Yale University
Department of Computer Science

On the Power of Preemption

Jeffery Westbrook[1]

YALEU/DCS/TR-999
January 1994

[1]Department of Computer Science, Yale University, New Haven, CT 06520. Email: westbrook@cs.yale.edu.

Abstract

We study the use of preemption in on-line load-balancing problems. A centralized scheduler must assign tasks to servers, processing on-line a sequence of task arrivals and departures. The goal is to keep the assignment of tasks well-balanced; that is, to minimize the maximum load on any processor. The scheduler may occasionally preempt and reassign tasks, in an attempt to decrease the maximum load. Only a limited amount of preemption is allowed, however. We give a general strategy, based on the idea of a *witness*, and then apply it to derive solutions to various load balancing problems, including those of identical and related machines, restricted assignment tasks, and virtual circuit routing. In each case, only a limited amount of preemption is used but the load is kept substantially lower than possible without preemption.

1 Introduction

A number of recent papers have studied on-line or dynamic load balancing problems [1, 2, 4, 3, 5, 6, 10, 12]. A typical dynamic load balancing problem consists of a fixed set V of n servers and a dynamic set of tasks U that arrive and depart on-line. As soon as each task arrives, it must immediately be assigned to some server, increasing the load on that server. Eventually the task completes and departs. The goal is to keep the maximum load on any server minimized. A fundamental aspect of the problem is that arrival and departure times are determined by an unknown adversary. An on-line load-balancing algorithm knowns neither what the next task will be nor when any currently existing task will terminate. In this paper, we consider the following load balancing problems.

- Identical Machines. Each task u has an associated weight w_u, and can be served by any one of the servers. The load on server v is the sum of the weights of tasks assigned to it.
- Related Machines. The same as the previous problem, except that each server v has a capacity, cap_v. The load on v is the sum of the assigned weights divided by the capacity.
- Restricted Assignment. Each task has a unit weight but can only be served by one of some subset of the machines. The load on a machine is the number of assigned tasks.
- Virtual Circuit Routing. The servers are the edges of an undirected graph. Each edge has an associated capacity cap_e. A task u is a request for a connection between two nodes s_u, t_u. The set of servers assigned to the task must form a path between s_u and t_u. Each task has an associated weight w_u. The load on an edge is the sum of the weights of the connections using that edge divided by its capacity.

Although virtual circuit routing is the only problem that explicitly addresses communication networks, most of these load balancing problems can occur in heterogeneous networks containing workstations, I/O devices, etc. Servers correspond to communication channels and tasks to requests for communication links between devices. Task weight reflect desired

bandwidth. A network controller must coordinate the channels so that no channel is too heavily loaded. The request sequence is unknown in advance. Load balancing problems like those above also arise in assigning computational subtasks of a distributed or parallel computation to timesharing servers. Such situations arise in programs written using LINDA [7, 8]. An even distribution of load provides fairness in scheduling and helps guarantee that the computation does not get held up too long at synchronization barriers waiting for all tasks to catch up. This improves throughput. Finally, the restricted assignment problem arises in implementation of fast network flow algorithms [11, 12].

The *active tasks* at time t are those tasks that have arrived but not yet departed. An optimal assignment of the active tasks minimizes the maximum load. Let λ_t^* be the maximum load in an optimal assignment of the tasks active at time t. For an on-line load-balancing algorithm A, let $A(t)$ denote the maximum load on any server at time t if A is used to determine assignment of tasks to servers. Note that λ_t^* depends only on the set of active tasks at time t, whereas $A(t)$ in general depends on the entire history of task arrivals and departures.

We say A is c-competitive against *peak load* if at for all t, $A(t) \leq \max_{t' \leq t} c\lambda_{t'}^*$. That is, the maximum load on an on-line server is bounded by c times the maximum optimal load ever seen. We say A is c-competitive against *current load* if for all t, $A(t) \leq c\lambda_t^*$. That is, the maximum load on an on-line server is no more than c times the maximum load in an optimal assignment of the currently active tasks.

The distinction between peak load and current load is quite significant. For example, consider Graham's [9] greedy heuristic for the identical machines problem, which simply assigns each new task to the least loaded server, and never reassigns a task. It is noted in [4] that this algorithm is 2-competitive against peak load. On the other hand, an algorithm that never reassigns tasks cannot be better than n-competitive against current load. An adversary may generate n^2 unit cost tasks. Some server v must have load at least n. The adversary then deletes all tasks except for those on v.

With the exception of [12], all previous papers on load balancing have given algorithms that are competitive against peak load, but not against current load. The peak load measurement is useful in network design, since if it is known that the peak optimum load will be limited by some value, one can add enough server capacity to guarantee that the on-line scheduler will never load a server beyond its capacity. On the other hand, the current load measure is more appropriate if one is interested in providing fairness in processor utilization and in guaranteeing that parallel subtasks can proceed at roughly the same rate. The behavior of load-balancing algorithms that are competitive against current load cannot be skewed by some transient peak. The current load measure is also required in the application of load balancing to network flow described in [12]. Naturally, an algorithm that is c-competitive against current load is c-competitive against peak load.

The algorithms described in this paper are competitive against current load. As observed above this requires some amount of preemptive rescheduling. The use of preemption, however, presents other problems. We cannot simply measure an on-line algorithm by a competitive ratio, since if the algorithm is free at any point to reassign all tasks, it can always guarantee a competitive ratio of 1. This would generally be undesirable in prac-

tice. Preempting a task can be an expensive process and one may be willing to accept an imbalance in the load in order to avoid the expense of reassignments.

To measure the cost of reassignment, we assume that each task u has an associated *restart cost*, r_u. The restart cost is incurred every time u is assigned or reassigned to some server or connection path. We assume that r_u is time-invariant. The algorithms in this paper are designed so that the total cost of all reassignments is bounded by some small function, usually linear, of the sum of the startup costs of all tasks, denoted S. Since each task incurs its restart cost at least once, when it is first assigned, the restart cost is at least S. The algorithms we present are designed and analyzed under the assumption that $r_u = w_u$. In the final section we give a general method to remove this restriction, at the price of an increase in the competitive ratio. It is useful to distinguish between *lazy* and *eager* rebalancing. In lazy rebalancing, tasks are rebalanced only in response to other tasks being deleted from the system. In eager rebalancing, tasks may be reassigned after both arrival and departure of other tasks. If tasks never depart, however, no lazy algorithm can improve on the lower bounds known for permanent tasks without preemption.

We first present a general *witness-based* strategy for constructing algorithms competitive against current load. We then give an algorithm for the identical machines case that is 16-competitive with total restart cost $2S$. Section 3 gives a 24-competitive algorithm for related machines with total restart cost $4S$. Previously, Azar *et al.* gave an algorithm for related machines that is 20-competitive against peak load [3]. Their algorithm is non-preemptive, *i.e.*, the restart cost is S.

In Section 4 we study the restricted assignment problem, and give an eager rebalancing scheme for the case that $w_u = 1$ for all weights. The scheme is parameterizable to trade of competitive ratio against restart cost; its best competitive ratio is $O(1)$ at a restart cost of $O(S \log n)$. Previously, Phillips and Westbrook [12] give a preemptive algorithm that is $O((\log n)/\rho)$-competitive against current load while incurring restart cost ρS, where $0 < \rho \leq 1$ is a user-specified parameter. Their algorithm works for arbitrary weights. Azar *et al.*[2] give an eager algorithm for the case of unit weights that is $O(1)$ competitive against peak load *if the optimum peak load is* $\Omega(\log n)$, and incurs restart cost $O(S(\log n))$.

In Section 5 we give an algorithm for virtual circuit routing that is $O(\log n)$-competitive against current load, with restart cost $O(S \log n \log(C/\mathrm{cap}_{min}))$, where C is the sum of edge capacities and cap_{min} is the minimum edge capacity. This algorithm is based on an algorithm of Azar *et al.*[2] that is $O(\log n)$ competitive against peak load and incurs an assignment cost $O(S \log n)$. Finally, in Section 6 we give a general method to remove the restriction that $r_u = w_u$.

2 Witness-Based Algorithms for Preemptive Balancing

The paradigm used in all sections of this paper is as follows. Assume for the moment a known lower bound λ_0 on the minimum possible load. We maintain a set of levels i for $i = 0, 1, 2, \ldots$. Each level is treated as a separate load balancing game, managed by a scheduling algorithm that obeys the following rules:

1. The load on a server due to jobs assigned to level i is $c(n)\lambda_i$, where $c(n)$ is some function of n and $\lambda_i = 2^i\lambda_0$. The scheduler is allowed to reject attempted insertions of tasks as necessary to maintain this condition.

2. If task t is rejected by level i, then the current optimum load is at least λ_i. Task t is called a *witness for load* λ_i.

3. For all $i < q$, then in some level $j > i$ there is a witness for load λ_{j-1}.

We call this a witness-based strategy.

Lemma 1 *An on-line algorithm that satisfies the conditions for a witness-based strategy is* $4c(n)$*-competitive against current load.*

Proof: The on-line load is at most

$$\sum_{i=0}^{q} 2^i c(n)\lambda_0 \leq 2c(n)\lambda_0 2^q, \tag{1}$$

where q is the maximum level. The existence of a witness for level $q-1$ implies that the optimum load is at least $\lambda_0 2^{q-1}$. Combining this with equation 1 gives the lemma. □

2.1 Example: identical machines

We now give an example for the case of identical machines. There are several non-preemptive algorithms that are competitive against peak load with a ratio $2-\epsilon$ for some small constant ϵ [6, 9, 10], but as mentioned above, none of these are better than n-competitive against current load. Although the related machines case of the following section subsumes this case, the algorithm described here is simpler than that for related machines and achieves a better competitive constant.

Let $W_{i,j}$ be the current total weight of all tasks assigned to server j in level i. Then the load on server j is simply $\sum_{i=0}^{q} W_{i,j}$.

To insert task u, compute the minimum level i such that there is some server j satisfying $W_{i,j} + w_u \leq 4\lambda_i$. Assign u to level i, server j and increase $W_{i,j}$ by w_u.

To delete task t from server j in level i, decreasing $W_{i,j}$ by w_t and apply the following rebalancing procedure.

1. Select a task u currently assigned to some server k in the maximum level, q.

2. Locate a server j satisfying $W_{q-1,j} + w_u \leq 2\lambda_{q-1}$.

3. If there is no such server, u is a witness for load λ_{q-1}. Terminate. Otherwise, reassign u to server j in level $q-1$. Decrease $W_{q,k}$ by w_u and increase $W_{q-1,j}$ by w_u. Goto to step 1 and repeat.

To show that this is a witness algorithm, we need to check that the three defining conditions hold. Trivially, the maximum load on level i is $4\lambda_i$. We confirm that if level $q-1$ rejects task u, the load is at least λ_{q-1}. Consider two cases. On the one hand, suppose $w_u > \lambda_{q-1}$. Then the optimum load is trivially greater than λ_{q-1}. On the other hand, suppose $w_u \leq \lambda_{q-1}$. Then for all $1 \leq j \leq n$, $W_{q-1,j} > \lambda_q$, else we could have placed u on some server. Hence the total weight of all tasks in the system is $> n\lambda_{q-1}$ and by the pigeonhole principle the optimum algorithm must have load $> \lambda_{q-1}$. It is easily verified by induction on the number of insertions and deletions that at all times, there is a witness for any level less than the current maximum.

Theorem 2 *The witness algorithm maintains load within* 16 *times the current load. Over all assignments and reassignments, the total weight that is reassigned* $2W$*, where* $W = \sum_{u \in U} w_u$.

The competitiveness of the algorithm follows directly from the fact that the algorithm is a witness algorithm. The bound on the total reassigned weight follows from a potential function analysis. Since a very similar analysis is given in the next section, we omit it from this example. Under the assumption that $w_u = r_u$, the total cost of all reassignments is $2S$, where $S = \sum_{u \in U} r_u$.

3 Related Processors

In this section, we consider the case that each server i has a particular capacity or throughput cap_i. Let W_i denote the sum of the weights of tasks assigned to server i. Then the load on server i is given by W_i/cap_i. A non-preemptive algorithm for permanent tasks is given in [1].

Without loss of generality, we assume that $\text{cap}_i \geq \text{cap}_{i+1}$ for all $1 \leq i < n$. We begin with a lazy rebalancing scheme that keeps the load bounded by 3λ for a given parameter λ, but that may reject insertions in doing so. This scheme is an adaptation of the algorithm of [1]. Let W_j be the sum of the weights of tasks assigned to processor j. We also define a quantity M_j, which will roughly be the maximum weight ever on processor j.

Insertion. Let u be the new task, with weight w_u. Let j be the maximum such that $(W_j + w_u)/\text{cap}_j \leq 3\lambda$. Assign task u to processor j, increase W_j by w_u, and set $M_i = \max\{W_i, M_i\}$. If there is no such j, then reject task u.

Deletion. Let u be the job that is departing, say from processor i. Decrease the value of W_i by w_t. Then apply the rebalancing procedure described below.

1. Let $x = \max\{j \mid \sum_{i=0}^{j} M_i - 2W_i \geq 0\}$
2. If there is no such x, then stop.

3. Otherwise, for all $i \leq j$, set $M_i = 0$, $W_i = 0$. Preempt each job t currently on such a processor and re-insert it using the insertion algorithm.

Lemma 3 *Suppose task u is rejected upon attempted insertion. Let λ^* be the maximum load in the optimum assignment of all the tasks currently active, including u. Then $\lambda^* > \lambda$.*

Proof: Let k be minimal such that $M_k/\text{cap}_k < 2\lambda$. If there is no such k, define $k = n+1$.

Suppose $k = 1$. Since u is rejected it must be the case that $(W_1 + w_u)/\text{cap}_1 > 3\lambda$. Thus we conclude $\lambda^* \geq w_u/\text{cap}_1 > \lambda$.

Suppose $k > 1$. For all servers $j < k$, $M_j/\text{cap}_j \geq 2\lambda$. The deletion rebalancing routine guarantees the property that $\sum_{i=1}^{j}(M_i - 2W_i) < 0$, for all $1 \leq j \leq k$. Let X be the set of tasks currently assigned to some server $j < k$.

Consider any optimal assignment in which all tasks in X are assigned to servers $j < k$. Let W_j^* be the weight on j in the optimal assignment. We have $\lambda^* \geq W_j^*/\text{cap}_j$ for all $j < n$. Hence $\lambda^* \sum_{i=1}^{k-1} \text{cap}_i \geq \sum_{i=1}^{k-1} W_i^* \geq \sum_{i=1}^{k-1} W_i > \sum_{i=1}^{k-1} M_i/2 \geq \lambda \sum_{i=1}^{k-1} \text{cap}_i$. Thus $\lambda^* > \lambda$.

Now consider any optimal assignment in which some task $x \in X$ is assigned to some server $j \geq k$. At the time that task x was last reassigned, all servers $j \geq k$ had load at most M_j. This follows from the fact that M_j is decreased only in a rebalancing operation, at which time all jobs on servers numbered lower than j are reassigned. Since x was not placed on server k, it must be the case that $(M_k + w_x)/\text{cap}_k > 3\lambda$, and since $M_k/\text{cap}_k < 2\lambda$, it follows that $w_x/\text{cap}_j \geq w_x/\text{cap}_k > \lambda$ for all $j \geq k$. Hence $\lambda^* > \lambda$. □

This basic algorithm can be used to give an algorithm that is 12-competitive against peak load. To get a witness-based algorithm competitive against current load, a small modification is used. As usual, we maintain levels $\lambda_i = 2^i\lambda_0$, using the basic algorithm on each level. As each new task t arrives, it is placed into the minimum level which will accept it. The algorithm is allowed to accept t into level i, however, if it can be done without increasing the level-i load on the destination machine beyond $6\lambda_i$.

To delete task t, remove it from its current level, and apply the rebalancing procedure described above. Then run the following additional rebalance procedure. Determine q, the maximum occupied level and j, the minimum numbered processor holding a task in level q. Select any task t assigned to j at level q. Attempt to insert t into level $q-1$. In this case, the algorithm rejects t if it cannot be placed without increasing some level $q-1$ load beyond 3λ. If t is rejected by level $q-1$, terminate. Otherwise, reassign t to level $q-1$ and repeat.

Theorem 4 *The witness-based algorithm is 24-competitive against current load. If the total weight of all tasks is W, then the total weight that is reassigned is $4W$.*

Proof: It is not hard to verify that Lemma 3 still holds for the modified algorithm. Hence the modified algorithm satisfies the conditions of a witness-based algorithm, with $c(n) = 6$. The competitive ratio follows from Lemma 1.

To show the bound on total weight reassigned, we perform an amortized analysis. Let $\Phi_{i,j} = 2(M_{i,j} - W_{i,j})$, where $M_{i,j}$ and $W_{i,j}$ are the values of M_j and W_j, respectively, in level i. Let $\Phi = \sum_{i=1}^{q} \sum_{j=1}^{n} \Phi_{i,j}$.

For purposes of analysis, we will label each server j as *marked* or *unmarked* in level i. We define a second potential function,

$$\Psi_{i,j} = \begin{cases} \min\{0, 3\lambda_i \text{cap}_j - W_{i,j}\} & \text{if } j \text{ is marked} \\ 0 & \text{if } j \text{ is not marked} \end{cases}$$

Let $\Psi = \sum_{i=1}^{q} \sum_{j=1}^{n} \Psi_{i,j}$.

Server j becomes marked in level i when the load on the server in level i exceeds $3\lambda_i$ and i is not the top level. Once server j is marked, it remains marked until i is becomes the top level, at which time it is unmarked.

Both Φ and Ψ are non-increasing when a new job u is inserted. The weight on a server can only increase. If a new top level q is started, a server i may become marked at level $q-1$ only if $W_{q-1,j} \geq 3\lambda_{q-1}\text{cap}_i$. Hence the amortized cost of the insertion is w_t.

Suppose u is deleted from server j in level i. Then $\Phi_{i,j}$ increases by $2w_u$ and $\Psi_{i,j}$ increases by at most w_u, for a total amortized cost of $3w_u$.

Suppose job u is reassigned from top level q to server j in level $q-1$. Since u can fit on j, we have that $w_u/\text{cap}_j \leq 3\lambda_{q-1}$. This implies that at the time u was first assigned to a level higher than $q-1$, the load on j in level $q-1$ must have been $> 3\lambda_{q-1}$, or else u could have been placed on it. Hence server j must be marked in level q. Hence $\Psi_{q-1,j}$ decreases by w_u when u is reassigned, for a total amortized cost of 0.

Finally, consider a rebalancing done within level j. One may verify that the decrease in Φ is sufficient to pay both for the reassignments of tasks and for the increase in Ψ. □

4 Eager Load Balancing for Restricted Assignment

In the restricted assignment problem, each task u has an associated subset of servers on which it can be executed. It must be assigned to one server in that subset. If no tasks ever depart and tasks cannot be preempted, the best possible competitive ratio for both randomized and deterministic algorithms is $\Omega(\log n)$; this ratio is achievable with a simple greedy strategy [5]. Azar *et al.*[4] showed that when tasks both arrive and depart, no non-preemptive algorithm can be better than $O(\sqrt{n})$ competitive against peak load. An non-preemptive algorithm that is $O(\sqrt{n})$ competitive against peak load is given in [3].

In this section we give an eager algorithm for the case of unit weights. Its competitive ratio is parameterized by a value $1 < q < \log n$, which determines both the ratio and amount of reassignments. It is competitive against current load.

Regard the problem as a game on a dynamic bipartite graph. On one side are the servers, V; on the other side are the tasks, U. An edge $\langle u, v \rangle$ indicates that u can be assigned to v. Edge $\langle u, v \rangle$ is *matching* if u is assigned to v.

We begin with a value, λ, that upper-bounds the current optimum load. Given $X \subseteq U$, let $Y(X) = \{v \in V \mid \exists \langle u, v\rangle \in E, u \in X\}$. In other words, Y is the set of columns v such that some element in X has an edge to v. A simple observation by the pigeonhole principle is that $\lambda \geq \max_{X \subseteq U} |X|/|Y(X)|$.

A *balancing path* is an even-length sequence of alternating matched and unmatched edges $\{v_1, u_1\}, \{u_1, v_2\}, \{v_2, u_2\}, \ldots, \{u_{m-1}, v_m\}$ with the property that $\text{load}(v_i) < \text{load}(v_1)$ for $1 \leq i \leq m-1$ and $\text{load}(v_m) < \text{load}(v_1) - \lambda$.

We may use a balancing path to reduce the maximum load on servers $v_1, v_2, \ldots, v_m$ by reassigning u_i to v_{i+1} for $1 \leq i \leq m-1$. We say the set of servers are *r-balanced* if there is no balancing path of length r or less.

Lemma 5 *If the set of n servers is $2q$-balanced, $1 \leq q \leq \ln n$, then the maximum on-line load is $c\lambda$, where c satisfies the equation*

$$\ln n/q \geq c(\ln c - 1) \tag{2}$$

Proof: Let h be the maximum j such that there is an on-line server of height at least $j\lambda$. For all $j \geq 0$, let Y_i^j be the set of columns $v \in V$ that are reachable by an alternating path of length at most $2i$, $1 \leq i \leq q$, starting from a column of height at least $j\lambda$. Thus Y_0^j is the set of columns of height at least $j\lambda$. Let $y_i^j = |Y_i^j|$.

No column in Y_i^j has height less than $(j-1)\lambda$, since otherwise there would be a balancing path of length $2i \leq 2q$. Let X_i be the set of items on columns of height at least λi. We have $|X_i| \geq i\lambda y_0^i$.

For any j and $i \leq q-1$, the set Y_{i+1}^j contains all servers adjacent to some item that is currently placed on a server in Y_i^j. There are at least $\lambda(j-1)y_i^j$ such items, and hence by the pigeonhole principle

$$y_i^j(j-1)\lambda/y_{i+1}^j \leq \lambda. \tag{3}$$

We also have (2) $y_0^j \geq y_q^{j+1}$. This follows from the observation that no server of load less than $j\lambda$ is reachable in $2q$ or fewer steps from a server of load $(j+1)\lambda$, or else there must be a $2q$-balancing path. Combining equations 3 and (2) we derive the following recurrence:

$$y_0^j \geq (j)^q y_0^{j+1} \qquad y_0^0 = n$$

Solving the recurrence, we find $n \geq y^h(h!)^q$. Since y^h is at least 1, this reduces to $n \geq (h!)^q$. Taking the natural logarithm of both sides and applying Stirling's approximation we derive $\ln n/q \geq h(\ln h - 1)$. □

Lemma 6 *For any q, all servers can be kept $2q$-balanced using $O(qh)$ reassignments per insertion or deletion.*

Proof: To insert an item u, place it on any server v to which it is adjacent. This increases the height of v and may create a balancing path starting at v. It cannot, however, create a rebalancing path originating at any other node. Rebalance along any balancing

path starting at v and terminating at v_1. At the conclusion of the rebalancing, v is returned to its initial height, server v_1 has increased in height by 1, and all other servers have unchanged height. Recursively apply the same procedure starting at v_1. By the definition of a rebalancing path, at the conclusion of rebalancing v_1 has height equal to the height of load$(v) - \lambda$. Hence the procedure can only be applied $O(h)$ times before reaching a column of height 1. Each call performs $O(q)$ reassignments.

The case of deletion is similar, except that deleting an item from v may create a rebalancing path terminating at v, and recursive calls occur at servers that are increasing in height by λ. □

Theorem 7 *The eager rebalancing witness-based algorithm is $O(h)$ competitive against current load and performs $O(qh)$ rebalances per item, where h is the value of c satisfying equation 2 for parameter q.*

Proof: As usual we let level i have a λ value of $2^i\lambda_0$. For level j, define $\Phi_i^j = (M_i^j - W_{j,i})qh$, where M_i^j is the maximum load on column j in level i, and W_i^j is the current load on column j. The proof is a straightforward case analysis. □

Corollary 8 *For any $c < 1$, there is an algorithm that performs $O(\log n / \log\log n)$ reassignments and keeps the on-line load within a factor of $O((\log n)^c / \log\log n)$ of the current load.*

Corollary 9 *There is an algorithm that performs $O(\log n)$ reassignments and keeps the on-line load within a factor of $O(1)$ of the current load.*

Corollary 8 follows from setting $q = (\log n)^{1-c}$, and Corollary 9 from setting $q = \log n$. This improves on the previous result of [2] in two ways: first it works for all values of the optimum load and second it is competitive against current rather than peak load.

5 Virtual Circuit Routing

In the virtual circuit routing problem one is given a communication network modeled by an undirected graph. Each edge $e \in E$ has an associated capacity cap_e. A task u is a triple (w_u, s_u, t_u), indicating the need for a weight w_u connection between nodes s_u and t_u. An on-line algorithm must choose a path in the graph between s_u and t_u to serve as the virtual connection. The current weight on each edge, W_e, is increased by w_u. The load on an edge is W_e/cap_e. We assume that the restart cost r_u is fixed and is independent of the number of edges in the connection. As usual we assume $r_u = w_u$; the restriction can be removed using the method of Section 6.

Azar *et al.* [2] give an algorithm that is $O(\log n)$ competitive against peak load and incurs restart cost $O(S \log n)$, where A is the sum of assignment sizes. In particular, the algorithm uses an estimate of the current load λ. We use this algorithm as the basis for an

witness algorithm that is competitive against current load. Unfortunately, we are unable to find a scheme that amortizes the startup cost associated with reassigning an item from a higher level to a lower level against the startup costs of departed jobs, as we are in all the other situations. Therefore we take a different approach.

Let $C = \sum_{e \in E} c(e)$ and let $W = \sum_{u \in U} w_u$, where U is the set of *active* tasks. Let h be the maximum load incurred by the on-line algorithm. We have

$$\begin{aligned} h &\geq W_e/\mathrm{cap}_e \quad \forall e \in E \\ h \sum_{e \in E} c(e) &\geq \sum_{e \in E} W_e \\ h &\geq W/C \end{aligned}$$

On the other hand, $h < W/\mathrm{cap}_{min}$, where $\mathrm{cap}_{min} = \min_{e \in E}\{c(e)\}$.

We modify the witness algorithm as follows. The lowest level has load $\lambda_0 = 2^{\lfloor \log W/C \rfloor}$. The highest level has load $\lambda_q = 2^{\lceil \log W/\mathrm{cap}_{min} \rceil}$. Level i has load $2^i \lambda_0$.

For an active task u, we bound the number of times u can be rerouted between changes in the value of λ_0. Since a task can be rerouted $O(\log n)$ times per level, and there are $O(\log C/\mathrm{cap}_{min})$ levels, u can be rerouted $O(\log n \log C/\mathrm{cap}_{min})$ times, at cost w_u per rerouting. If λ_0 decreases by 1, W has decreased from S to $S/2$. The additional level may add one to the number of levels the remaining jobs may drop down. This is charged to the jobs that have left, for an additional $O(\log n)$ per job. The total reassignment cost is $O(W \log C/\mathrm{cap}_{min} \log n)$ and the algorithm is $O(\log n)$ competitive against current load.

6 Unrelated Weights and Startup Costs

In the previous sections we gave algorithms that provide competitive bounds while guaranteeing that the total weight of tasks that are assigned and reassigned is bounded by a small function f in sum of the weights of the input tasks. Under the assumption that $w_u = r_u$, this implies that the total assignment cost is bounded by the same function f on the total startup cost. In this section we show how to extend these algorithm to handle the case that $w_u \neq r_u$. We will partition tasks based on the startup cost per unit weight of each task, r_u/w_u. Let a be the minimum value of this ratio over all input tasks and b the maximum value.

Choose a parameter $1 < \delta \leq (b/a)$. Partition the input tasks into $O(\log_\delta(b/a))$ classes such that task u is in class i if

$$a\delta^i \leq \frac{r_u}{w_u} < a\delta^{i+1}.$$

Within each class run the load-balancing algorithm appropriate to the problem being solved.

Lemma 10 *Let A be an algorithm that achieves a competitive ratio of c while the total weight of reassigned items is bounded by $g(W)$, where $W = \sum_{u \in U} w_u$ and $g(x) = \Omega(x)$. Then there exists an algorithm A_δ that achieves a competitive ratio of $c \log_\delta(b/a)$ and incurs a total reassignment cost $a\delta^{i+1} g(S/a\delta^i)$, where $S = \sum_{u \in U} r_u$.*

Proof: Within each of the classes, the algorithm is c competitive with the optimum load for tasks within that class. Let λ^* be the maximum over classes of the optimum load within that class. This is a lower bound on the true optimum load of all tasks. Since within each class no server has load greater than $c\lambda^*$, the maximum on-line load is $O(c \log_\delta(b/a)$. Within class i, the reassignment cost per unit weight is at most $a\delta^{i+1}$, hence the total cost of reassignments is $a\delta^{i+1}g(W)$. On the other hand, $W \geq S/(a\delta^i)$. □

Using Lemma 10 we have the following:

- Algorithms for identical and related machines that are $O(\log_\delta(b/a))$ competitive against current load and incur total assignment cost $O(\delta S)$.
- An algorithm for the restricted machines problem that is $O(\log_\delta(b/a))$ competitive against current load and incurs reassignment cost $O(\delta \log n)$ when all weights are unit, and an algorithm for the restricted machines problem that is $O(\frac{1}{\rho} \log n \log_\delta(b/a)$ competitive against current load and incurs a reassignment cost $O(\rho\delta S)$ for arbitrary weights, where ρ is any parameter between 0 and 1.
- An algorithm for virtual circuit routing that is $O(\log n \log_\delta b/a)$ competitive against peak load and incurs a reassignment cost $O(\delta S \log n \log(C/\text{cap}_{min})$.

All results follow by applying the lemma to algorithms presented herein, with the exception of the algorithm for general weights in the restricted subset case, which follows by applying the lemma to the algorithm in [12]

While the bounds of this section are perhaps not ideal, in that they depend on the value of a ratio between input costs, it is worth noting that they remain independent of the number of tasks.

References

[1] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 623–631, 1993.

[2] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitive routing of virtual circuit with unknown duration. In *Proc. ACM/SIAM Symp. on Discrete Algorithms*, 1994. To appear.

[3] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Proc. 1993 Workshop on Algorithms and Data Structures (WADS 93), Lecture Notes in Computer Science 709.* Springer-Verlag, Aug. 1993.

[4] Y. Azar, A. Karlin, and A. Broder. On-line load balancing. In *Proc. 33nd Symp. of Foundations of Computer Science*, pages 218–225, 1992.

[5] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 203–210, 1992.

[6] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24nd ACM Symp. on Theory of Computing*, 1992.

[7] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[8] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *Proc. 1992 ACM Int. Conf. Supercomputing*, July 1992.

[9] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[10] D. R. Karger, E. Torng, and S. J. Phillips. A better algorithm for an ancient scheduling problem. In *Proc. 1994 ACM/SIAM Symp. on Discrete Algorithms*, 1994. To appear.

[11] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 157–164, 1992.

[12] S. Phillips and J. Westbrook. On-line load balancing and network flow. In *Proc. 1993 Symp. on Theory of Computing*, Apr. 1993.

A Parallel 3D Parabolic Wave Equation Solver

Ding Lee[1], Diana C. Resasco, Martin H. Schultz[2]
Faisal Saied[3]

YALEU/DCS/RR-995
January 1994

The authors were supported in part by the Office of Naval Research (ONR) under contracts N00014-89-J-1671 and N00014-93-WX-24092, by the Naval Undersea Warfare Center (NUWC) independent research project A10003 and by grants from IBM and NSF ASC 92 09502 RIA.

[1]Naval Undersea Warfare Center
[2]Yale University, Dept. of Computer Science
[3]Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science

Abstract

Three dimensional (3D) models of sound propagation in the ocean can lead to very large scale computations. With the advent of parallel computing, we have the chance of doing these computations at an acceptably fast rate.

We describe our work towards porting a 3D parabolic equation solver to current parallel computers. The code is FOR3D, developed at NUWC, and is based on the Lee-Saad-Schultz model. This model takes the azimuthal coupling into account, and marches the solution out in range, with an AD scheme, which requires alternated sweeps, solving independent tridiagonal systems alternating in the depth and azimuth directions at every range step.

Our parallel implementation is in Linda, a language that allows parallel algorithms to be expressed in terms of a machine-independent model of parallel computing. Codes parallelized in Linda will run on any parallel computer on which Linda is implemented. We will focus on a cluster of workstations viewed as a distributed memory multiprocessor.

We report on the improvement in performance that can be obtained through a combination of algorithmic improvements and parallel computing. We touch upon some software engineering issues that have a strong bearing on the process of porting existing codes to parallel architectures.

We demonstrate that workstation clusters can be very effective for scientific codes, particularly when a fast interconnect or switch is used.

1 Introduction

In this paper, we describe our progress in implementing a portable parallel version of FOR3D, a code developed at the Naval Undersea Warfare Center (NUWC) for the prediction of sound propagation in the ocean. Section 2 describes the equations solved by the original FOR3D code. In Section 3 we discuss some general considerations on implementing a parallel version of the code. A brief description of Linda is given in Section 4 and the hardware used for our experiments is described in Section 5. In Section 7 we show performance data, we discuss possible future directions of our research in Section 8, and in Section 9 we present our conclusions.

2 Description of the Problem and Numerical Approach

FOR3D solves the following parabolic equation, which models one-way outgoing propagation:

$$u_r = ik_0(-1+\sqrt{1+X+Y})u \quad , \tag{1}$$

where u is a function of depth (z), azimuth (θ) and range (r), k_0 is a constant, the reference wavenumber, and the differential operators X and Y are given by

$$\begin{aligned} X &= n^2(r,\theta,z) - 1 + \frac{1}{k_0^2}\rho\frac{\partial}{\partial z}(\frac{1}{\rho}\frac{\partial}{\partial z}) \\ Y &= \frac{1}{k_0^2 r^2}\rho\frac{\partial}{\partial \theta}(\frac{1}{\rho}\frac{\partial}{\partial \theta}) \end{aligned} \tag{2}$$

where $n(r,\theta,z)$ is the index of refraction and ρ is the density. In FOR3D, ρ can be a step-wise linear function of z and θ [5].

A local solution to (1) can be written down symbolically as:

$$u(r+\Delta r,\theta,z) = e^{-ik_0\Delta r}e^{ik_0\Delta r\sqrt{1+X+Y}}u(r,\theta,z). \tag{3}$$

The numerical scheme designed by Lee, Saad and Schultz [4] makes use of the following rational approximation to the square root operator:

$$\sqrt{1+X+Y} \approx 1+\frac{1}{2}X-\frac{1}{8}X^2+\frac{1}{2}Y.$$

By assuming near commutativity of the operators X and Y, (3) becomes:

$$u(r+\Delta r,\theta,z) = e^{-ik_0\Delta r}e^{ik_0\Delta r(1+\frac{1}{2}X-\frac{1}{8}X^2)}e^{ik_0\Delta r\frac{1}{2}Y}u(r,\theta,z).$$

Then, the exponential operators are further approximated by rational functions, giving the following marching scheme:

$$u(r+\Delta r,\theta,z) = \frac{I+\alpha X}{I+\bar{\alpha}X}\ \frac{I+\beta Y}{I+\bar{\beta}Y}\ u(r,\theta,z) \quad , \tag{4}$$

where $\alpha = \frac{1}{4} + i\frac{1}{4}k_0\Delta r$ and $\beta = i\frac{1}{4}k_0\Delta r$.

A finite difference discretization of (4) leads to a problem of the form

$$ABU^+ = A^*B^*U \tag{5}$$

where U and U^+ are the computed solution vector at the present range r and at $r+\Delta r$, respectively, and A and B represent the discretizations of the operators $I+\alpha X$ and $I+\beta Y$ respectively, computed at range $r+\frac{1}{2}\Delta r$.

The computational domain at each range step is an N_z by N_θ grid, originally numbered column-wise, i.e., by depth first, then by sector. With such ordering, the matrix A is block-diagonal, with tridiagonal blocks in the diagonal. Similarly, if the gridpoints are numbered by sector first (i.e., row-wise), then the matrix B is block-diagonal, with tridiagonal blocks in the diagonal.

At each range step, computations are arranged in two AD half-steps, alternated with a transposition (reordering) of the partial solution vector:

- **z-half-step**: Compute right hand side and solve
$$A\tilde{U} = A^*B^*U \tag{6}$$
- **Transpose** intermediate solution $\tilde{U}$
- **θ-step**: Solve
$$B\dot{U}^+ = \tilde{U} \tag{7}$$
- **Transpose** new solution U^+ back to original ordering.

The FOR3D code is described in detail in [1].

3 Parallel Implementation: Some General Considerations

In this section we briefly outline some general features of our approach to parallelizing FOR3D, independent of any particular parallel hardware/software systems. In the subsequent sections we will describe how this approach is implemented on clusters of workstations using LINDA.

The basic structure of a range step in FOR3D consists of a set of independent tridiagonal solves in depth followed by another set of independent tridiagonal solves in the azimuthal direction. In addition, matrix vector products involving

tridiagonal matrices are required in each of these directions to form the right hand side.

Our approach to parallelizing these computations is to assign one or more of the (depth dependent) tridiagonal systems to each processor. Each processor forms the tridiagonal matrices it needs, and hence the matrices are not involved in the communications. The solves are done locally, using Gaussian elimination adapted to the structure of the matrices.

In the second phase of the range step, each processor solves one or more tridiagonal systems in the θ direction. The matrices for this phase are again formed locally, and are not involved in any inter-processor communication. However the right hand sides of the tridiagonal systems in this phase depend on the intermediate solution obtained from the first half-step, and data movement is required between these phases.

This data movement can be conceptually viewed as matrix transposition and can be implemented in several different ways. The simplest approach involves each processor sending data to every other processor and receiving data from every other. The global nature of the communication reflects the global data dependencies inherent in the FOR3D model. Any 3D model that takes full θ-coupling into account will have analogous global communication requirements. It is important to exploit any features of the parallel hardware and software to make this phase as efficient as possible to minimize the overhead cost of parallelization.

Because of the well-structured nature of the computational kernel in FOR3D, we achieve good load balance across processors.

4 Linda

Linda is a coordination language that complements traditional languages for computation. In our application, we used Fortran-Linda from Scientific Computing Associates. A few simple commands are added to Fortran. The resulting language is architecture-independent, which makes the code highly portable.

The Linda coordination model is based on a form of virtual shared memory, called "tuple space", that is designed specifically to accommodate inter-process coordination. Linda-style shared memory has been efficiently implemented in settings such as distributed-memory parallel machines and local area networks, whose architectures preclude communication via conventional shared memory. A tuple space stores tuples, which are ordered aggregates of data objects. Linda provides three basic access operations with built-in synchronization: the `out` operation generates a tuple and adds it to memory; the `in` operation looks for some "matching" tuple and removes it, blocking if necessary until one is available; the `rd` operation is like `in`, but copies rather than removes the matched tuple. Tuple space is an associative memory: `in` or `rd` statements specify a "matching template" or anti-tuple which may include either values or typed place-holders

or both. Linda also provides a process-creation mechanism integrated with the tuple space abstraction: the `eval` operation generates and places in tuple space an unevaluated tuple. When each field of the unevaluated tuple has been fully evaluated, the unevaluated tuple turns into an ordinary tuple which can be read or removed using the standard operations.

For a more detailed description of Linda and its applications, see [2].

5 Hardware

For our experiments, we used a network of IBM RS6000/560 workstations. With the Ethernet interconnect, only one processor pair can communicate at a time. A faster interconnect was recently added, IBM's AllNode (or V7), a switch which supports multiple, high-bandwidth, low-latency connections between processors.

6 Parallel FOR3D

In our parallel version of FOR3D, the computational domain is initially distributed so that each processor gets a sector (i.e. a number of azimuth values). In homogeneous networks, each processor works on a data set of approximately the same size, but the code accepts data sets of different sizes, to provide flexibility for good load balance when working on heterogeneous networks. For each range step, the computations in the depth direction are performed locally, then the partial solution is transposed across processors, so that each processor can locally compute the azimuth step, and then the solution is transposed back.

Instead of approximating 4 by $U^+ = B^{-1}A^{-1}A^*B^*U$ (solved by steps (6) and (7) in section 2), we use the ordering $U^+ = B^{-1}B^*A^{-1}A^*U$. This ordering of the operators facilitates the organization of the computation, because it permits us to group the computations in each alternated direction. The two transpose operations at every range step make sure that the data for each half-step is local to each processor, therefore these steps are performed with perfect parallel speed-up. All communication between processors is done at the transpose phase.

- z-**half-step**: Compute right hand side and solve (Each processor computes a portion of the solution.)

$$A\tilde{U} = A^* U \tag{8}$$

- **Transpose** intermediate solution $\tilde{U}$
- θ-**step**: Compute right hand side and solve (Each processor computes a portion of the solution.)

$$BU^+ = B^* \tilde{U} \tag{9}$$

- **Transpose** new solution U^+ back to original ordering.

Master-worker model

Linda allows the user to design a parallel application using the master-worker model, in which one of the processors acts as master, assigning data and tasks to the spawn processes. Although it is not essential, we chose to use this model for our parallel FOR3D. The master process is in charge of processing and distributing the environmental data, and for gathering output for visualizing the solution when requested by the user. It is possible to program the master processor to become a worker in cases when it is likely to stay idle for long periods of time.

Fig. 1 illustrates the sequence of computation and communication at a particular set of range steps, for a case of a master process and three workers. The arrows represent data being communicated among processors. We need to point out that this picture is an oversimplification of the communication pattern, and it does not necessarily represent the Linda model. Linda preprocesses the communication pattern in order to minimize overhead.

The master process gathers the solution vector for visualization when necessary (in most applications, the solution is not printed out at every range step), while the workers can continue marching the computations in range. When new environmental data needs to be inputed (again, this is not usually done at every range step), it is the master processor's job to process the new data and distribute it among the workers.

7 Parallel Performance

In the current implementation, the transpose step is the communication bottleneck. The cost for the transpose step increases linearly with the volume of data being transposed. Since we get perfect speed-up for the arithmetic computations, the amount of speed-up that can be achieved for a given problem size, as we increase the number of workers (processors), is basically bounded by the cost of the transpose. In Fig. 2 we show run time curves for two problem sizes,

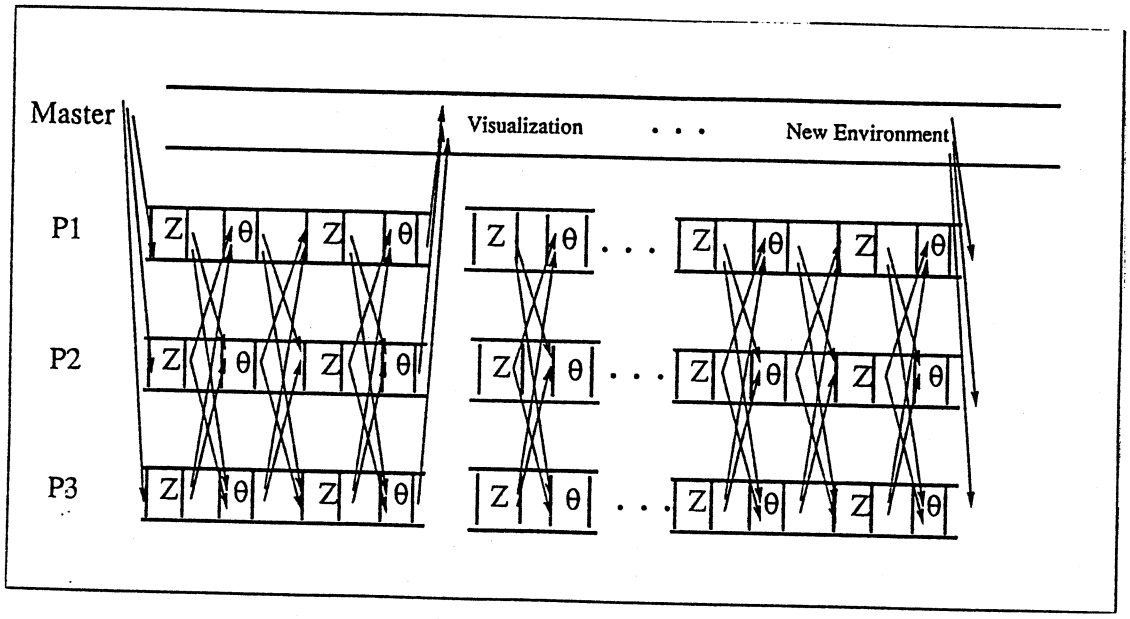

Figure 1: Communication and computation pattern

Table 1: Runtimes for $3D$ and $N \times 2D$ models
Run-time for one range step computation.
Grid size N_z=799 $\times$ N_θ=180

Number of Processors	3D	N×2D
1	4.20	3.20
2	2.80	1.61
3	1.97	1.12
4	1.62	0.85

for increasing number of processors. The solid lines indicate perfect speed-up (i.e. the one-processor time divided by number of processors). We can see that arithmetic times agree with the perfect speed-up curve, but as more workers are added, the time for the transpose dominates, making the parallel code less efficient. Fig. 3 shows speed-up curves for the same problems. These results are obviously heavily dependent on architectural features of the network or parallel machine. In Fig. 4 we can see the substantial improvement in efficiency achieved by the same processors when going from an Ethernet interconnect to the faster AllNode switch interconnect.

When θ coupling can be ignored, FOR3D can solve the equations in $N \times 2D$ mode, i.e. as a collection of independent two-dimensional problems. Basically, the θ-half-step is skipped. In this case, the code is not only faster, but obviously more parallelizable, since no transpose is needed. Table 1 shows runtimes per range step for a particular gridsize. We can see how the times for $N \times 2D$ follow perfect speed-up closely.

Finally, as an example of the improvement in performance that can be obtained through a combination of algorithmic and hardware improvements, code optimization, and parallel computing, we show in Table 2 the progression in computing time from a run of the original FOR3D code on a Sparc II workstation to our parallel version on five IBM RS6000/560 workstations.

8 Future Directions

In order to make FOR3D more efficient, we envision improvements to the sequential as well as the parallel versions.

One such improvement to the sequential code will be to replace

$$(I + \bar{\alpha}X)^{-1}\,(I + \alpha X) = \frac{\alpha}{\bar{\alpha}} I + (1 - \frac{\alpha}{\bar{\alpha}})(I + \bar{\alpha}X)^{-1}$$

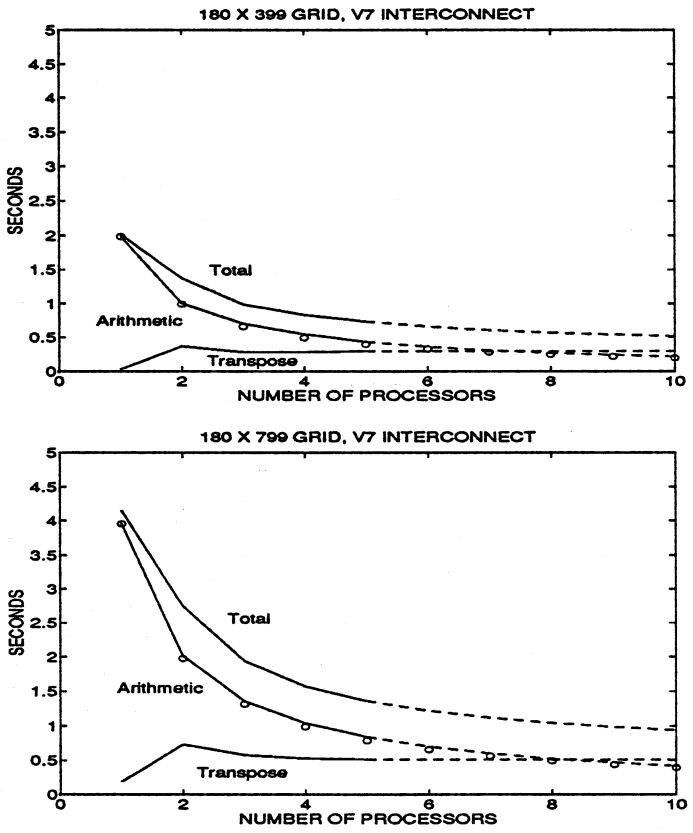

Figure 2: Parallel Performance of FOR3D
Run-time for two grid-sizes, as a function of the number of processors. Total time is the combination of transpose time plus arithmetic time. Dashed lines represent extrapolated values.

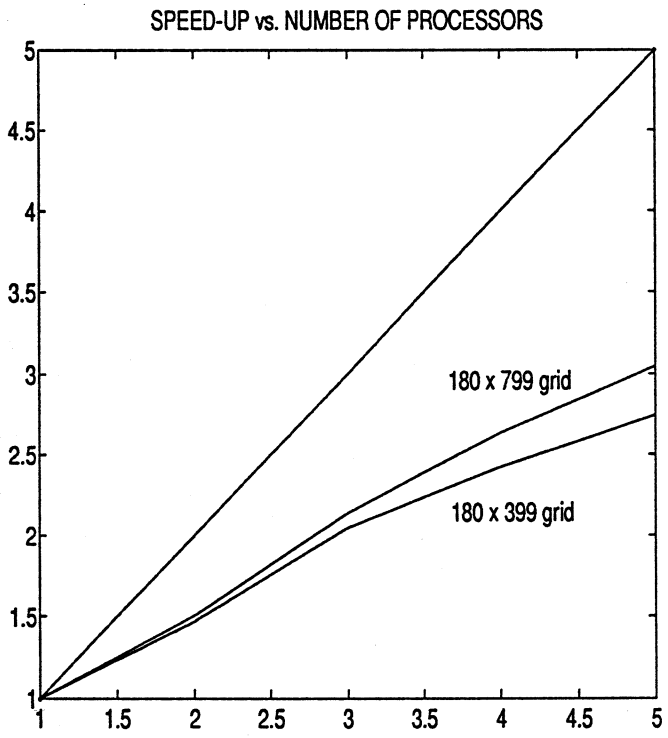

Figure 3: Parallel Performance of FOR3D: Speed-up

and

$$(I+\bar{\beta}Y)^{-1}\,(I+\beta Y) = \frac{\beta}{\bar{\beta}}I + (1-\frac{\beta}{\bar{\beta}})(I+\bar{\beta}Y)^{-1} = -I + 2(I+\bar{\beta}Y)^{-1}$$

in (4). This substitution eliminates the need to compute the right hand sides in the z and θ half-steps. We will report more on this at a later time.

In order to reduce the number of transpose steps needed, another idea is to use alternating sweep ordering, in which the (z and θ) half-steps are followed by (θ and then z) half-steps, thus eliminating the need for one of the two transposes per range step.

Since the cost for the transpose depends on the volume of data transfered, one variation of the algorithm is the use of substructuring (reduced system) techniques to solve the tridiagonal systems for the θ half-step. Instead of transposing the whole array, partial Gaussian elimination is applied to sections of

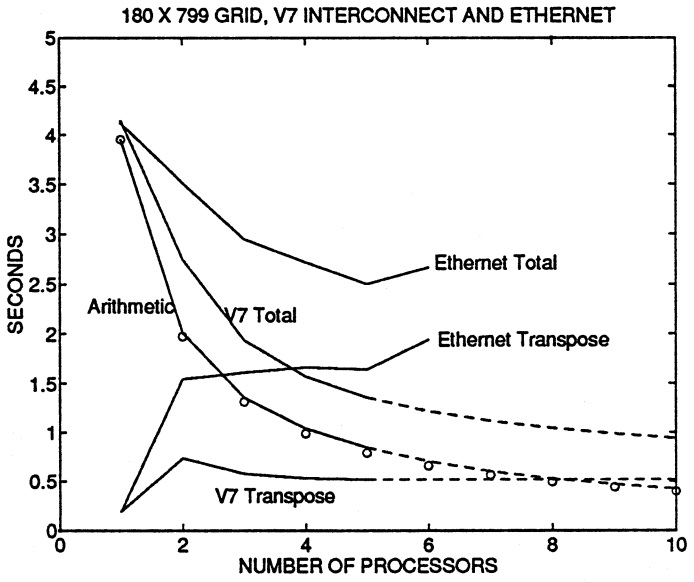

Figure 4: Comparison between Ethernet and AllNode interconnections

each system, and only a small reduced system is exchanged and solved through the network. This method doubles the operation count for the tridiagonal solves, but in most cases it will greately reduce the cost for the transpose, because the volume of data being transfered is reduced to the size of a separator set (about N_z vs. $\frac{N_z \times N_\theta}{p}$ per processor) [6, 3, 7, 8].

We finally point out that more parallelism is to be obtained in real applications by simultaneously solving for several frequencies. The computation for each frequency is completely independent.

9 Summary and Conclusion

We have described the implementation of a functioning portable parallel code for computational ocean acoustics. The tremendous advances in workstation technology have made the workstation cluster approach to parallel computing very attractive. Our experiments on a cluster of high-end workstations with a fast interconnect show that high performance can be achieved in a cost effective manner.

We used the coordination language Linda for our implementation. Linda is easy to use and has low overhead. Code written with Linda will run in any

Table 2: Speed improvement
Run-time for one range step computation.
Grid size $N_z = 799, N_\theta = 180$

Sparc II	
Original (sequential) code	49.0 sec
New (sequential) code	24.2 sec

IBM *RS*6000/560	
New (sequential) code	4.2 sec
Parallel code: 5 procs.	1.3 sec

parallel computer that supports Linda.

The experience gained by designing this application will be useful in parallelizing other similar acoustic codes.

10 Acknowledgements

This research was supported in part by the Office of Naval Research (ONR) under contracts N00014-89-J-1671 and N00014-93-WX-24092, by the Naval Undersea Warfare Center (NUWC) independent research project A10003 and by grants from IBM and NSF ASC 92 09502 RIA.

References

[1] G. Botseas, D. Lee, and D. King. FOR3D: A computer model for solving the LSS three-dimensional wide angle wave equation. Technical Report TR# 7943, Naval Underwater Systems Center, 1987.

[2] N. Carriero and D. Gelernter. *How to write Parallel Programs: A First Course.* MIT Press, Cambridge, MA, 1990.

[3] S. L. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. and Stat. Comput.*, 8:(354/392), 1987.

[4] D. Lee, Y. Saad, and M. H. Schultz. An efficient method for solving the three-dimensional wide angle wave equation. In *Computational Acoustics, Vol 1: Wave Propagation*, Amsterdam, 1988. North-Holland.

[5] D. Lee, Y. Saad, and M. H. Schultz. A three-dimensional wide angle wave equation with vertical density variations. In *Computational Acoustics:*

Ocean-Acoustic Models and Supercomputing, pages 143–154., Amsterdam, 1990. North-Holland.

[6] F. Saied. *Numerical techniques for the Solution of the Time-dependent Schrödinger Equation and their Parallel Implementation.* PhD thesis, Yale University, 1990. Available as Research Report YALEU/DCS/RR-811, Department of Computer Science, Yale University.

[7] A. Sameh and D. Kuck. On stable parallel linear system solvers. *J. ACM*, 25:(81/91), 1978.

[8] H. H. Wang. A parallel method for tridiagonal equations. *J. ACM Trans. Math. Softw.*, 7:(170/182), 1981.