A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources

Kenjiro Taura*
Computer Systems and Architecture Group
Computer Science and Engineering
University of California, San Diego
tau@csag.ucsd.edu

Andrew Chien
Computer Systems and Architecture Group
Computer Science and Engineering
University of California, San Diego
achien@csag.ucsd.edu

Abstract

*A heuristic algorithm that maps data-processing tasks onto heterogeneous resources (*i.e., *processors and links of various capacities) is presented. The algorithm tries to achieve a good* throughput *of the whole data-processing pipeline, taking both parallelism (load balance) and communication volume (locality) into account. It performs well both under compute-intensive and communication-intensive conditions. When all tasks/processors are of the same size and communication is negligible, it quickly distributes the compute load over processors and finds the optimal mapping. As communication becomes significant and reveals as a bottleneck, it trades parallelism for reduction of communication traffic. Experimental results using a topology generator that models the Internet show that it performs significantly better than communication-ignorant schedulers.*

1. Introduction

It is widely believed that future computing environment will consist of geographically distributed compute- and data-resources connected with diverse communication capacities, forming a so-called "computational Grid" environment [10]. Computational elements range from a desktop to clusters [4, 5] to supercomputers, and links range from phone lines to gigabits system area networks. Both CPU capacity and the network connectivity are improving in a rapid pace, but the recent trend indicates network bandwidth increases more rapidly than CPUs. As a consequence, communication-intensive parallel jobs, which we are currently able to run only on dedicated supercomputers or clusters, are likely to be hosted by a collection of desktops in laboratories or even home. This brings the Grid beyond just an aggregation of computational horsepower and enables a qualitatively different use of it. On the other hand, it presents significant resource management problems to all levels of parallel/distributed software developments.

One of the fundamental elements of such resource management problems is, given an application that consists of many communicating tasks, to select a suitable set of resources and map its tasks appropriately. To obtain a robust performance across a wide range of resource configurations, mapping algorithms must trade load balancing for the reduction of communication, and vice versa.

In this paper, we present a graph-theoretic formulation of this general problem and propose its heuristic algorithm. The algorithm takes as input a *task graph* and a *resource graph* and outputs the mapping from tasks to processors. A task graph models a data processing pipeline; a task in a pipeline continuously receives data from adjacent tasks, processes them, and sends processed data to other tasks. Weights of nodes and edges represent compute and communication *requirements* of these tasks, respectively. A resource graph models processors and links. Weights of nodes and edges represent their compute and communication *capacities*, respectively. If too many tasks are assigned on a processor or too much communication goes through a link, the processor or the link becomes a bottleneck and determines the overall throughput of the entire pipeline.

The key to achieving a good throughput is *clustering* of a task graph, a process which recognizes highly-connected components in a task graph. A cluster in a task graph represents a set of tasks that are intensively communicating with each other. These tasks should be placed in a single processor if available communication bandwidth is low. Among several graph clustering methods proposed in the literature [9, 13, 28], we use a simplified version of the stochastic flow injection method [29, 30] proposed by by Yeh et al.

Under a simple condition in which tasks and proces-

*Also affiliated to Departement of Information Science, University of Tokyo.

sors are of a uniform weight and communication is negligible, it guarantees to quickly give the optimal solution, in which tasks are uniformly distributed over processors. As communication becomes significant and reveals as a bottleneck, it co-locates highly communicating tasks to reduce communication traffic. We have implemented the algorithm in scripting language Python [16] and performed experiments using a simplified version of an Internet topology generator [7, 12] to generate a realistic resource graph. As we expected, our algorithm significantly outperforms simpler, communication-ignorant algorithms on communication-intensive conditions.

The rest of the paper is organized as follows. Section 2 gives a practical motivating scenario that we envision will commonly occur in emerging Grid applications. Section 3 is devoted to the problem formulation and Section 4 describes its algorithm. Section 5 shows experimental results. Section 6 mentions relationship to other work and Section 7 summarizes the paper and states future work.

2. A Practical Scenario

Consider an application which reads a large volume of data from geographically distributed source (storage server), processes them, and displays the result on a desktop. An example of such application is SARA [22], in which the data is surface data of the earth. Emerging distributed applications that use geographically distributed data such as digital libraries [1] and scientific data archives [6] will have more or less this kind of structure.

Even in this fairly simple setting, one question that arises is where the data should be processed. The best decision clearly depends on how computationally expensive the processing is, how much data it reads from the source and writes to the display, how computationally powerful are the desktop and the storage server, and how much bandwidth we have between these nodes. The decision is much more complex when we have a more involved data processing pipeline and more available resources such as parallel compute-servers. Finally, the availability of all these resources changes over time. For example, processing should be done on the desktop when the storage server is highly loaded.

It can easily be seen that it is, if not impossible, difficult and time-consuming for individual application developers to implement a decision that works in a wide range of resource configurations, even in a very simple case like this. Application-specific solutions, if any, would not generalize to even more complex and dynamic cases, in which we have hundreds of tasks that are created and ceased over time.

3. Problem Description

3.1. Preliminary Definitions and Notations

Resource Graph and Task Graph: A *resource graph* is a weighted graph (both nodes and edges are weighted).[1] A node of a resource graph represents a processor and an edge a link between a pair of processors. The weight of a node represents the processor's compute capacity (the amount of computation that can be performed in a unit time) and that of an edge the link's communication capacity (the amount of data that can go through the link in a unit time).

A *task graph* is also a weighted graph. A node of a task graph represents a task and an edge a continuous communication (stream) between a pair of tasks. The weight of a node represents the task's compute requirement (the amount of computation that must be done for this task to make a unit progress) and that of an edge the communication requirement of the connected tasks (the amount of data that must be communicated for these tasks to make a unit progress).

Note that a task graph is *not* a traditional dependence graph, in which an edge $s \rightarrow t$ represents the fact that task t can start its computation only after s has finished. Rather, our task graph models a data processing *pipeline*, in which all tasks continuously receive pieces of data, process them, and then send the processed data. A typical example is a multimedia data processing pipeline such as Smart Kiosk [21, 20], in which the natural unit of work is a frame. Typical tasks include compression, decompression, color tracking, object detection, and so on. A weight of a node is the amount of computation performed by the task per single frame, whereas that of an edge the size of transferred data per frame.

Unlike other formulations [14, 26], our model does not have an explicit notion of *parallelized tasks*. That is, a single node of a task graph can be mapped only on a single node of a resource graph. A parallelized task can be to some extent modeled by many nodes that together represent a single logical task.

Notations: Let G be a weighted graph. G_i is the weight of node i and $G_{i,j}$ the weight of edge $i \rightarrow j$. G^i is the weighted graph isomorphic to G, in which the weight of node i is one and that of all other nodes/edges is zero. $G^{i,j}$ is the weighted graph isomorphic to G, in which the weight of the edges along the path from i to j is one and that of all other nodes/edges is zero (Figure 1). If there are multiple paths between a pair of nodes, we fix one such path.

Let G and H be isomorphic weighted graphs. We define $G + H$ as node- and edge-wise addition of their weights. We similarly define $G - H$ and G/H. Let k be a scalar, kG

[1] Graphs can either be directed or undirected, but the following discussion assumes directed graphs.

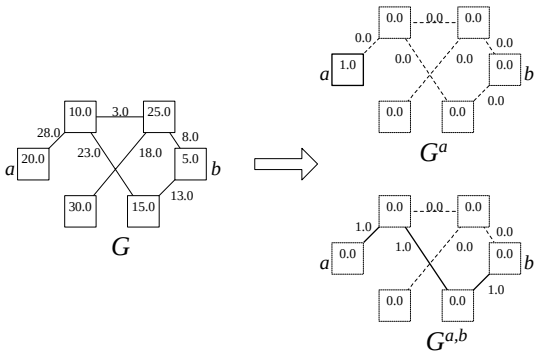

Figure 1. A weighted graph G, G^i, and $G^{i,j}$.

denotes a graph isomorphic to G whose weights are multiplied by k.

Interpretation: As we mentioned earlier, a task graph models a set of tasks each of which repeatedly receives data from other tasks, performs some computation on them to produce other data, and sends the produced data to other tasks. We make it more precise by showing the pseudo code for a task t in a task graph $G = \langle V, E \rangle$, as shown in Figure 2.

The progress rate of task t is determined by several factors. First, t will experience a certain amount of wait time at the **wait** phase, if tasks that are sending data to t cannot produce data fast enough or the bandwidth from these tasks to t are not enough. Second, more obviously, this task will spend some time at the **compute** step. Finally, the time taken at the **send** step will be determined by outgoing bandwidth and how fast receiving tasks can consume data.

As will be made clear in the next section, our problem formulation effectively makes idealizing assumptions that the progress rate of this task is determined by the maximum, rather than the summation, of these three factors. For example, if the wait step in isolation takes 5 time units, the compute step 3 time units, and the send step 2, then `unit_progress` as a whole takes only 5 time units, rather than $5 + 3 + 2 = 10$. This approximates a situation in which these three phases interleave in the infinitely fine-grained manner; that is, **compute** phase begins processing data when a single bit of data appears in the incoming stream, and the **send** phase sends data as soon as produced.

3.2. Formulation

We are interested in the throughput (the number of work units completed per unit time) of the system in equilibrium. Given a mapping from tasks to processors, it determines the amount of computation each processor must perform to

```
/* G = ⟨V, E⟩.
   a unit work task t repeats. */
unit_pregress(t)
{
  /* (1) wait */
  for s ∈ V s.t. s → t ∈ E {
    wait for G_{s,t} units (e.g., bytes) of data
      to arrive from s;
  }
  /* (2) compute */
  perform G_t units of computation upon the
    received data;
  /* (3) send */
  for u ∈ V s.t. t → u ∈ E {
    send G_{t,u} bytes of data to u;
  }
}

/* a task t simply repeats unit_progress forever */
task(t)
{
  while (1) {
    unit_progress(t);
  }
}
```

Figure 2. Pseudo code for task t.

make all tasks complete a unit work; it is simply the summation of task weights mapped on the processor in question. It similarly determines the volume of data each link must transfer to have all tasks make a unit-progress. By dividing the requirement at each node (edge) by its corresponding computation (communication) capacity, we have the time required to service requested computation/communication at the node (edge). We call it *occupancy* at the node (edge). The maximum occupancy over the entire graph gives us the time required to unit-progress all tasks. The goal is to make the maximum occupancy of the mapping as small as possible. Note that an occupancy is the inverse of the number of unit works finished per a unit time. Thus, minimizing the occupancy is equivalent to maximizing the throughput.

A more formal description follows. Let $G = \langle V_G, E_G \rangle$ be a task graph and $P = \langle V_P, E_P \rangle$ a processor graph. Let m be a mapping from V_G to V_P. We define the *load graph* of the mapping, denoted by $L(G, P, m)$, as:

$$L(G, P, m) = \sum_{t \in V} G_t P^{m(t)} + \sum_{(s,t) \in E} G_{s,t} P^{m(s),m(t)}$$

That is, a load graph is a graph whose weights represent the amount of computation and communication required (at each node and edge) to unit-progress all tasks.

Occupancy graph of the mapping, denoted by $O(G, P, m)$, is obtained by simply dividing the load by the capacity at each node and edge:

$$O(G, P, m) = L(G, P, m)/P$$

The goal is to find a mapping m that minimizes $\texttt{max}(O(G, P, m))$, where $\texttt{max}(X)$ is the maximum weight over nodes and edges in graph X. Figure 3 shows an example of a load graph and an occupancy graph.

Note that the above formulation effectively assumes that all tasks progress in the same pace; when any of the tasks takes x unit time to make a unit progress, all the other tasks also take x. In other words, resources are never used to make some tasks go faster than the others. This is a practical assumption because, assuming finite communication buffers, any pair of communicating tasks must progress in the same pace in equilibrium. Consequently, for connected task graphs, tasks must eventually match their paces with all the other tasks.

Finally, we state that this problem is NP-hard. We show that the corresponding decision problem TASKMAP, which asks if a mapping whose maximum occupancy is no greater than a specified limit exists, is NP-hard. There are several NP-hard problems that straightforwardly reduce to TASKMAP. Reducing Knapsack problem is particularly simple; we however use a reduction from the two-way graph partitioning problem which is also NP-hard [19], because we believe it better illustrates the difficulty of the problem

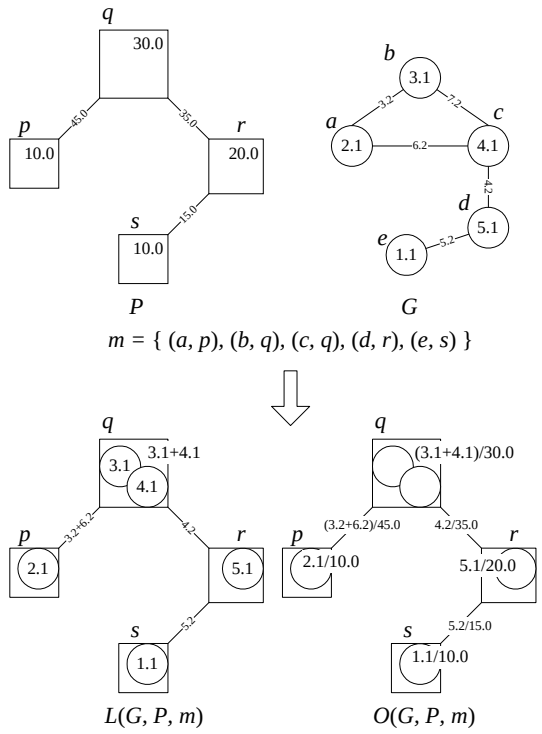

Figure 3. Load graph and occupancy graph.

(in particular it also shows the problem remains NP-hard even if we restrict all tasks to be the same size). The graph partitioning problem, PARTITION, takes an (unweighted) graph $G = \langle V, E \rangle$ and an integer c as input, and asks if there is a partition $V = V_1 + V_2$, such that V_1 and V_2 are disjoint and equal size (*i.e.*, $V_1 \cap V_2 = \emptyset$ and $|V_1| = |V_2| = |V|/2$) and the number of edges between V_1 and V_2 is $\leq c$.

Theorem 1 *TASKMAP is NP-hard.*

Proof: For a given instance of PARTITION $G = \langle V, E \rangle$ and c, we construct an instance of TASKMAP as follows.

- The task graph is a graph isomorphic to G, whose node weights and edge weights are all ones.
- The resource graph is a graph of two nodes, whose weights are both $|V|/2$, and the weight of the edge between the two is c.
- The maximum occupancy is one. That is, we ask if there is a mapping whose maximum occupancy is no greater than 1.

It is easily seen that if and only if there is such a mapping, there is a solution for the original graph partitioning problem, and the reduction can be performed in a polynomial time (Q.E.D).

4. The Algorithm

4.1. Motivating Example

If tasks are very compute-bound (communication is almost negligible), mapping is relatively straightforward, at least when task sizes are fairly uniform. It simply amounts to assigning each processor task weights roughly proportion to its compute capacity. Our main contribution is on cases where tasks are more communication intensive, thus such communication-ignorant mappings result in excess traffic that limits the performance. With increasing communication intensity of tasks, it becomes likely that mapping tasks that intensively communicate with each other on the same processor results in a significantly better performance.

As is the case in most combinatorial problems, the fundamental difficulty in achieving such mappings lies in the fact that the performance as a function of mappings is quite discontinuous and there are many local optima; the desired mapping is quite different from one communication intensity to another, and mappings that are in some sense 'between' these desired mappings are typically worse than both. Therefore it is difficult to move from one desired mapping to another by a series of greedy moves. To illustrate this, consider a task graph shown in Figure 4 where all nodes weigh one and all edges weigh c (a parameter). When c is very small, the desired mapping will typically be the one in which a single processor has a single task (assuming sufficient number of equally powerful processors). As c increases up to a certain threshold, the best mapping will typically become the one in which a single processor is assigned to a single cluster of tasks (as easily perceived by humans). Everything between these two extremes (for example, mappings in which a single processor has two tasks) are typically *worse than both*. This is because, when compared to the first extreme (one task per processor), the amount of traffic a single processor sends or receives increases, thus it does not reduce the communication bottleneck. The communication bottleneck can be eliminated only by moving *all* tasks of a cluster to a single processor. This property prohibits the use of a simple local search strategy which tries to find a task t and a processor p such that moving t to p improves the objective function. It is quite unlikely that a series of such moves eventually reaches the desired extreme, whichever is the better.

4.2. Overall Structure

As is easily seen from the example just discussed, the key to achieving a good mapping is to recognize highly-connected clusters, and use this clustering information to guide the mapping process. Our basic approach is to linearly order tasks in such a way that tasks within a cluster are close to each other, and put tasks to processors according to this order (as indicated by the labels in the figure). If a single processor is assigned to multiple tasks, they are likely to be in the same cluster, and therefore, when the tasks turn out to be communication-bound, processors can reduce communication simply by accommodating more tasks from the list.

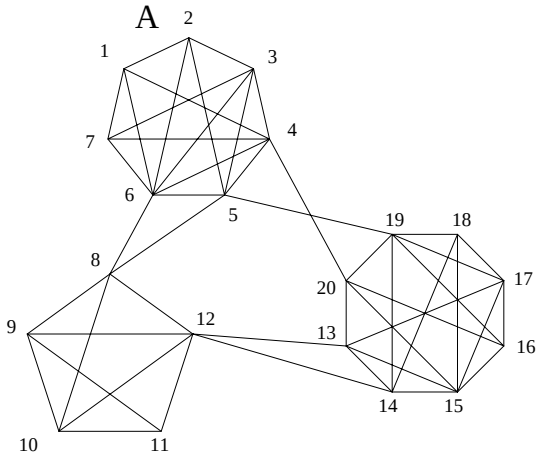

Figure 4. A graph with highly-connected subgraphs.

To continue the above example, we first pick up a processor and move tasks to it from the list. As tasks are ordered as shown in the figure, we exclusively choose tasks from a cluster (labeled A) in the beginning. The remaining problem is when we should stop this process and go onto the next processor. The best answer again depends on communication intensity; when c is small, it is typically when the compute-load is best balanced among processors, and otherwise when one or more clusters have just moved. Details are given in Section 4.4.

Our entire algorithm first obtains the appropriate order of tasks based on a simplified version of stochastic flow injection method proposed in [29, 30]. Given this information, it obtains an initial mapping and then improves it step by step. The elementary procedure mentioned above is used both to obtain the initial mapping and to improve it. The top-level structure of the algorithm is illustrated in Figure 5.

In the following sections, we first describe the clustering algorithm to obtain the order of tasks in Section 4.3, the elementary procedure that moves tasks to a processor from the list in Section 4.4, and how to improve the mapping once obtained in Section 4.5. Throughout the sections, $G = \langle V_G, E_G \rangle$ and $P = \langle V_P, E_P \rangle$ refer to the given task graph and the resource graph, respectively. As a convention, we do not update data structures in place (we always rebind a variable to signify an update). Variables assigned in one iteration of a loop and used in the next is subscripted

```
/* G = ⟨V_G, E_G⟩ : task graph.
   P = ⟨V_P, E_P⟩ : resource graph. */
taskmap()
{
  <_G = clustering(G); — (section 4.3)
  m = { }; /* empty map */
  m = map_tasks(m); — (section 4.4)
  repeat {
    m' = m;
    m =improve(m'); — (section 4.5)
  } while (O(G, P, m) < O(G, P, m'))
}
```

Figure 5. The overall structure of the algorithm.

by a loop index, even though it is a single variable in the real program.

Finally, we made various simplifications for the purpose of presentation. For example, the following algorithm calculates $L(G, P, m)$ many times, with m's that only slightly differ from each other. The actual program keeps track of $L(G, P, m)$ all the time and incrementally updates it as m changes. This kind of practical optimizations are not explicit in the description.

4.3. Clustering Task Graph

The clustering algorithm is shown in Figure 6. Given a graph H, it first creates a tree that hierarchically decompose the task graph into clusters (line 3). The root of the tree represents the entire set of nodes, whereas a leaf a singleton set of a node. Children of a node are partitions of the parent node, obtained by a simplified stochastic flow injection method as described below. Once such a tree is obtained, we determine a total order between nodes, $<_H$, simply by traversing the tree in a depth-first order (line 4).

The stochastic flow injection was originally proposed for VLSI circuit partitioning and works as follows:

1. Randomly pick up two nodes s and t of the given graph G.
2. Find the shortest path between s and t.
3. Decrement the weights of all the edges on the path by a (small) constant Δ (*i.e.*, inject a flow Δ between s and t).
4. Remove edges whose weight become zero or negative.

```
1:  clustering(H)
    {
      T =recursive_clustering(H);
      <_H= depth-first traversal order of T;
5:    return <_H;
    }

    recursive_clustering(H)
      H = (V, E) /* a subgraph of the task graph */
10: {
      if (V is singleton (= {v})) {
        return leaf(v)
      } else {
        H_1, ···, H_n =clusters obtained by
15:       stochastic flow injection (see text);
        return node(recursive_clustering(H_1),
                    ···, recursive_clustering(H_n));
      }
    }
```

Figure 6. Clustering Task Graphs.

5. Repeat 1-4 until the graph becomes unconnected.
6. When graphs are disconnected, each connected component is a cluster.

The intuition is that if only a small number of edges bridge two (or more) large clusters, such edges are likely to be decremented often, and the graph soon becomes disconnected by these edges.

In the original stochastic flow injection method, another phase follows to merge some of the clusters hereby obtained, but we simply skip this phase, because our purpose is to recursively decompose clusters until each cluster becomes a singleton. We also slightly modified the above step 1, so that a task is chosen by a probability proportional to its weight; this is necessary because the original stochastic flow injection method assumes uniform weights (as in the case in their application).

4.4. The Elementary Move

Procedure `map_tasks` shown in Figure 7 takes as a parameter m, a partial mapping from tasks to processors (it is partial because some tasks are not mapped). It maps tasks not mapped in m onto V_P, by simply making a series of calls to a more elementary procedure `map_tasks_on`, which maps some tasks to a specified processor.

The procedure `map_tasks_on` takes three parameters, m, q, and Q; m is a partial mapping from tasks to pro-

```
1:  map_tasks(m)
    {
      Q = V_P;
      while (Q ≠ {}) {
        q = a processor ∈ Q;
5:      Q = Q − { q };
        m = map_tasks_on(m, q, Q);
      }
      return m;
    }
10: /* move some of the tasks not mapped in m to
         processor q, taking open communication and
       the balance between { q } and Q into account */
    map_tasks_on(m, q, Q)
    {
15:   U_{m_0} = { t | not mapped in m };
      for i = 1, ..., |U_{m_0}| {
        t = the minimum task ∈ U_{m_{i-1}} w.r.t. <_G;
        m_i = m_{i-1}[t/q]; /* add mapping t → q */
        U_{m_i} = U_{m_{i-1}} − { t };
20:     O = O(G, P, m_i);
        O_comp = O_comp(G, P, m_i, Q);
        /* O_comp = ∞ if Q = {} */
        O_→ = O_→(G, P, m_i, U_{m_i}, q);
        O_← = O_←(G, P, m_i, q, U_{m_i});
25:     M_i = max(O, O_comp, O_→, O_←);
      }
      find i that gave minimum M_i (i = 1, ..., |U_{m_0}|);
        break ties by selecting largest i.
      return m_i;
30: }
```

Figure 7. The elementary move operation.

cessors, q a processor $\in V_P$ onto which some tasks are going to be mapped by the procedure, and Q a subset of V_P ($q \notin Q$) yet unused. The goal is to put an appropriate number of tasks on q, so that we are likely to reach a good final mapping, if the remaining tasks are mapped on Q. As mentioned earlier, it puts tasks one after another in the order obtained by the clustering; as we add more tasks to q, we obtain a series of mappings $m_0 = m, m_1 = m_0[t_1/q], m_2 = m_1[t_2/q], \cdots, m_n = m_{n-1}[t_n/q]$,[2] where $t_1 <_G t_2 <_G, \cdots, <_G t_n$ and m_n is the total mapping from V_G to V_P. So the only question is which m_i we should choose.

Let U_m denote the set of tasks that are not mapped in m. At each step, we keep track of the following four (three in case of undirected graphs) values to evaluate the situation.

- (Line 20): The current occupancy $O(G, P, m_i)$.
- (Line 21): A hypothetic occupancy O_{comp}. $O_{\text{comp}}(G, P, m_i, Q)$ is an occupancy estimated by assuming that tasks $\in U_{m_i}$ are perfectly mapped on Q, ignoring communication. That is, it is simply the total compute requirement of these tasks over the total compute capacity of Q:

  $$O_{\text{comp}}(G, P, m, Q) = \frac{\sum_{t \in U_m} G_t}{\sum_{p \in Q} P_p},$$

  For convenience we define this to be ∞ when $Q = \{\}$.
- (Lines 23 and 24): Hypothetic occupancies $O_{\rightarrow}(G, P, m_i, U_{m_i}, q)$ and $O_{\leftarrow}(G, P, m_i, q, U_{m_i})$, which we call occupancies induced by *open communication*. Given a set of tasks T and a processor q, we define open communication from T to q (from q to T) to be the total communication volume from tasks in T to tasks on q (from tasks on q to tasks in T). $O_{\rightarrow}(G, P, m, T, q)$ refers to open communication from T to q divided by the total edge capacity adjacent to q. Similarly for $O_{\leftarrow}$. That is:

  $$O_{\rightarrow}(G, P, m, T, q) = \frac{\sum_{x \in T, m(y) = q} G_{x,y}}{\sum_{(p,q) \in E_P} P_{p,q}}, \text{ and}$$
  $$O_{\leftarrow}(G, P, m, q, T) = \frac{\sum_{x \in T, m(y) = q} G_{y,x}}{\sum_{(p,q) \in E_P} P_{q,p}}.$$

  When graphs are undirected, these two give the same value and are collectively referred to as $O_{\leftrightarrow}$.

At each step, we calculate the above four (or three in undirected case) values and record the *maximum* of them (M_i at line 25). The procedure returns m_i that minimizes M_i (lines 27-29).

[2] $m' = m[t/q]$ is an extension of m, s.t. $m'(t) = q$ and $m'(x) = m(x)$ for $x \neq t$.

The first item will be intuitive. The second one, O_{comp}, tries to estimate how much is the final occupancy going to be. Given this estimate, we determine how many tasks should be accommodated to the current processor q. For example, suppose compute capacity of q is 1, the total compute capacity of Q 99, and the total compute requirement of tasks yet to be mapped 1,000. Ideally, we like to obtain a mapping whose maximum occupancy is close to 1,000/(1 + 99) = 10. Put differently, when we compare a series of mappings $m_1, m_2, \cdots$, *any mapping whose occupancy is below 10 is equally good*; there is no points in quitting at m_i, when the occupancy of m_{i+1} is still below 10.

The third item, $O_{\rightarrow}$ ($O_{\leftarrow}$) or, *open communication metric* is to identify m_i at which the communication volume between tasks already mapped on q and those that are not is small. Keeping track of such communication is necessary because it is not taken into account by $O(G, P, m_i)$, which only counts tasks mapped in m_i. This guides the mapping process, by giving following pieces of information: "rather than choosing an m_5 at which open communication is so large, accommodate more tasks and choose m_8, at which the processor is more loaded, but communication traffic is much smaller." Accurate estimation clearly requires not only communication volume, but also the link bandwidth from q to processors that accommodate the other tasks. An obvious problem is we are yet to know how remaining tasks will be mapped, so we do not precisely know how much will the occupancy of these links be. We simply estimate this by: (1) calculating the total communication volume between tasks on q and the remaining tasks, and (2) dividing it by the total link capacity *adjacent to* q. This effectively assumes such communication will be routed evenly across all adjacent links and internal links (not adjacent to a processor) will not be bottleneck. These assumptions, the first one in particular, may be optimistic and need be more sophisticated when q has multiple adjacent links. In our experiments, a processor is adjacent only to a single link, thus this is not an issue.

To illustrate how the procedure works, let us look at a process that maps tasks to a processor as shown in Figure 8. We start from the empty mapping and add tasks to the left processor, in the order indicated by the numbers. Edges and nodes in the task graph weigh one. The edge of the resource graph weighs one and the two nodes five. Figure 9 plots O, O_{comp}, and $O_{\leftrightarrow}$ (graphs are undirected) at every step. Observe that the open communication metric goes up and down and that O_i (the maximum of the three values) minimizes at m_8, even though compute load between the two processors best balances at m_{11} (the point where two graphs O_{comp} and O intersect). Therefore the procedure will choose to put 8 tasks on the left processor, which is optimal. If edges of the task graph weigh much smaller (say, 0.1), on the other hand, the graph of $O_{\leftrightarrow}$ will become much lower, giving the best M_i at m_{11}. So in this case, the first processor will get 11 tasks, which is again optimal.

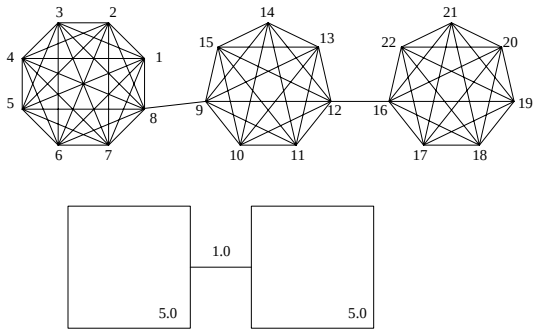

Figure 8. Example graph to illustrate `map_tasks_on`.

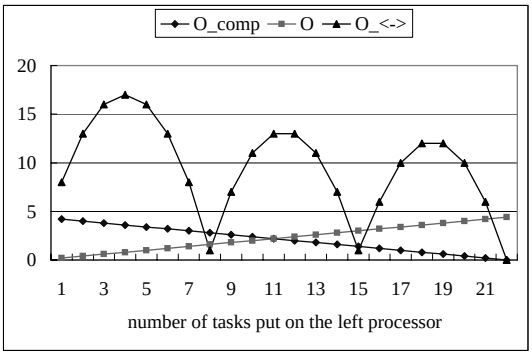

Figure 9. How O, O_{comp}, and $O_{\leftrightarrow}$ changes as we put tasks to the left processor in Figure 8.

Note that in general, for the easy case where communication is negligible and task and processors weigh uniformly (w.o.l.g. assume they weigh 1), the procedure `map_tasks`({}) is guaranteed to return the optimal mapping in which no processors get more than $\lceil N/P \rceil$ tasks, where N is the number of tasks and P the number of processors. To see this, consider what happens in the first call to `map_tasks_on`({}, q, $V_P - \{ q \}$). As communication is negligible, it simply amounts to finding the intersection of two graphs $O = i$ and $O_{\text{comp}} = (N - i)/(P - 1)$. Solving the equation $O = O_{\text{comp}}$ gives $i = N/P$ and thus the best value is obtained either at $\lceil N/P \rceil$ or $\lceil N/P \rceil - 1$. We can repeat this argument to show that this is the case for other processors. This property ensures our mapping procedure quickly gives a good solution for compute-mostly jobs without iterating improvements.

Other Implementation Notes: The actual implementation of the procedure is a bit more sophisticated to avoid useless computation.

- `map_tasks_on` quits as soon as $O(G, P, m_i)$ becomes greater than any of M_j ($j < i$). Since $O(G, P, m_i)$ is monotonically non-decreasing with respect to i, once this condition is observed, we have:

  $M_k \geq O(G, P, m_k) \geq O(G, P, m_i) > M_j$ for all $k > i$.

  Thus there is no chance that we observe a better M_k in future. Again, this guarantees that in the easy case mentioned above, `map_tasks_on` quits as soon as it puts $\lceil N/P \rceil + 1$ tasks on a processor.

- Both `map_tasks` and `map_tasks_on` optionally take one more parameter, u, which specifies the occupancy they should at least achieve. `map_tasks_on` quits as soon as $O(G, P, m_i)$ becomes greater than this value. `map_tasks` aborts the entire process as soon as $O_{\text{comp}}(G, P, m_i, Q)$ gets larger than u in an iteration. This is useful when we already know a mapping and try to improve it. In such circumstances, we determine u based on the current occupancy (*e.g.*, $u =$ the current occupancy $\times$ 0.9) and give it to `map_tasks`.

4.5. Iterative Improvement

Procedure `improve` in Figure 10 tries to improve a given (total) mapping m by first removing some tasks from m (line 3) and then applying `map_tasks` to the partial mapping obtained this way. Obviously, the key is to identify a small subset of tasks whose removal gives us a good chance to improve the mapping. A silly selection algorithm could remove all the tasks from m, effectively applying `map_tasks` again from the empty mapping.

The selection algorithm works as follows.

1. First calculate the current max occupancy and multiply it by an acceleration factor (currently 0.75). We remove tasks until the resulting mapping gives max occupancy below this value (line 10).

2. We scan nodes and edges of the resource graph, trying to find an edge or a node whose occupancy is greater than it.

3. If such a node is found, let p be the node. Find tasks s_i ($i = 1, 2, \cdots$), such that s_i is mapped on p and is not deleted yet. Among all such tasks, select the heaviest task.

4. If such an edge is found, let l be the edge. Find pairs of tasks (s_i, t_i) ($i = 1, 2, \cdots$), such that the route between s_i and t_i (on the processor graph) uses l and either s_i

```
1: improve(m)
   {
     m = remove_bottlenecks(m);
     m = map_tasks(m);
5:   return m;
   }

   remove_bottlenecks(m)
   {
10:  o = 0.75 × max(O(G, P, m));
     D = {}; /* set of deleted mappings */
     while (max(O(G, P, m − D) > o)) {
       L = L(G, P, m − D);
       find if any p ∈ P and q ∈ P s.t. L_{p,q}/P_{p,q} > o;
15:    if found {
         select s, t ∈ V_G s.t. (s ∉ D or t ∉ D),
           P_{p,q}^{m(s),m(t)} = 1, and G_{s,t} is maximum;
         D = D + {(s, m(s)), (t, m(t))};
       } else {
20:      there must be p ∈ P s.t. L_p/P_p > o;
         select s ∈ V_G s.t. s ∉ D,
           m(s) = p, and G_s is maximum;
         D = D + {(s, m(s))};
       }
25:  }
     return m − D;
   }
```

Figure 10. The procedure to improve the current mapping.

or t_i is not deleted yet. Among all such pairs, select the most heavily communicating pairs and delete them.

5. Repeat steps 2-4 until the occupancy becomes less than the target value computed at step 1.

It basically tries to identify a set of tasks that form *bottlenecks*, tasks making the current occupancy so large. It finds an edge or node in the resource graph whose occupancy is larger than the target value calculated from the current occupancy. If found, tasks contributing to the edge or the node are candidates.

While reasonable, this algorithm still has a room for further improvements which we are yet to experiment with. It does not pay attention to communication induced between deleted tasks and undeleted tasks. If the communication between them is large, attempts to moving those deleted tasks unavoidably induce a large communication traffic and are likely to fail. Among many ways to select candidate tasks, we like to select a set of tasks that do not intensively communicate with the other tasks. If such selection cannot be obtained, it makes sense to co-migrate some of the other tasks too, even if they do not constitute the bottleneck.

5. Experiments

5.1. Graph Generation

We used a simplified version of the Internet topology model described in [7, 12] to generate resource graphs. While they model WAN, MAN, and LAN, we omit MANs for simplicity and model resource graphs by two level (WAN and LAN) hierarchy. Given a configuration that describes such parameters as the number of WAN nodes, LANs, nodes within a LAN, and compute capacity of a processor, it generates a graph as follows.

- First generate the specified number of WAN nodes and randomly place them in a specified rectangle. Create edges between all pairs of nodes, associating a cost proportional to its length with each edge. Then make the minimum spanning tree of the resulting complete graph.

- Generate the specified number of LANs. For each LAN, first create a gateway and randomly place it in the specified rectangle. Connect gateway to its nearest WAN node. Then generate a randomly chosen number of nodes in the LAN. LAN is modeled as a (shallow) tree whose root is connected to its gateway and each node has a randomly chosen number of children. The compute capacity within a single LAN is uniform and chosen randomly.

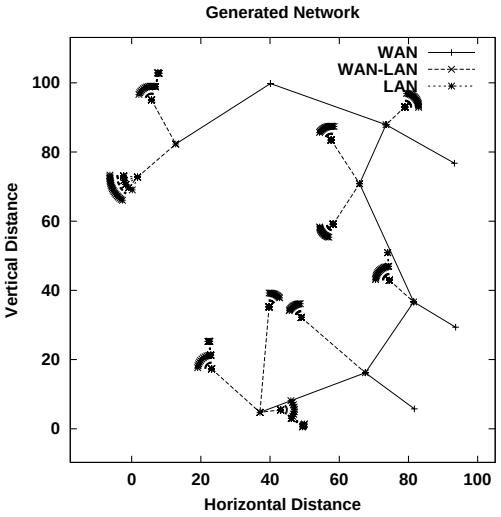

Figure 11. A typical resource graph used by the experiments. It was generated by a simplified Internet topology generator.

Table 1 lists relevant parameters and Figure 11 shows a typical graph generated by this model. A sector in the figure is a LAN, which has from 10 to 20 nodes. Depth of some sectors are one and that of others two.

For task graphs, we generate a pipeline of parallel jobs for each run as follows.

1. Randomly choose the number of tasks in a parallel job (m), and create a complete graph of m nodes. Nodes within a single parallel job are equally weighted and the weight is randomly chosen.

2. Repeat the step 1 a randomly chosen number (n) of times and obtain n complete graphs.

3. Connect these complete graphs to form a simple pipeline (without branches and merges). To connect two complete graphs A and B, we simply form a complete bipartite graph (create an edge between every task in A and every task in B). Each edge weighs $1.0/(a \times b)$, where a and b are the number of nodes in A and B, respectively. The total communication volume between two parallel jobs is always 1.0.

Resource Graph	
the number of WAN nodes	10
the number of LANs	10
bandwidth between WAN nodes	1000.0
WAN $\leftrightarrow$ LAN bandwidth	500.0
LAN bandwidth	50.0
the number of children for a LAN node	[5,20]
compute capacity of a processor	[3.0,15.0]
Task Graph	
the number of clusters in a task graph	[5,10]
the number of tasks in a cluster	[5,10]
compute requirement of a task	[1.0,3.0]
(total) comm. between a pair of clusters	1.0
communication intensity parameter	c (see text)

Table 1. Parameters used in the experiments. $[a, b]$ means that a value is chosen randomly from $[a, b]$ for each run.

Edges within a single parallel job are equally weighted and the weight is chosen randomly from $[1, c]$, where c is a parameter that controls the communication intensity of the tasks. We compare performance of several algorithms for various values of c.

Let us perform a rough calculation to see how communication intensity of tasks vary according to c. Since compute requirement per task is from 1.0 to 3.0, and capacity per processor is 3.0 to 15.0, the occupancy of a processor ranges from $1.0/15.0$ to 1.0, assuming a single processor accommodates a single task. When sufficiently many tasks are created, one of the processors is likely to get an occupancy close to 1.0. On the other hand, since the number of tasks in a parallel job is from 5 to 10, the communication volume per task is from $5c$ to $10c$ (ignoring inter-cluster communication, which is a fraction). Considering LAN bandwidth, which is 50.0, occupancy of an edge adjacent to a processor is $0.1c$ to $0.2c$, again assuming a single task on a single processor. Comparing the expected node occupancy (≈ 1.0) and this value, clustering is unlikely to be necessary for $c \approx 1$. In this sense, for $c \approx 1$, tasks are hardly communication intensive. For $c \approx 16$, on the other hand, an edge occupancy will range from 1.6 to 3.2, much larger than the expected processor occupancy. Therefore when $c \approx 16$, a good solution is likely to use clustering.

5.2. Results

We compare the following four algorithms for $c = 1$, 2, 4, 8, 12, and 16.

Base: Do not use the open communication metric described in Section 4.4. Also do not perform the improvement phase described in Section 4.2.

Base + improve: Do not use the open communication metric. Apply the improvement phase after an initial mapping is obtained, again without open communication metric.

Open: Use the open communication metric. But do not perform the improvement phase.

Open + improve: Use the open communication metric and apply the improvement phase to the initial mapping.

For each value of c, we generate 32 instances of the problem and run the four algorithms. For each instance and for each algorithm, we calculate the improvement of the occupancy against **Base**. Graphs in Figure 12 show the result. A dot corresponds to an instance and the value represents the relative improvement over Base (Note that in **Base + improve**, the number of dots looks much smaller than 32. This is because results are in many cases 1; *i.e.*, no improvement is observed). Figure 13 shows the average improvement over 32 instances.

It is clear that taking open communication into account becomes significant as tasks become communication intensive. As we have expected, all four algorithms perform equally well for $c \approx 1$. Adding the iterative improvements to **Open** slightly improved performance, but not very much. As we have discussed in Section 4.5, our task selection algorithm is not very sophisticated yet, so we need more experiments to be conclusive.

6. Related Work

6.1. Task Scheduling

There are a number of studies on task scheduling in heterogeneous environments [8, 11, 15, 17, 18, 27]. To the author's knowledge, most of these work have been focusing on scheduling DAGs, in which a task graph represents dependencies between tasks. DAG scheduling problem and the throughput optimization problem discussed in this paper are quite different, both in terms of basic techniques employed and target applications. In terms of techniques, most algorithms for DAG scheduling are more or less based on a list scheduling, whereas the basic model of the throughput optimization is graph partitioning. For target application, DAG scheduling applies to a set of many tasks that rarely communicate with each other, whereas the throughput optimization problem to tasks communicating via high-bandwidth streams. While both are important, we believe the throughput optimization problem discussed in this paper will increasingly become important for emerging multimedia and data-intensive applications on wide area.

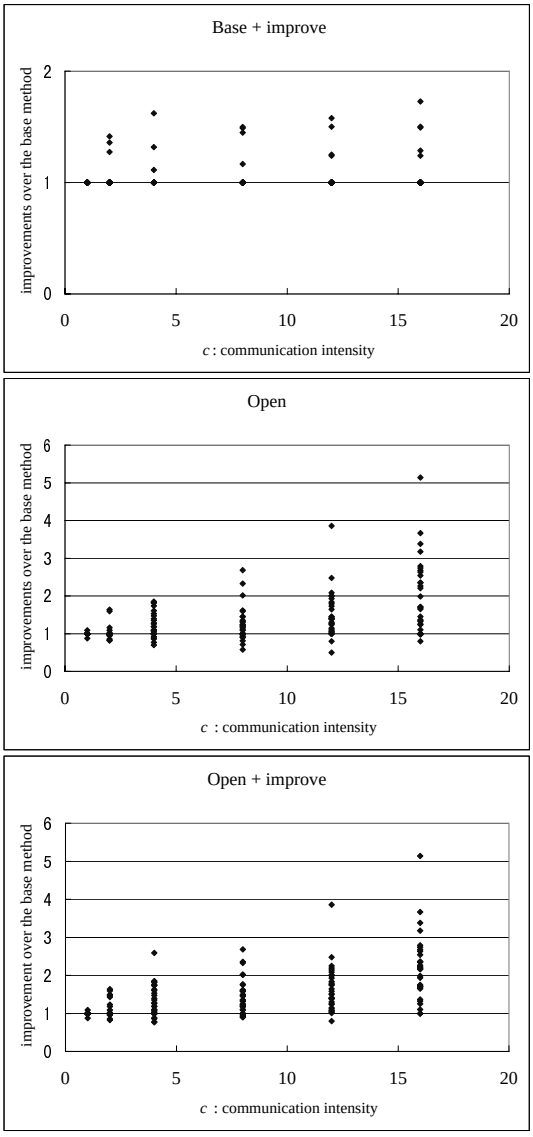

Figure 12. Improvements of the various methods over the Base method (Internet model).

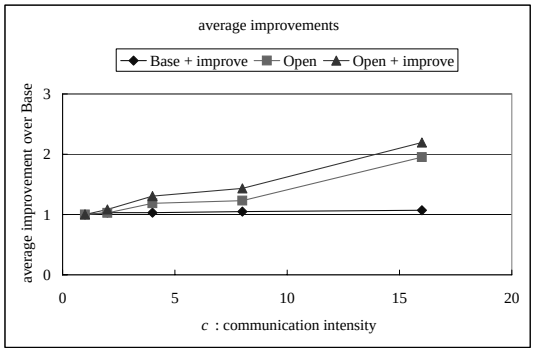

Figure 13. Average improvements.

Several studies on scheduling with bandwidth metrics have been done. Subhlok et al. [25, 26] studied optimal processor allocation for a set of communicating data parallel tasks, both with latency and bandwidth metrics. In their problem setting, performance of a task is a function of the number of processors allocated for that task and does not depend on which processors are used. They make a similar assumption on communication performance. Therefore the problem amounts to determining *how many* processors should be allocated for each task. This effectively assumes two things. One is that processor speed is uniform. The other is that link bandwidth is not only uniform but also very high, so the locations of communicating tasks do not matter. This will be a good model for system-area cluster, which is their target environment, but will not be directly applicable to multimedia/data-intensive applications on wide area.

Developing applications that exhibit robust performance over a wide range of resource conditions have become such an important issue. Several frameworks have been proposed [3, 24] and many practical studies on adaptive applications in heterogeneous environments have been conducted [2, 23]. While such studies are certainly instructive, it is difficult for individual programmers to perform such studies for every single application. We believe that task mapping should be much more automated.

6.2. Graph Partitioning

Graph partitioning tries to cut a graph into two ore more sub-graphs each of which is more connected than the entire graph. Our problem shares the common difficulty with this basic problem, in that moving any single node or exchanging any single pair of nodes is not likely to improve the objective function.

Kernighan and Lin [13] dealt with the basic two-way partitioning problem to cut the graph into two graphs of exactly the same size and gave the basic idea to overcome the local optima. Fidducia and Mattheyses [9] proposed a faster

algorithm for a slightly different problem, in which a certain amount of difference between the sizes of the two subgraphs is accepted. Wei et al. further proposed a ratio cut [28], which automatically achieves a balance between a low cut size and a good ratio of the sub-graph sizes. Finally, Yeh et al. proposed multi-way partitioning based on stochastic flow injection method [29, 30].

While our current algorithm can basically use any good partitioning algorithm as the preprocessing of a task graph, the following property of the Yeh's method is particularly attractive for our purpose; it can not only find highly connected components from a graph, but also finds the (negative) fact that no more natural clusters exist in a graph, in which case it typically divides the graph into many singletons. Having only two-way partitioning, we still have to apply two-way partitioning recursively. This is computationally expensive and does not improve quality.

7. Summary and Future Work

We have presented a heuristic algorithm for a task mapping problem, which takes compute and bandwidth requirements into account. The key to achieving good performance is clustering, a process that recognizes intensively-communicating tasks. We use this clustering information to obtain the order in which tasks should be put on processors. Open communication metric was introduced to decide how many tasks should be put in a processor. The algorithm is able to incrementally improve a given mapping, moving only those tasks that form the bottleneck. Therefore it can efficiently fix a significant load imbalance caused by a small number of tasks. We observed expected experimental results, indicating that our communication-sensitive algorithm significantly outperforms simpler, communication-ignorant algorithms for communication-intensive jobs.

We are planning to enhance this work in several ways. First, we are going to improve the task selection algorithm for incremental improvements, so that it moves clusters that do not intensively communicate with the rest of the tasks. Second, we will analyze computational complexity of the algorithm in detail. Third, we will try to identify other cases where this algorithm guarantees to produce a result within a constant of the optimal. Practical goals include developing a system that automatically selects resources and maps tasks on wide area, which helps Grid application designers develop performance-portable Grid code. We hope this work serves as a sound, logical step toward achieving this goal.

Acknowledgements The research described is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-99-1-0534, F30602-97-2-0121, and F30602-96-1-0286. It is also supported by NSF Young Investigator award CCR-94-57809 and NSF EIA-99-75020. It is also supported in part by funds from the NSF Partnerships for Advanced Computational Infrastructure – the Alliance (NCSA) and NPACI. Support from Microsoft, Hewlett-Packard, Myricom Corporation, Intel Corporation, Packet Engines, Tandem Computers, and Platform Computing is also gratefully acknowledged. The paper was written when the first author was visiting UCSD as an exchange visitor, supported by the Ministry of Education of Japan.

References

[1] G. Aloisio, M. Cafaro, and R. Williams. The digital puglia project: An active digital library of remote sensing data. In *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe*, volume 1593 of *Springer Lecture Notes in Computer Science*, pages 563–572, 1999. `http://www.cacr.caltech.edu/SDA/digital_puglia.html`.

[2] F. Berman and R. Wolski. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, 1996.

[3] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, 1997. `http://www-cse.ucsd.edu/groups/hpcl/apples/apples.html`.

[4] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Connelly, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane. Design and evaluation of an HPVM-based windows NT supercomputer. *The International Journal of High-Performance Computing Applications*, 13(3):201–209, Fall 1999. `http://www-csag.ucsd.edu/projects/hpvm.html`.

[5] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP)*, 1997. `http://now.cs.berkeley.edu/`.

[6] Digital Sky Project. Center for Advanced Computing Research (CACR) at California Institute of Technology. `http://www.cacr.caltech.edu/SDA/digital_sky.html`.

[7] K. C. Ellen W. Zegura and S. Bhattacharjee. How to model an Internetwork. In *Proceedings of IEEE Infocom '96*, 1996.

[8] M. Eshaghian and Y. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of Heterogeneous Computing Workshop*, 1997.

[9] C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *Proeedings of 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[11] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Proceedings of Heterogeneous Computing Workshop*, 1998.

[12] M. D. Ken Calvert and E. W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, June 1997.

[13] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[14] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proceedings of SC'99*, 1999. `http://www.sc99.org`.

[15] Y.-K. Kwok and I. Ahmad. *High Performance Cluster Computing*, volume 1, chapter 23, Parallel Program Scheduling Techniques, pages 553–578. Prentice Hall, 1999.

[16] M. Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[17] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of Heterogeneous Computing Workshop*, 1998.

[18] H. Nakada, A. Takefusa, S. Matsuoka, M. Sato, and S. Sekiguchi. A scheduling framework for global computing. In *Proceedings of Joint Symposium on Parallel Processing*, pages 277–284, 1999. `http://ninf.etl.go.jp/`.

[19] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*. Dover Publications, 1998.

[20] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-time memory: A parallel programming abstration for interactive multimedia applications. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999.

[21] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated task and data parallel support for dynamic applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 167–180, 1998.

[22] SARA: The Synthetic Aperture Radar Atlas. University of Lecce and California Institute of Technology. `http://sara.unile.it/sara/`.

[23] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998.

[24] P. Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, 1999.

[25] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[26] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of 8th Annual ACM Symposium on Parallel Alogorithms and Architectures (SPAA)*, pages 62–71, 1996.

[27] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of Heterogeneous Computing Workshop*, 1999.

[28] Y. Wei and C. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design*, 10:911–921, 1991.

[29] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. A probabilistic multicommodity-flow solution to circuit clustering problems. In *IEEE International Conference on Computer-Aided Design*, pages 428–431, 1992.

[30] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. Circuit clustering using a stochastic flow injection method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(2):154–162, 1995.

Kenjiro Taura is a research associate of Department of Information Science, University of Tokyo. He is also a visiting researcher of the Department of Computer Science and Engineering at the University of California at San Diego. He received BS, MS, and PhD from University of Tokyo. Contact him at UCSD/CSE-AP&M 6414, 9500 Gilman Drive, Dept. 0114 La Jolla, CA 92093-0114 USA; tau@csag.ucsd.edu.

Andrew A. Chien is the Science Applications International Corporation Chair Professor in the Department of Computer Science and Engineering at the University of California at San Diego. Andrew Chien leads the Concurrent Systems Architecture Group and is involved with joint projects with both NCSA and NPACI . He received BS, MS, and PhD from the Massachusetts Institute of Technology. Contact him at UCSD/CSE-AP&M 4808, 9500 Gilman Drive, Dept. 0114 La Jolla, CA 92093-0114 USA; achien@cs.ucsd.edu.