

Improving Load Balance with Flexibly Assignable Tasks

Ali Pinar, *Member, IEEE Computer Society*, and Bruce Hendrickson, *Member, IEEE*

Abstract—In many applications of parallel computing, distribution of the data unambiguously implies distribution of work among processors. But, there are exceptions where some tasks can be assigned to one of several processors without altering the total volume of communication. In this paper, we study the problem of exploiting this flexibility in assignment of tasks to improve load balance. We first model the problem in terms of network flow and use combinatorial techniques for its solution. Our parametric search algorithms use maximum flow algorithms for probing on a candidate optimal solution value. We describe two algorithms to solve the assignment problem with $\log W_T$ and $|P|$ probe calls, where W_T and $|P|$, respectively, denote the total workload and number of processors. We also define augmenting paths and cuts for this problem, and show that any algorithm based on augmenting paths can be used to find an optimal solution for the task assignment problem. We then consider a continuous version of the problem and formulate it as a linearly constrained optimization problem, i.e., $\min \|Ax\|_\infty$, s.t. $Bx = d$. To avoid solving an intractable ∞ -norm optimization problem, we show that, in this case, minimizing the 2-norm is sufficient to minimize the ∞ -norm, which reduces the problem to the well-studied linearly constrained least squares problem. The continuous version of the problem has the advantage of being easily amenable to parallelization. Our experiments with molecular dynamics and overlapped domain decomposition applications proved the effectiveness of our methods with significant improvements in load balance. We also discuss how our techniques can be extended to heterogeneous parallel computers.

Index Terms—Parallel computing, load balancing, flexibly assignable tasks, maximum flow, constrained least squares.

1 INTRODUCTION

IN many applications of parallel computing, the distribution of data among processors implies a corresponding distribution of work. However, there are important exceptions to this rule that arise for one of two reasons. First, some portions of the data may be replicated on multiple processors, any of which could perform the associated work. Second, tasks may involve multiple data items which may not all reside on the same processor. Thus, all the interacting data will need to be combined on a single processor before the computation can be completed. In principle, any processor could perform this task (see, e.g., [1]), but for the purposes of this paper, we will consider only those processors owning a portion of the relevant data—other options would increase the communication requirements.

Examples of such *flexibly assignable* work are common in scientific applications. In molecular dynamics simulations, a force is computed between any pair of particles that are close to each other. For large problems, these calculations are usually parallelized by dividing the particles among the processors [2]. If two close-by particles reside on different processors, then either processor could perform the computation.

Another example arises in finite element simulations. These calculations consist of several computational phases,

some of which are element-based, while others are node-based. If, for instance, the mesh is partitioned so that processors own full elements, then nodes at the boundary between elements will be duplicated on at least two processors. Any of these processors could perform the node-based operations for these shared nodes. If, instead, the mesh is partitioned by nodes, then some elements will be divided among multiple processors. Any of these processors could be employed to perform the element computation.

A third example comes from an important class of preconditioners known as overlapped Schwarz domain decomposition [3], [4]. In this preconditioning scheme, processors perform one calculation on subdomains that overlap each other and another calculation on disjoint subdomains. With the overlapped domains, some portions of the data are duplicated on multiple processors. Any of these processors could perform the calculations for these duplicated objects in the disjoint portion of the computation.

The freedom to assign work to any of several processors raises the question of how best to exploit this flexibility. In this paper, we investigate using this freedom to improve load balance. That is, we want to give most of this flexibly assignable work to processors that would otherwise have too little to do. More formally, we address the following *task assignment problem*:

Given: A set of unit tasks and the (possibly singleton) set of processors that can perform them.

Find: An assignment of tasks to processors that minimizes the number of tasks assigned to the maximally loaded processor.

Despite its practical utility and the rich literature on formulations of load balancing problems (e.g., [3], [5], [6],

- A. Pinar is with the High Performance Computing Research Department, Lawrence Berkeley Lab, Berkeley, CA 94720-8139. E-mail: apinar@lbl.gov.
- B. Hendrickson is with the Discrete Algorithms and Math Department, Sandia National Laboratories, Albuquerque, NM 87185-1110. E-mail: bah@sandia.gov.

Manuscript received 9 Sept. 2003; revised 14 Oct. 2004; accepted 31 Dec. 2004; published online 22 Aug. 2005.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0158-0903.

[7]), to our knowledge, this particular problem has not been defined or addressed previously. After providing some basic definitions in Section 2, we investigate several combinatorial approaches to address the task assignment problem in Section 3. Our experimental results prove that significant gains in load balance can be achieved.

Besides formulating a new and practically important problem, this paper makes several technical contributions. First, we describe a parametric search solution that uses a standard maximum-flow solver as a probe function. This solution is simple to implement and allows for the use of any maximum flow solver as a black box.

Our second combinatorial algorithm involves a more detailed analysis of the structure of the problem. Specifically, we devise a maximum-flow/minimum-cut theorem for our nonstandard objective function. This result gives significant insight into the structure of the problem, and we use it to devise an augmenting path algorithm that mimics the structure of Ford-Fulkerson methods for maximum flows. The result is an asymptotically more efficient approach, but one that cannot be built upon standard maximum-flow solvers. It is also worth noting that our approach solves the problem of finding a maximum flow with the property that the largest flow on any terminal edge is minimized.

These combinatorial algorithms are sufficient for many problems in which a serial computation can determine the assignment as a preprocessing step to a parallel calculation. However, in some instances, the characteristics of the parallel computation change over time, and the assignment must be recomputed. Our combinatorial methods are not particularly amenable to parallelization. For this reason, in Section 4, we present a continuous approximation to the problem that leads to a more easily parallelized numerical approach. The continuous approximation is closely related to the diffusion methodology widely employed for determining work transfers in dynamic load balancing [5]. We show that the flexibly assignable work problem can be formulated as a linearly-constrained optimization problem, i.e., $\min \|Ax\|_\infty$, s.t. $Bx = d$. Here, the linear constraints $Bx = d$ guarantee that assignment of tasks is valid, and Ax is the vector of processor loads. Minimizing the ∞ -norm of this vector corresponds to minimizing the maximum processor load. Since it is a nonsmooth function, minimizing the ∞ -norm is difficult. However, we are able to show that in this context, minimizing the 2-norm is sufficient to minimize the ∞ -norm, which reduces the problem to the well-studied numerical kernel known as a linearly constrained least squares problem. We then show that there are efficient parallel approaches to solve this problem. Of course, the discretized solution to the continuous approximation may not be identical to the actual solution to the discrete problem.

The load balancing problem for heterogeneous systems is slightly different, since merely assigning equal amounts of work to processors is not sufficient and processor speeds must be taken into account. In Section 6, we discuss how our techniques can be enhanced for such systems and show that, by only minor modifications, all of our proposed methods can be enhanced for load balancing for heterogeneous systems.

2 PRELIMINARIES

A flow network is defined by a directed graph $G = (V, E)$, with a source vertex s , a terminal vertex t , and a capacity for each edge (i, j) , which we denote by $c(i, j)$.

We will define a flow f to be a function $f : E \rightarrow Z^+$ from edges to integers and use $f(i, j)$ to denote the volume of flow along edge (i, j) . A flow must satisfy the capacity constraints on edges (i.e., $f(i, j) \leq c(i, j)$) and the flow conservation constraints

$$\sum_{(i,k) \in E} f(i, k) = \sum_{(j,i) \in E} f(j, i) \text{ for all } i \in V \setminus \{s, t\}.$$

The value of a flow $|f|$ is defined by the flow leaving the source s ,

$$|f| = \sum_{(s,v) \in E} f(s, v).$$

A *maximum flow* (max-flow) is a flow that maximizes $|f|$.

A flow is *complete* if its value is equal to the cumulative capacity of edges leaving the source, i.e., $|f| = \sum_{(s,v) \in E} c(s, v)$. A graph that can support a complete flow will be called a *complete-flow* graph.

Given a graph G and flow f , the *residual graph* G_f has the same set of vertices as G and all edges in G (referred to here as *forward* edges), plus a matching set of *backward* edges that point in the opposite direction. The capacity of a forward edge is equal to its capacity in G minus the flow assigned to that edge in f . The capacity of a backward edge is equal to the flow on the corresponding forward edge.

In the max-flow problem, an augmenting path is defined as a path from s to t along which more flow can be pushed. The capacity of a path is defined by the minimum of the capacities of its edges. Any path in G_f from s to t with nonzero capacity is an augmenting path and can be used to increase the total flow.

Finding a maximum flow is a fundamental problem in combinatorial algorithms and has been the subject of numerous research efforts. Fundamentals of network flow algorithms can be found in [8], [9]. In a more recent work, Goldberg and Rao give a history of maximum-flow bounds and relevant references [10].

The assignment of tasks to processors can be modeled as a flow on a network $G = (T, P, E)$, where each task is represented by a vertex in T and each processor is represented by a vertex in P . All processor-vertices are connected to the terminal t by *terminal edges* and the source s is connected to all the task-vertices by *source edges*. Task-vertices have *assignment edges* connecting them to all the processors to which the associated task can be assigned. The graph can be simplified by combining all vertices that have identical sets of processor neighbors. We call such sets *task groups*. An example can be found in Fig. 1.

The capacity of an edge from the source to a task-vertex is defined by the size of the corresponding task group. We will set the capacities for assignment edges and terminal edges to be infinite. We will call this graph an *assignment* graph. Notice that assignment graphs are complete-flow graphs.

Fig. 1 illustrates an assignment graph. There are seven task groups and four processors. v_1 corresponds to a task

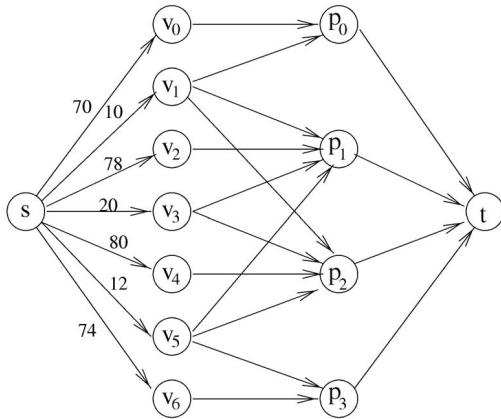


Fig. 1. Example of an assignment graph.

group of 10 tasks and can be assigned to processors p_0 , p_1 , and/or p_2 . Notice that some of the task groups can be assigned to only one processor. This situation often arises in practice and these tasks correspond to work that can be assigned only to a single processor.

We can consider the assignment of tasks as a flow from the task-vertices to processor-vertices. Our objective is to find a flow that assigns all the tasks to processors while minimizing the maximum load of any processor. We need the flow to be complete (full capacity of source edges is used) to guarantee assignment of all the tasks. We can define the flow problem as follows:

Given an assignment graph $G = (T, P, E)$, find a complete flow f in G that minimizes

$$\max_{p \in P} f(p, t). \quad (1)$$

Although we define our flow problem for very specific flow graphs, the algorithms and analysis in the following sections are valid for any complete-flow graph. For a general graph, we would redefine P in (1) to be the set of vertices that are connected to the terminal vertex (i.e., $P = \{v : (v, t) \in E\}$). Thus, the problem we are solving is equivalent to that of finding a maximum flow that minimizes the largest flow along any terminal edge.

In Fig. 2, a solution to an assignment problem is illustrated. Numbers on the edges correspond to flow assignments for these edges. This assignment gives an optimal solution, where p_1 and p_2 are the maximally loaded processors with 89 tasks. Out of 20 tasks of task group v_3 , 11 will be performed on p_1 and 9 will be performed on p_2 .

3 COMBINATORIAL ANALYSIS

As outlined in Section 2, assignment of tasks to processors can be formulated as a flow in a network. In this section, we will investigate the relation between classical flow problems and the task assignment problem. First, we will discuss parametric search solutions that use standard max-flow techniques as a probe function and describe a polynomial time algorithm for the task assignment problem. Then, we will show how Ford-Fulkerson methods can be used to solve our problem. Specifically, we will revise the definition of augmenting paths and cuts and show that any

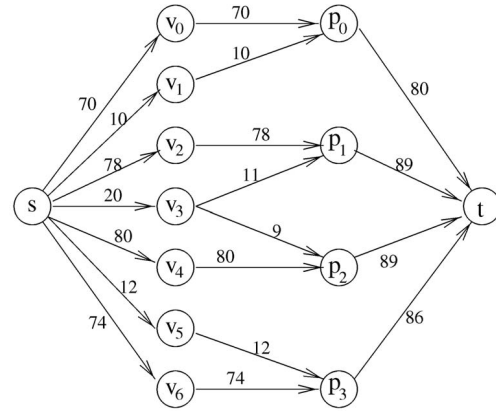


Fig. 2. Example of a solution on an assignment graph.

maximum-flow algorithm based on the Ford-Fulkerson method can be used for the task assignment problem.

3.1 Parametric Search

A parametric search algorithm has two components: a *probe* function that determines whether there is a solution with a cost less than a specified value, and a method to search on the space of candidate optimal solution values. Below, we first show how standard maximum-flow algorithms can be used as a probe function for our problem. Our cost function is the maximum work assigned to any single processor. Then, we discuss two techniques to search the space of candidate values. The following lemma formalizes our claim for maximum-flow algorithms being used as a probe function.

Lemma 1. *There is a solution to the task assignment problem with cost $\leq B$ if and only if there exists a complete flow on the modified graph where all terminal edges have capacity B .*

Proof. Construct the assignment graph as described in Section 2 and change the capacities of all terminal edges to B . We claim that there is a solution to the task assignment problem with cost $\leq B$ if and only if the maximum flow uses the capacity of all the source edges. Proof of this claim follows.

Sufficiency. Bounds on capacities of terminal edges guarantee that no processor is assigned more than B units of work and, if a flow uses all the source edge capacity, then all work is assigned to processors. Moreover, the flow solution provides the corresponding task assignments.

Necessity. Assume there is a solution to the task assignment problem where no processor is assigned more than B units of work. We can use the assignments of tasks in this solution to find a corresponding flow solution. \square

To solve the task assignment problem, we must find the minimal value of B for which a max-flow solution uses all the source edge capacities. We present two algorithms for finding this value in the following two subsections.

3.1.1 Bisection Search

Bisection search is a standard technique used in parametric search algorithms. It starts with a lower and an upper bound on the optimal solution value and discards half of the interval by probing on the midpoint of the current bounds. This gives an ϵ -approximation algorithm for real-valued solutions, but

finds an exact solution when the optimal solution value is an integer, as in the case of our problem.

For the task assignment problem, the total number of tasks is an upper bound on the cost of an assignment, and the number of tasks divided by $|P|$ is a lower bound. Thus, a bisection search gives the following result.

Theorem 1. *If W_T is the total number of tasks, bisection search solves the task assignment problem optimally with $O(\log W_T)$ probe calls.*

3.1.2 Incremental Search

An incremental search starts with a lower bound for the optimal solution value and increases it until the optimal value is found. The increments should be small enough to avoid missing the optimal value, but large enough for efficiency. The following lemma and theorem show how the lower bound can be increased after a failed probe call—that is, a max-flow problem with terminal edge capacities B in which not all the source edge capacity is utilized.

Lemma 2. *Let (u, t) be a terminal edge that is not saturated in a maximum-flow solution f for a probe value B (i.e., $f(u, t) < c(u, t) = B$). Then, for any probe value $B' > B$, there is a maximum-flow solution f' in which $f'(u, t) \leq f(u, t)$.*

Proof. When the Ford-Fulkerson method is used to achieve an optimal solution f' for bound $B' > B$ by using f as an initial solution, we can get an optimal solution with $f'(u, t) \leq f(u, t)$. First, note that u is not reachable from s in G_f , and increasing terminal edge capacities does not make u reachable from s . Furthermore, u will not be reachable while the flow is being modified via augmenting paths. Consider the first augmenting path that will add a vertex to the set of vertices from which u is reachable. Observe that such a path should reach a vertex that can reach to u , which contradicts u 's nonreachability from s . \square

Theorem 2. *For a failed probe with terminal edge capacity B , let $W_r > 0$ be the total unused source edge capacity and let K be the number of saturated terminal edges. Then, there is no feasible solution to the task assignment problem with cost less than $B + W_r/K$.*

Proof. By the result of Lemma 2, additional flow cannot go to any of the unsaturated terminal edges. In the best case, additional flow will be equally distributed among the set of saturated terminal edges. \square

As a result of Theorem 2, a failed probe value B can be increased to $B + \frac{W_r}{K}$, as exploited in Algorithm IncSearch.

Algorithm IncSearch

```

 $B \leftarrow W_T/|P|;$ 
while not Probe( $B$ )
  Let  $K$  be the number of saturated terminal
  edges
  and  $W_r$  be the total unused source edge capacity.
   $B \leftarrow B + \lceil \frac{W_r}{K} \rceil;$ 
return  $B$ ;
```

The following lemma proves that Algorithm IncSearch terminates and gives a bound on the number of probes it makes.

Lemma 3. *Algorithm IncSearch terminates after at most $|P|$ probes.*

Proof. When a probe value is increased, if all previously saturated edges remain saturated, then the probe call will succeed. Thus, when a probe fails, at least one new terminal edge is not saturated. That is, each failed probe decreases the number of saturated terminal edges by at least one. \square

Theorem 3. *Algorithm IncSearch finds an optimal solution and makes $O(|P|)$ probes.*

Proof. We start with a lower bound. According to Theorem 3, increases on B are minimal. Thus, we do not miss the optimal value. Lemma 3 ensures that the algorithm terminates after $O(|P|)$ probes with an optimal solution. \square

Notice that successive probes solve max-flow problems on the same graph in an incremental manner, where only the capacities of the terminal edges increase. Thus, the previous flow solution gives a feasible solution (though not optimal) for the next flow solution, which might be exploited for efficiency. Using the Ford-Fulkerson method in its simplest way will give a complexity of $O(W_T * |E|)$ for all the probes and, thus, for the algorithm.

3.2 Ford-Fulkerson Method

The Ford-Fulkerson method has been the basis of a number of algorithms to solve the max-flow problem. It is built on three basic concepts: *residual* graphs, *augmenting* paths, and *cuts* [8]. In this section, we will discuss how it can be adopted to the task assignment problem. First, we will revise the definitions of augmenting paths and cuts for the task assignment problem, then state and prove a version of the maximum-flow/minimum-cut theorem for the task assignment problem. The result will enable every algorithm based on the Ford-Fulkerson method to be used to solve our problem.

The generic Ford-Fulkerson method starts with a zero flow and continues to add to the flow along augmenting paths until no augmenting paths are left. In the task assignment problem, we will use augmenting paths to shift flow (tasks) from a maximally loaded terminal edge (processor) to a less loaded terminal edge. Formally, an augmenting path (pt, u, v) is a path pt in G_f that starts with the vertex u of a maximally loaded terminal edge (u, t) , ends at the vertex v of a less-loaded terminal edge (v, t) , and does not go through t . We define the *capacity* $c(pt, u, v)$ of an augmenting path to be the minimum of the capacities of its edges and half of the difference between the flow on the first and last terminal edges, rounded down to an integer,

$$c(pt, u, v) = \min \left(\left\lfloor \frac{f(u, t) - f(v, t)}{2} \right\rfloor, \min\{c_f(i, j) : (i, j) \text{ on } pt\} \right).$$

This implies that the capacity of a path between two processors whose loads differ by just one is zero since such an augmentation will not yield a more balanced distribution.

We can update flow assignment in the graph for edges on the path and the two terminal edges connecting processor-vertices to the terminal to obtain a more balanced distribution, as stated by the following lemma.

Lemma 4. Let f be the current flow, pt be an augmenting path, and $c > 0$ be the capacity of this path. Define $f^+(i, j)$ for all edges as

$$f^+(i, j) = \begin{cases} f(u, t) - c & \text{if } i = u, j = t \\ f(v, t) + c & \text{if } i = v, j = t \\ f(i, j) + c & \text{if } (i, j) \text{ is on } pt \\ f(i, j) - c & \text{if } (j, i) \text{ is on } pt \\ f(i, j) & \text{otherwise.} \end{cases}$$

Then, f^+ does not change the total flow (i.e., $|f^+| = |f|$), but decreases either the maximum load or the number of maximally loaded terminal edges.

Proof. For the total flow to change, we must decrease flow from s . This is possible only if there is an edge (v, s) in pt , but this edge must be followed by another edge (s, u) and, thus, total flow from s does not change. With the same argument, flow conservation constraints are satisfied for all vertices. Since pt does not go through t , augmentation will affect only two terminal edges: (u, t) and (v, t) . By definition of an augmenting path, (u, t) is a maximally loaded terminal edge, and we decrease its load. By definition of the capacity of an augmenting path, the load of (v, t) cannot be as high as (u, t) , after we increase it. \square

In traditional flow problems, a *cut* (S, T) in G is defined as a partition of vertices into S and T in which $s \in S$ and $t \in T$. The cost of a cut is defined as the sum of capacities of edges from S to T . The cost of a minimum cut and value of a maximum flow are equal. A minimum cut corresponds to a bottleneck in the flow from source to terminal. For the task assignment problem, we will define a *cut* as a bipartitioning (P_1, P_2) of the processor-vertices. The *cost* of a cut is defined to be the maximum load in P_1 when 1) processors in P_1 are equally loaded and 2) all the tasks that can be assigned to a processor in P_2 are assigned to processors in P_2 .

By "equally loaded," we mean that the loads of any two processors differ by at most one. Cuts will help to identify a bottleneck in the problem, just as in maximum flow problems. A bottleneck in our problem is a group of processors that have to perform a large set of tasks. Unlike the maximum flow problem, cuts provide lower bounds on the cost.

With the above definitions, we have the following maximum-flow/minimum-cut theorem for the task assignment problem.

Theorem 4. The following statements are equivalent:

1. Flow f minimizes the load of the maximally loaded processor.
2. There is no augmenting path that decreases the maximum load in G_f .
3. The maximum load is equal to the cost of a cut (P_1, P_2) .

Proof.

- (1) \implies (2). Assume the contrary, that there exists an augmenting path to decrease the maximum load. Then, we can use this path to decrease the maximum load, thus f is not an optimal flow.
- (2) \implies (3). Let P_1 be the set of processors with maximum load plus processors reachable from a

maximally loaded processor in G_f . The set P_2 contains the remaining processors. By construction, there are no augmenting paths from P_1 to P_2 . This guarantees that all tasks that could be assigned to either P_1 or P_2 processors are assigned to P_2 processors. Also, since there are no augmenting paths in G_f , the loads of all processors in P_1 are either equal to or one less than the maximum load. That is, the processors in P_1 are equally loaded.

- (3) \implies (1). Since all tasks which could be assigned to either P_1 or P_2 are assigned to processors in P_2 , the work currently assigned to processors in P_1 must be performed by processors in P_1 . The best we can do is to assign all work equally, which is guaranteed by the first condition in the definition of a cut, so f is an optimal solution. \square

Corollary 1. Every algorithm based on the Ford-Fulkerson method can be used to solve the task assignment problem.

It is worth noting that, although any algorithm based on the Ford-Fulkerson method might be used to solve this problem optimally, the complexity results might vary from those of the conventional max-flow problem.

Below, we present an algorithm, AugPath, which finds an optimal solution using augmenting paths.

Algorithm AugPath

```
find a complete flow  $f$  in  $G$ ;
while there is an augmenting path  $pt$ 
  augment flow along  $pt$ ;
```

Theorem 5. Algorithm AugPath finds an optimal solution in $O(|E| * \log |P| * W_T)$ -time.

Proof. Correctness of the algorithm is implied by Corollary 1. Finding an augmenting path takes $O(E)$ -time in the worst case. By Lemma 4, each augmenting path either decreases the maximal load or the number of maximally loaded processors. This gives a loose $|P| * W_T$ bound on the number of augmenting paths required. A better bound is possible, however. When the maximum load is in the range $[W_T/2, W_T]$, only one processor might have the maximum value and, thus, each augmenting path will decrease the maximum load by one. Generally, when the maximum is in the range $[W_T/(k+1), W_T/k]$, there can be at most k processors with the maximum load and k augmenting paths may be needed to decrease the maximum load. So, the total number of augmenting paths can be computed as

$$\begin{aligned} \sum_{1 \leq k \leq |P|-1} \left(\frac{W_T}{k} - \frac{W_T}{k+1} \right) k &= W_T \sum_{1 \leq k \leq |P|-1} \left(\frac{k+1-k}{k(k+1)} \right) k \\ &= W_T \sum_{1 \leq k \leq |P|-1} \frac{1}{k+1} \\ &= O(W_T * \log |P|). \end{aligned}$$

\square

Note that, since we are using the continuous formulation as an approximation to a discrete problem, a low accuracy numerical solution is sufficient.

One way to deal with linear equality constraints is the method of *weighting* [14]. This method moves the equality constraints into the objective function, but severely penalizes slack in these rows by weighting these equations. So, (3) is transformed to the form

$$\min_x \left\| \begin{pmatrix} \gamma B \\ A \end{pmatrix} x - \begin{pmatrix} \gamma d \\ 0 \end{pmatrix} \right\|_2 \text{ subject to } x \geq 0,$$

where γ is a large number used to penalize slack in the equality constraints. This transforms our problem to an instance of a nonnegative least squares problem,

$$\min_x \|Cx - b\|_2 \text{ subject to } x \geq 0.$$

The nonnegative least squares problem is equivalent to the following optimization problem:

$$\min_x \frac{1}{2} x^T E x - c^T x \text{ subject to } x \geq 0,$$

where $E = C^T C$ and $c = C^T b$. Cryer proposed the following SOR iteration for solving nonnegative least squares problems [15],

$$x_i^{k+1} = \max \left\{ 0, x_i^k - \frac{\omega}{e_{ii}} \left(c_i - \sum_{j < i} e_{ij} x_j^{k+1} - \sum_{j \geq i} e_{ij} x_j^k \right) \right\},$$

where ω is the overrelaxation parameter. This is a standard stencil operation and requires only local communication with neighbor processors. Hence, it is amenable to efficient parallelization.

The least squares technique will give a nonintegral solution, which needs to be discretized for task assignment. The continuous solution can be easily mapped to a feasible solution by adjusting the assignments for each task group. Consider a group of tasks that can be assigned to either p_0 or p_1 . We can round the total assignment to p_0 up to an integer and assign that many tasks to p_0 and assign the remainder to p_1 . Notice that this adjustment only has local effects and is easy to generalize for more processors. We do not have any bounds on the impact of such rounding operations on solution quality.

5 EXPERIMENTAL RESULTS

We have applied our techniques to problems from two application domains—molecular dynamics and overlapped domain decomposition. In each case, as described in Section 4, we solved the least squares formulation of the problem in serial and used Gauss-Seidel iterations [16] to generate a new distribution of work among processors. We define load imbalance as $(\max - \text{avg}) * 100 / \text{avg}$, where \max and avg denote the maximum and average processor load, respectively.

For the molecular dynamics application, we used data provided by Plimpton which came from his spatial decomposition code [2]. In a molecular dynamics simulation, the work is dominated by the number of forces that need to be computed between pairs of nearby atoms. In this

TABLE 1
Results on Molecular Dynamics Problems

Problem	N	F	P	Load Imbalance	
				Initial	Improved
membrane	7,134	4,052,598	8	5.80	0.32
			64	21.17	0.21
			512	58.34	2.16
mixer.half.1	4,818	17,220	8	121.55	110.78
			64	163.70	134.07
mixer.half.2	4,818	17,360	8	146.48	130.40
			64	195.59	155.15

code, a bounding box encloses all the atoms, the box is divided into P regions of equal volume, and each of P processors is responsible for atoms residing within one of the boxes. Flexibility arises when two atoms belonging to different processors are close enough to interact. Larger interaction cutoffs and smaller regions each increase the fraction of flexibly assignable work. If the atoms are uniformly distributed through the bounding box (e.g., for simulations with periodic boundaries), then the load will generally be well balanced. But, for problems in which the atom distributions are inhomogeneous, significant load imbalance arises.

We present results for two types of problems in Table 1. In this table, N represents the number of particles and F is the total number of pairwise force computations. A simple way to partition a set of flexibly assignable force computations is to assign half to each of the two processors they span. The load imbalance induced by this strategy is detailed in column “Initial” in the table. Column “Improved” contains the load imbalance resulting from using our least squares algorithm to assign work. Clearly, our approach can result in a significant reduction in load imbalance. It is worth noting that these are problems of modest size and, so, do not require very large numbers of processors.

Of the problems in Table 1, the first comes from a biological simulation of a membrane in which the atomic densities are higher within the membrane than within the surrounding fluid. For large numbers of processors, we are able to reduce the load imbalance from an initial 58 percent to just over 2 percent. The next two problems are instances of a simulation of a rotating drum being used to mix solid particles, as described in [17]. The particles fill only a fraction of the volume of the drum, leading to significant load imbalance. Specifically, some of the processors are responsible for regions of space that have few or no particles. For this problem, the particles are treated as rigid bodies and, so, the cutoff distances are very short. As a consequence, there are few flexibly assignable interactions, which limits our ability to improve load balance. Despite these inherent difficulties, we are still able to significantly improve the overall load balance.

The second data set comes from an important class of preconditioners known as overlapped Schwarz domain decomposition [3], [4]. In this preconditioning scheme, processors perform one phase of the calculation on

TABLE 2
Results for the Overlapped Subdomain Preconditioners

Matrix	N	NNZ	P	Load Imbalance	
				Initial	Improved
Braze	1,344	142,296	4	1.25	0.00
			8	31.52	0.34
			16	77.25	1.36
			32	126.47	43.16
Defroll	6,001	173,718	4	13.28	0.08
			8	25.47	7.28
			16	17.89	5.86
			32	41.66	15.94
DIE3D	9,873	1,723,498	4	20.60	0.13
			8	28.52	0.19
			16	91.39	2.50
			32	138.07	9.08
dday	21,180	1,033,324	4	1.52	0.14
			8	3.59	0.07
			16	13.08	0.55
			32	18.26	0.75
visco	23,439	1,136,966	4	16.28	0.30
			8	34.19	1.77
			16	53.11	3.41
			32	84.49	5.52
sls	36,771	2,702,280	4	2.28	0.15
			8	14.36	0.01
			16	16.88	0.07
			32	29.74	0.80
ocean	143,437	819,186	4	4.92	0.21
			8	9.22	1.43
			16	17.39	3.35
			32	41.61	6.15

subdomains that overlap each other, and another phase on disjoint subdomains. To achieve high performance, it is important to balance the load of each phase. To accomplish this, we first choose a balanced set of overlapped subdomains [3]. For the disjoint phase, we can then exploit flexibility in task assignment. Specifically, overlapped portions of the initial decomposition correspond to duplicated data and any processor owning that data can perform the associated task in the disjoint phase. Table 2 displays our experimental results on a set of sparse test matrices arising from applications where overlapped subdomain preconditioners are used [3]. In this table, N denotes the number of rows and columns of the matrix, NNZ is the number of nonzeros in the matrix, and P is the number of processors. These are all modest sized problems and, so, do not require a very large number of processors. The column labeled “Initial” describes the load imbalance associated with assigning the disjoint subdomains to processors. The column labeled “Improved” details the load imbalance resulting from our least squares solution. The results indicate that significant improvements in load balance are achieved by our techniques.

It is worth noting that a few Gauss-Seidel iterations are sufficient for the algorithm to converge and each iteration consists of a simple traversal of tasks. Thus, our algorithms are efficient and solution times are negligible.

6 HETEROGENEOUS SYSTEMS

Discussions prior to this section were limited to homogeneous systems, where all processors have identical execution speeds. This assumption does not hold, however, for many current parallel systems; processors are often heterogeneous, i.e., they have different processing powers. In particular, clusters of workstations, which are very likely to have heterogeneity, are gaining popularity and significant research efforts are devoted to their development. Heterogeneity of the processors might lead to load imbalance even when all processors are assigned equal amounts of work. Varying processor speeds should be taken into account while distributing the work to avoid processor idle times. The objective must be to minimize the maximum completion time among all processors, as opposed to minimizing the maximum load.

In the following sections, we will discuss how our techniques can be extended to exploit flexibly assignable tasks to improve balance for heterogeneous processors. We define the execution speed of a processor to be the number of unit operations it can perform in a unit time, and use e_i to denote the execution speed of the i th processor.

6.1 Parametric Search Algorithms

To employ a parametric search algorithm, we first need a probe function. For heterogeneous problems, we can still use a max-flow solver for probing after minor modifications. Here, probe values will be the completion time of processors, i.e., $Probe(B)$ decides if there is an assignment of tasks for which all processors can complete in B units of time. By defining the capacity of a terminal edge (u, t) as $B * e_u$, where e_u is the processing speed of the respective processor, we can guarantee that all processors will complete in B units since the work assigned to a processor is proportional to its execution speed. We also need all tasks to be assigned to processors, which is guaranteed by a complete flow in the graph. Thus, a complete flow in the graph gives a feasible solution to the task assignment problem. Similarly, a feasible assignment of tasks defines a flow on the graph and, thus, a max-flow solver can be used to determine the existence of a solution within a specified completion time.

6.1.1 Bisection Search

The bisection algorithm, as described in Section 3.1, requires a lower and an upper bound, in addition to a probe function. A lower bound LB to start can be chosen as the ideal completion time, where the load is distributed perfectly and, thus, all processors finish at the same time. That is, we can choose a lower bound LB as

$$LB = \frac{W_T}{\sum_{i=1}^{|P|} e_i}.$$

An upper bound UB can be found by assigning all tasks to the fastest processor, i.e.,

$$UB = \frac{W_T}{\max_{i=1}^{|P|} e_i}.$$

An optimal solution for the task assignment problem can be found by using binary search on the interval

$$\left[\frac{W_T}{\sum_{i=1}^{|P|} e_i}, \frac{W_T}{\max_{i=1}^{|P|} e_i} \right].$$

6.1.2 Incremental Search

Incremental search requires, in addition to a probe function, a lower bound and a strategy to increase this bound after each probe. We already described a lower bound to be used for the bisection search. The strategy we used in Section 3.1 to increase the lower bound was based on trying to distribute the unassigned work evenly among active processors. We can use the same idea for heterogeneous systems, but this time we should consider the total execution power—as opposed to the number—of active processors. Let A_f be the set of processors whose terminal edges are saturated in flow f , i.e., $A_f = \{u : f(u, t) = c(u, t)\}$, and let W_r be the total volume of unassigned work or in graph theoretical terms, the total unused source edge capacity in f . In Lemma 2, we showed that increasing the capacity of an unsaturated terminal edge will not increase the total from source to the terminal, thus unassigned tasks after a probe need to be distributed among processors whose terminal edges are saturated. This means, after a failed probe value B , we can increase the bound by

$$\frac{W_r}{\sum_{u \in A_f} e_u},$$

which corresponds to distributing the remaining unassigned work evenly (with respect to computational speeds) among the active processors. Notice that an optimal solution can be found after at most $|P|$ probes since each probe either succeeds or turns an active processor to inactive, as in Section 3.1.

Theorem 7. *An incremental search algorithm that starts probing with*

$$B = \frac{W_T}{\sum_{i=1}^{|P|} e_i},$$

and increments the bound by

$$\frac{W_r}{\sum_{u \in A_f} e_u},$$

finds an optimal solution and makes $O(|P|)$ probes.

The proof is very similar to the proof sequence for Theorem 3, hence will not be repeated here.

6.2 Ford-Fulkerson Method

The Ford-Fulkerson method of Section 3.2 can be enhanced to solve the load balancing problem for heterogeneous systems through minor modifications of the basic definitions. A residual graph G_f for a flow f is defined the same

way as in Section 2. An augmenting path is still a path in the residual graph from the vertex of an overloaded terminal edge to the vertex of an underloaded terminal edge and, thus, helps us to shift work from an overloaded processor to an underloaded processor for better load balance. The capacity of a path is a function of the edge capacities on the path and the load difference between the two processors. Edge capacities constrain how much work can be shifted from the first processor to the last processor. After shifting load from one processor to another, we want the execution times of two processors to be equal for better balance. With simple algebra, we can show that we need to shift

$$d(u, v) = \frac{e_v f(u, t) - e_u f(v, t)}{e_u + e_v}$$

units of work from processor u to processor v to equalize the loads of these processors. The capacity of an augmenting path $c(pt, u, v)$ can then be defined by the minimum of $d(u, v)$ and the minimum edge capacity along the augmenting path $\min\{c_f(i, j) : (i, j) \text{ on } pt\}$. However, integrality constraints should be taken into account, thus formally the capacity of a path pt is defined as

$$c(pt, u, v) = \begin{cases} \min(\lfloor d(u, v) \rfloor, \min\{c_f(i, j) : (i, j) \text{ on } pt\}) & \text{if } e_u > e_v \\ \min(\lceil d(u, v) \rceil, \min\{c_f(i, j) : (i, j) \text{ on } pt\}) & \text{otherwise.} \end{cases}$$

In this definition, we use two conditions to assign the fragmented unit of work to the faster of the two processors.

A cut is still defined as a bipartitioning of processors (P_1, P_2) . The cost of a cut is the maximum load among P_1 processors when all tasks that can be assigned to a P_2 processor are assigned to a P_2 processor and remaining tasks are distributed so that the completion times of P_1 processors are equal.

With the modified definitions, Theorem 4 still holds, so we can use any algorithm based on augmenting paths to solve the flexibly assignable task problem for optimal load balancing on heterogeneous systems.

6.3 Numerical Formulation

In Section 4, we used the Ax vector to define the loads of processors. To determine the completion time for each processor, we define a diagonal matrix E with the i th diagonal entry being the reciprocal $1/e_i$ of the execution speed of the i th processor. Then, EAx defines the completion times of processors and minimizing $\|EAx\|_\infty$ will minimize the maximum completion time among all processors. Thus, we can state the load balancing problem for heterogeneous systems as follows:

$$\min_x \|EAx\|_\infty \text{ subject to } Bx = d \text{ and } x \geq 0.$$

In Section 4, we showed that any solution that minimizes $\|Ax\|_2$ also minimizes $\|Ax\|_\infty$. Observe that our proof is still valid when Ax is replaced by EAx , thus our numerical formulation reduces the problem of assigning flexibly assignable tasks for optimal load balance on heterogeneous systems to the linearly constrained least squares problem.

7 CONCLUDING REMARKS

We have posed and addressed the problem of distributing flexibly assignable work to processors to minimize load imbalance. This paper considers the problem in a general form, whereas exploiting problem-specific information might yield more efficient solutions. For instance, in the molecular dynamics application and in many other cases, each task can be assigned to one of at most two processors. We can exploit this fact to formulate the problem as a bounded least squares problem, $\min \|Ax + b\|$ s.t. $0 \leq x \leq u$, where u is a vector of upper bounds on decision variables. This formulation grants simpler and more efficient solution techniques than the more general linearly constrained least squares formulation.

We also suggest several research directions. First, the structure of this problem may allow specialization of flow techniques. It will be interesting to investigate if and how the advanced techniques for max-flow problems can be suited to our problem for more efficient combinatorial algorithms. Second, it would be helpful to generalize these techniques for nonunit tasks. Although the general problem corresponds to number partitioning (and NP-hard problem), one can look at special cases like Cartesian partitions, as in the case of molecular dynamics applications. Finally, using these techniques in different applications will be interesting. We keep identifying new sources of flexibly assignable tasks where our techniques can be used to improve load balance.

ACKNOWLEDGMENTS

This work was funded by the Applied Mathematical Sciences program, US Department of Energy, Office of Energy Research, and performed at Sandia, a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the US DOE under contract number DE-AC-94AL85000. Ali Pinar is also supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the US Department of Energy under contract DE-AC03-76SF00098. The authors are indebted to Erik Boman for pointing out the linear programming formulation in Section 4.

REFERENCES

- [1] L. Kalé, M. Bhandarkar, and R. Brunner, "Load Balancing in Parallel Molecular Dynamics," *Proc. Fifth Int'l Symp. Solving Irregularly Structured Problems in Parallel*, 1998.
- [2] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *J. Computational Physics*, vol. 117, pp. 1-19, 1995.
- [3] A. Pinar and B. Hendrickson, "Partitioning for Complex Objectives," *Proc. Int'l Parallel and Distributed Processing Symp.*, 2001.
- [4] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge Univ. Press, 1996.
- [5] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel Distributed Computing*, vol. 7, pp. 279-301, 1989.
- [6] B. Hendrickson and T. Kolda, "Graph Partitioning Models for Parallel Computing," *Parallel Computing*, vol. 26, pp. 1519-1534, 2000.
- [7] R.L. Carino and I. Banicescu, "Load Balancing Parallel Loops on Message-Passing Systems," *Proc. 14th IASTED Int'l Conf. Parallel and Distributed Computing and Systems*, pp. 362-367, 2002.

- [8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. Cambridge, Mass.: MIT Press and McGraw-Hill, 1990.
- [9] R.E. Tarjan, *Data Structures and Network Algorithms*. SIAM, 1983.
- [10] A.V. Goldberg and S. Rao, "Beyond the Flow Decomposition Barrier," *J. ACM*, vol. 45, pp. 783-797, 1998.
- [11] V. Ramachandran, "The Complexity of Minimum Cut and Maximum Flow Problems in an Acyclic Network," *Networks*, vol. 17, pp. 387-392, 1987.
- [12] R. Diekmann, A. Frommer, and B. Monien, "Efficient Schemes for Nearest Neighbor Load Balancing," *Parallel Computing*, pp. 789-812, 1999.
- [13] R. Elsässer, B. Monien, and R. Preis, "Diffusive Load Balancing Schemes on Heterogeneous Networks," *Proc. 12th ACM Symp. Parallel Algorithms Architecture (SPAA)*, pp. 30-38, 2000.
- [14] Å. Björck, *Numerical Methods for Least Squares Problems*. SIAM, 1996.
- [15] C. Cryer, "The Solution of a Quadratic Programming Problem Using Systematic Overrelaxation," *SIAM J. Control and Optimization*, vol. 9, pp. 385-392, 1971.
- [16] A. Dax, "Bounded Least Squares Problem," *ACM Trans. Math. Software*, 1991.
- [17] L. Silbert, D. Ertas, G. Grest, T. Halsey, D. Levine, and S.J. Plimpton, "Granular Flow Down an Inclined Plane: Bagnold Scaling and Rheology," *Physica Rev. E*, vol. 64, p. 51302, 2001.



ACM, and the IEEE Computer Society.

Ali Pinar received the BS and MS degrees in computer engineering from Bilkent University, Turkey, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He has been working at Lawrence Berkeley National Laboratory since 2001. His research interests include combinatorial scientific computing, combinatorial algorithms, parallel algorithms, and electric power systems. Dr. Pinar is a member of SIAM, the



Bruce Hendrickson received degrees in mathematics and physics from Brown University, followed by the PhD degree in computer science from Cornell University. He has been at Sandia National Labs in Albuquerque since 1990, where he is currently a distinguished member of the technical staff in the Discrete Algorithms and Math Department. He also holds an appointment in the Computer Science Department at the University of New Mexico. Dr. Hendrickson is the author of more than 70 peer reviewed scientific papers and a number of patents, and has served on the editorial boards of several SIAM and IEEE journals. His interests include combinatorial scientific computing, parallel algorithms, linear algebra, graph algorithms, scientific software, and data mining. He is a member of the IEEE and 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.