E_f .*

Proof. For the sake of contradiction, suppose that there exists a path $\{s, u_1, \dots, u_k, t\}$ in E_f . Without loss of generality, this is a simple path, and $k \leq |V| - 2$. According to Lemma 4,

$$\begin{aligned} h(s) &\leq \max(h(u_1) + 1, h(s) - |V| - 1), \\ h(u_1) &\leq \max(h(u_2) + 1, h(s) - |V| - 1), \\ &\dots \\ h(u_{k-1}) &\leq \max(h(u_k) + 1, h(s) - |V| - 1), \\ h(u_k) &\leq \max(h(t) + 1, h(s) - |V| - 1). \end{aligned}$$

Hence,

$$h(s) \leq \max(h(t) + k + 1, h(s) - |V| - 1 + k).$$

Since node t is never relabeled, $h(t)$ remains 0. Therefore,

$$h(s) \leq \max(|V| - 1, h(s) - 3).$$

This is impossible because $h(s) > |V| - 1$ and $h(s) > h(s) - 3$. \square

Lemma 6. *During the execution of the RIPR algorithm, after the initialization phase, and if no adaptation can be applied, then for each $u \in V - \{s, t\}$ such that there exists a simple path from s to u in E_f , $e(u) \geq 0$.*

Proof. Immediately after the initialization phase, $e(u) \geq 0$ for each $u \in V - \{s, t\}$; the lemma holds trivially.

During the execution of the algorithm, when no adaptation can be applied, one of the following two conditions must be occurring: 1) no edge capacities have ever changed and 2) some edge capacities changed, the corresponding adaptations have been performed, and no further edge capacity changes have occurred yet. In both conditions, $f(s, u)$ is first set to c_{su} . (In Condition 1, the value setting is performed in the initialization phase; in Condition 2, the value setting is performed in the

adaptation operations). The value of $f(s, u)$ is then (possibly) modified by some push operations either after the initialization phase or after the adaptation operations.

Suppose for the sake of contradiction that there exists a node $u \in V - \{s, t\}$ such that $e(u) < 0$ and there exists a simple path $\{s, u_1, \dots, u_k, u\}$ in E_f . Without loss of generality, this is a simple path, and $k \leq |V| - 2$.

According to Lemma 4,

$$\begin{aligned} h(u_1) &\leq \max(h(u_2) + 1, h(s) - |V| - 1), \\ &\dots \\ h(u_{k-1}) &\leq \max(h(u_k) + 1, h(s) - |V| - 1), \\ h(u_k) &\leq \max(h(u) + 1, h(s) - |V| - 1). \end{aligned}$$

According to Corollary 1, $h(u) \leq h(s) - |V| - 1$ since $e(u) < 0$. Combining these inequalities, we can see that

$$\begin{aligned} h(u_1) &\leq \max(h(u) + k, h(s) - |V| - 2 + k) \\ &\leq \max(h(s) - |V| - 1 + k, h(s) - |V| - 2 + k) \\ &\leq h(s) - |V| - 1 + |V| - 2 \\ &< h(s). \end{aligned}$$

On the other hand, consider the first hop (s, u_1) along this path. $(s, u_1) \in E_f$ implies that $f(s, u_1) < c_{su_1}$. Recall that the value of $f(s, u_1)$ is set to c_{su_1} immediately after the initialization phase and each adaptation operation. The only operation that can reduce the value of $f(s, u_1)$ is a push from u_1 to s . However, $Push(u_1, s)$ is applied only when $h(u_1) > h(s)$. This contradicts the claim $h(u_1) < h(s)$ that we just derived. \square

Similar to the standard flow problem, for the Relaxed Flow Problem, a *cut* is defined as a binary partition (L, R) of V such that $L \cup R = V$, $L \cap R = \emptyset$, $s \in L$, and $t \in R$. The capacity c_{LR} of a cut (L, R) is defined as $c_{LR} = \sum_{u \in L, v \in R} c_{uv}$. The next lemma shows that the value of a relaxed flow cannot exceed the capacity of any cuts.

Lemma 7. *Given graph $G(V, E)$ with source s and sink t , a relaxed flow f , and an arbitrary cut (L, R) of G , $\sum_{u \in V} f(s, u) \leq c_{LR}$.*

Proof. We have $e(u) \leq 0$ for $u \in V - \{s, t\}$. Therefore,

$$\begin{aligned} \sum_{u \in L - \{s\}} e(u) &= \sum_{v \in V, u \in L - \{s\}} f(v, u) \leq 0 \\ \Rightarrow \sum_{v \in L, u \in L - \{s\}} f(v, u) &+ \sum_{v \in R, u \in L - \{s\}} f(v, u) \leq 0 \\ \Rightarrow \sum_{u \in L - \{s\}} f(s, u) &+ \sum_{v \in L - \{s\}, u \in L - \{s\}} f(v, u) \\ &+ \sum_{v \in R, u \in L - \{s\}} f(v, u) \leq 0 \\ \Rightarrow \sum_{u \in L - \{s\}} f(s, u) &+ \sum_{v \in L - \{s\}, u \in L - \{s\}} f(v, u) \\ &\leq \sum_{v \in R, u \in L - \{s\}} f(v, u) \\ \Rightarrow \sum_{u \in L - \{s\}} f(s, u) &\leq \sum_{v \in R, u \in L - \{s\}} f(v, u) \\ \Rightarrow \sum_{u \in L - \{s\}} f(s, u) &+ \sum_{u \in R} f(s, u) \leq \sum_{v \in R, u \in L - \{s\}} f(v, u) \\ &+ \sum_{u \in R} f(s, u) \\ \Rightarrow \sum_{u \in V - \{s\}} f(s, u) &\leq \sum_{u \in L, v \in R} f(v, u). \end{aligned}$$

Since $f(u, v) \leq c_{uv}$ for each $u, v \in V$, additionally, because $f(s, s) = 0$, we have

$$\sum_{v \in V} f(s, v) = \sum_{u \in V - \{s\}} f(s, u) \leq \sum_{u \in L, v \in R} c_{uv} = c_{LR}.$$

\square

Lemma 7 states a property of the relaxed flow (not a property of the RIPR algorithm). This property is used to prove Lemma 8, which shows that the RIPR algorithm will find the maximum relaxed flow if no ‘‘push’’ and ‘‘relabel’’ operations can be applied, assuming that capacities do not change any more. After proving Lemma 8, we will show that the number of ‘‘push’’ and ‘‘relabel’’ operations is indeed bounded from above.

Lemma 8. *If no capacity changes after the last adaptation operation and none of the nodes need to perform either the ‘‘push’’ or ‘‘relabel’’ operation, the RIPR finds the maximum relaxed flow.*

Proof. According to Lemma 3, if the algorithm terminates, then $e(u) \leq 0$ for each $u \in V - \{s, t\}$. Hence, f is a flow if the algorithm terminates.

Given such an f , we construct a cut of G as follows:

$$\begin{aligned} L &= \{u \in V \mid \text{there exists a simple path from } s \text{ to } u \text{ in } E_f\}, \\ R &= V - L. \end{aligned}$$

According to Lemma 6, $e(u) \geq 0$ for $u \in L - \{s\}$. Note that $e(u) \leq 0$ for each $u \in V - \{s, t\}$ upon termination of the algorithm. Hence, $e(u) = 0$ (that is, $\sum_{v \in V} f(v, u) = 0$) for each $u \in L - \{s\}$. Then, it is easy to show that $\sum_{u \in L} f(s, u) = \sum_{u \in L - \{s\}} f(s, u) = \sum_{u \in L - \{s\}, v \in R} f(u, v)$.

Therefore,

$$\begin{aligned} \sum_{v \in V} f(s, v) &= \sum_{v \in L} f(s, v) + \sum_{v \in R} f(s, v) \\ &= \sum_{u \in L - \{s\}, v \in R} f(u, v) + \sum_{v \in R} f(s, v) \\ &= \sum_{u \in L, v \in R} f(u, v). \end{aligned}$$

We claim that $f(u, v) = c_{uv}$ for each $u \in L$ and $v \in R$ because, otherwise, $f(u, v) < c_{uv}$ implies that edge $(u, v) \in E_f$; hence, v can be reached by s in E_f . However, this contradicts the definition of R .

Therefore,

$$\sum_{v \in V} f(s, v) = \sum_{u \in L, v \in R} c_{uv} = c_{LR}.$$

According to Lemma 7, such a relaxed flow f is a maximum relaxed flow. \square

Now, we show that the RIPR algorithm indeed terminates.

Lemma 9. *If the adaptation is applied n ($n \geq 0$) times, then the number of relabel operations that can be performed is less than $(2n + 2)|V|^2$.*

Proof. If the adaptation is applied n times, then $h(s) = (2n + 1)|V|$. According to Lemma 4, $h(u) \leq h(s) + |V| - 1 = (2n + 2)|V| - 1$ for each $u \in V$. Each time $Relabel(u)$ is applied, $h(s)$ is increased at least by 1. Since $h(s) = 0$ initially, $Relabel(u)$ is applied at most

$(2n + 2)|V| - 1$ times. There are $|V|$ nodes in the system; hence, the total number of Relabel that can be performed is at most $((2n + 2)|V| - 1)|V|$, which is less than $(2n + 2)|V|^2$. \square

Lemma 10. *If the adaptation is applied n ($n \geq 0$) times, then the number of saturated push operations that can be performed is less than $(n + 1)|V| \cdot |E|$.*

Proof. Consider edge $(u, v) \in E$. Suppose that a saturated push $Push(u, v)$ is first applied. For a second saturated push to be applied over (u, v) , $Push(v, u)$ must be applied before the second saturated push. Because $h(u) > h(v)$ for the first push (otherwise, the first push will not be applied), then $h(v)$ must be increased at least by 2; otherwise, if $h(v) \leq h(u)$, then $Push(v, u)$ will not be applied. Similarly, $h(u)$ must be increased at least by 2 for the second saturated push $Push(u, v)$ to occur and so on and so forth. Because

$$h(u) \leq h(s) + |V| - 1 = (2n + 2)|V| - 1,$$

$h(u)$ and $h(v)$ cannot increase to infinity. It is easy to show that a saturated push can occur at most $((2n + 2)|V| - 1)/2$ times for edge (u, v) . There are $|E|$ edges in the graph. The total number of saturated push operations is less than $((2n + 2)|V| - 1)/2 \cdot |E|$, which is less than $(n + 1)|V| \cdot |E|$. \square

Lemma 11. *If the adaptation is applied n ($n \geq 0$) times, then the number of nonsaturated push operations that can be performed is less than $(n^2 + 2n + 1) \cdot (4|V|^3 + 2|V|^2|E|)$.*

Proof. Define a potential function $\Phi = \sum_{e(u) > 0} h(u)$. $\Phi = 0$ initially. Obviously, $\Phi \geq 0$.

According to Lemma 4,

$$h(u) \leq h(s) + |V| - 1 = (2n + 2)|V| - 1;$$

hence, a relabel operation increases Φ by at most $(2n + 2)|V| - 1$. According to Lemma 9, there can be at most $(2n + 2)|V|^2$ relabel operations. The increase in Φ induced by all relabel operations is at most $((2n + 2)|V| - 1) \cdot (2n + 2)|V|^2$. A saturated push $Push(u, v)$ increases Φ by at most $(2n + 2)|V| - 1$ since $e(v)$ may become positive after the push, and $(2n + 2)|V| - 1$ is the highest value that $h(v)$ can be. According to Lemma 10, the increase in Φ induced by all saturated push is at most $((2n + 2)|V| - 1) \cdot ((n + 1)|V| \cdot |E|)$.

For a nonsaturated push $Push(u, v)$, $e(u) > 0$ before the push and $e(u) = 0$ after the push; hence, $h(u)$ is excluded from Φ after the push. If $e(v) > 0$ after the push, Φ is decreased at least by 1 because $h(u) - h(v) > 0$. If $e(v) \leq 0$ after the push, then Φ is decreased by $h(u) \geq 1$.

Therefore, the total increase in Φ is at most

$$\begin{aligned} & ((2n + 2)|V| - 1) \cdot (2n + 2)|V|^2 + ((2n + 2)|V| - 1) \\ & \cdot ((n + 1)|V| \cdot |E|) < (n^2 + 2n + 1) \cdot (4|V|^3 + 2|V|^2|E|), \end{aligned}$$

whereas each nonsaturated push decreases Φ at least by 1. Therefore, the total number of nonsaturated push operations that can be performed is at most $(n^2 + 2n + 1) \cdot (4|V|^3 + 2|V|^2|E|)$. \square

Theorem 4 is recited below.

Theorem 4. *Given graph $G(V, E)$ with root s and sink t and assuming that no capacity changes occur after the n th adaptation operation, then the number of “push” and “relabel” operations executed by the RIPR algorithm is bounded from above by $O(n^2 \cdot |V|^2 \cdot |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the graph. Additionally, the RIPR algorithm finds an optimal solution to the Relaxed Flow Problem when no “push” or “relabel” operation need to be performed.*

Proof. Immediate from Lemmas 8, 9, 10, and 11. \square

ACKNOWLEDGMENTS

This paper was supported by the US National Science Foundation under award No. ACI-0305763. A preliminary version of this paper appeared in the Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2004.

REFERENCES

- [1] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, “Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms,” *IEEE Trans. Parallel Distributed Systems*, vol. 15, no. 4, pp. 319-330, Apr. 2004.
- [2] C. Banino, O. Beaumont, A. Legrand, and Y. Robert, “Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids,” *Proc. Int’l Conf. Applied Parallel Computing (PARA ’02)*, pp. 423-432, 2002.
- [3] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, “Assessing the Impact and Limits of Steady-State Scheduling for Mixed Task and Data Parallelism on Heterogeneous Platforms,” *Proc. Int’l Conf. Heterogeneous Computing (HeteroPar ’04)/Proc. Int’l Symp. Parallel and Distributed Computing (ISPD’04)*, 2004.
- [4] O. Beaumont, A. Legrand, Y. Robert, L. Carter, and J. Ferrante, “Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms,” *Proc. Int’l Parallel and Distributed Processing Symp. (IPDPS ’02)*, Apr. 2002.
- [5] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, “The GrADS Project: Software Support for High-Level Grid Application Development,” *Int’l J. Supercomputer Applications*, vol. 15, no. 4, 2001.
- [6] 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,” *J. Parallel and Distributed Computing*, vol. 61, pp. 810-837, 2001.
- [7] D. Collins and A. George, “Parallel and Sequential Job Scheduling in Heterogeneous Clusters: A Simulation Study Using Software in the Loop,” *Simulation*, vol. 77, no. 6, pp. 169-184, Dec. 2001.
- [8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. MIT Press, 1992.
- [9] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture, a Hardware/Software Approach*. Morgan Kaufmann, 1999.
- [10] Distributed.net, <http://www.distributed.net>, 2007.
- [11] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On Power-Law Relationships of the Internet Topology,” *Proc. ACM Ann. Conf. Special Interest Group on Data Comm. (SIGCOMM ’99)*, pp. 251-262, 1999.
- [12] R.F. Freund and H.J. Siegel, “Heterogeneous Processing,” *IEEE Computer*, vol. 26, no. 6, pp. 13-17, 1993.
- [13] M.G. Gouda and F.F. Haddix, “The Alternator,” *Proc. Int’l Conf. Distributed Computing Systems (ICDCS ’99) Workshop Self-Stabilizing Systems*, pp. 48-53, 1999.
- [14] B. Hong and V.K. Prasanna, “Maximum Data Gathering in Networked Sensor Systems,” *Int’l J. Distributed Sensor Networks*, vol. 1, no. 1, 2005.
- [15] *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, eds. Morgan Kaufmann, 1999.

- [16] O. Ibarra and C. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *J. ACM*, vol. 24, no. 2, pp. 280-289, 1977.
- [17] C. Inc. Cray XT3 Datasheet, <http://www.cray.com>, 2007.
- [18] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home-Massively Distributed Computing for SETI," *Computing in Science and Eng.*, Jan. 2001.
- [19] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "On the Interference of Communication on Computation in Java," *Proc. Int'l Workshop Performance Modeling, Evaluation, and Optimization on Parallel and Distributed Systems (PMEO-PDS '04)*, Apr. 2004.
- [20] S.S. Kulkarni, C. Bolen, J. Oleszkiewicz, and A. Robinson, "Alternator in Read/Write Model," Technical Report MSU-CSE-04-28, Dept. of Computer Science, Michigan State Univ., East Lansing, Michigan, July 2004.
- [21] 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*, R. Grant, ed., Horizon Press, 2002.
- [22] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *J. Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107-131, 1999.
- [23] A. Medina, I. Matta, and J. Byers, "On the Origin of Power Laws in Internet Topologies," *ACM Computer Comm. Rev.*, vol. 30, no. 2, pp. 18-28, Apr. 2000.
- [24] B.M.E. Moret, D.A. Bader, and T. Warnow, "High-Performance Algorithm Engineering for Computational Phylogeny," *J. Supercomputing*, vol. 22, no. 1, pp. 99-111, 2002.
- [25] R.V. Nieuwpoort, T. Kielmann, and H.E. Bal, "Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications," *Proc. Eighth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '01)*, pp. 34-43, 2001.
- [26] P. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann, 1996.
- [27] U. Rencuzogullari and S. Dwardadas, "Dynamic Adaptation to Available Resources for Parallel Computing in an Autonomous Network of Workstations," *ACM SIGPLAN Notices*, vol. 36, no. 7, pp. 72-81, 2001.
- [28] G. Shao, F. Berman, and R. Wolski, "Master/Slave Computing on the Grid," *Proc. Ninth Heterogeneous Computing Workshop*, May 2000.
- [29] K. Taura and A.A. Chien, "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources," *Proc. Heterogeneous Computing Workshop*, pp. 102-115, 2000.
- [30] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, A.J.G. Hey, and G. Fox, eds., John Wiley & Sons, 2003.
- [31] J.B. Weissman, "Scheduling Multi-Component Applications in Heterogeneous Wide-Area Networks," *Proc. Heterogeneous Computing Workshop, Int'l Parallel and Distributed Processing Symp. (IPDPS '00)*, May 2000.



puting in networked sensor systems. He is a member of the IEEE.



member of the Center for Applied Mathematical Sciences (CAMS) at the University of Southern California. He served as the division director for the Computer Engineering Division during 1994-1998. His research interests include parallel and distributed systems, embedded systems, configurable architectures, and high performance computing. He has published extensively and consulted for industries in the abovementioned areas. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He is the steering cochair of the International Parallel and Distributed Processing Symposium (merged IEEE International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP)) and is the steering chair of the International Conference on High Performance Computing (HiPC). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *Proceedings of the IEEE*. He is the editor in chief of the *IEEE Transactions on Computers*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a fellow of the IEEE and a member of the IEEE Computer Society.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.

Bo Hong received the MEng degree from Tsinghua University, Beijing, in 2000 and the PhD degree in electrical engineering from the University of Southern California in 2005. He is currently an assistant professor in the Electrical and Computer Engineering Department at Drexel University. His research interests include parallel and distributed processing, modeling and optimization of high performance computing systems and applications, and distributed com-

Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from the Pennsylvania State University in 1983. Currently, he is a professor in the Department of Electrical Engineering, as well as in the Department of Computer Science, at the University of Southern California, Los Angeles. He is also an associate