A parallel Poisson solver using the fast multipole method on networks of workstations

June-Yub Lee*(jylee@math.ewha.ac.kr, jylee@cims.nyu.edu)
Dept. of Math, Ewha Womans University, Seoul 120-750, KOREA,
Karpjoo Jeong (jeong@mail.lns.cornell.edu, jeong@cs.nyu.edu)
Laboratory for Nuclear Studies, Cornell University, Ithaca, NY14853

December 19, 1997

Abstract

We present a parallel Poisson solver on distributed computing environments. In the solver, the parallel implementation of the Fast Multipole Method (FMM) is designed to minimize amount of data communication and the number of data transfers and synchronizations. The experimental results show linear speedup, good load balancing, and reasonable performance under failure and demonstrate the viability of loosely coupled heterogeneous workstations for large scale scientific computations.

Keyword: Volume integral method, Fast direct Poissone solver, High order of accuracy, Adaptive quad-tree, Domain decomposition.

1 Introduction

A variety of problems in computational science and engineering require the solution of the Poisson equation: $\Delta u = f$. Solving the Poisson equation is generally computation-intensive and therefore, parallel processing becomes inevitable as a problem grows in size. In the last few decades, a great deal of effort has been directed at parallelizing numerical methods for the Poisson equation [1, 2, 3, 4]. In this paper, we present a parallel Poisson solver which can be used to solve large scale real world problems. The reason we chose the Poisson equation as our target problem is that it is one of the most important partial differential equations (PDE) in scientific computation and also a good model problem to test and validate numerical schemes for more general elliptic PDEs.

Among currently available methods for solving the Poisson equation, we have chosen the direct, adaptive method [5] which solves the Poisson equation by directly evaluating the corresponding volume integral where the right-hand side f is defined on the leaf nodes of an adaptive quad-tree data structure. The method is based on a kind of domain decomposition or spectral element approach [6, 7], in which local solutions are patched together using the Fast Multipole Method [8]. It allows a substantial amount of parallelism among intermediate steps, but also contains complicated data dependencies among them. Therefore, parallelizing the method requires clever strategies for data distribution and scheduling; otherwise, communication and synchronization due to the data dependency can dominate runtime overhead and result in poor performance.

*This work was partially supported by Ewha womans university research grant, 1996 and by Korea Science and Engineering Foundation, KOSEF:970701-01013.

As our computing platform, we have targeted distributed computing environments which do not support physical shared memory or high performance communication among processors. A typical example is networks of workstations (NOW) with tens or even hundreds of powerful workstations. It is quite common in many workplaces and most of workstations are idle even at the busiest time of day. Therefore, there has been a great deal of interest in developing tools to harness the potential of these under-utilized workstations. A common approach is to make workstations join and retreat parallel computation when they become idle and busy, respectively. However, it is still considered a significant challenge to solve real world large scale problems by parallel processing on NOW. Part of the reason is due to the problems unique to this kind of distributed computing environment :

1. **Communication overhead:** Communication between computing elements is not only slower than numeric computation but also orders of magnitude slower than that between processing elements of a monolithic parallel computer.

2. **Heterogeneity and likelihood of failure:** NOW is in general heterogeneous and the load on the individual machines or the delays on the communication links can be arbitrary and unpredictable. Even worse, individual components may fail at any point (actual failure) and the utilization of idle CPU cycles requires owner activity on each workstation to be treated as failure (simulated failure).

Because of its slow communication speed, a set of distributed and heterogeneous computers would not be a suitable platform for fine grained parallelization which spreads a small portion of subroutine-level computation into many processing units. Though NOW has been proven to be an effective tool for coarse grained parallel computation, there has been little research on the use of NOW for medium grained parallel applications which allocate subroutine or small algorithm level jobs to different computing units. Therefore, the suitability of NOW for such applications is not yet clear. Hence, we are, by no means, claiming that NOW is a replacement for supercomputers, but simply that large scale scientific problems can be solved by by medium grained parallel processing on NOW. This requires 1. *A numerical algorithm* to minimize inter-process communication 2. *A parallelization tool* to utilize widely spread workstations in a simple and reliable manner which facilitates long running computation.

In this paper, we present a parallel algorithm for the Poisson equation and explain the implementation issues of a numerical algorithm on a distributed computing environment focusing on parallelization of the Fast Multipole Method [9, 8]. We address issues relevant to parallelization such as efficient data distribution, scheduling, and reliability which become more critical and challenging for distributed computing environments.

The rest of this paper is organized as follows. In section 2, we present a sequential algorithm for the Poisson problem based on the fast multipole method (FMM) where its hierarchical structure provides an opportunity for data and computation decomposition. Mathematical preliminaries, data structures, data dependency, and parallelism of the FMM are discussed and we then outline a parallel version of the Poisson solver. In section 3, we discuss data distribution, load balancing, and fault tolerance which are important for parallelization under NOW. Brief overviews of Linda and PLinda, a fault-tolerant extension of Linda, are given. We also discuss implementation issues related to optimizing communication costs between subcomputations of the Poisson solver under the Linda environment. In section 4, we experimentally demonstrate the viability of NOW for large scale, complex numerical computations by showing performance results of our prototype system. Section 5 concludes this paper.

2 A Parallel Poisson Solver using FMM

In this section, we briefly describe a direct, adaptive numerical method for the Poisson equation based on local polynomial solutions which are globally patched together using the FMM. We then discuss adaptive data structures and their dependencies along with the issues of communication and synchronization involved in parallelizing the sequential method.

2.1 Mathematical Preliminaries

To simplify the discussion, we restrict our attention to the Poisson equation

$$\Delta u = f \text{ in } R^2 \tag{1}$$

in the absence of physical boundaries, where the source distribution f has bounded support.

Mathematically a Poisson solver is a mapping from a source distribution $f(x)$, which might have complicated structures such as oscillations or internal layers, into the solution $u(x)$ of the Poisson equation in a given domain D. Thus, a numerical algorithm for this problem provides the value of $u(x)$ at each of a set of discretization points from a description of the source distribution f and a desired accuracy.

There are quite a few approaches to this problem. The most standard of which are finite difference, finite element and spectral methods, but fast direct solvers[10], relying on cyclic reduction or the fast Fourier transform (FFT), are limited to regular, tensor-product meshes. For more complex discretizations, using finite difference or finite element methods, it is common to rely on iterative solution procedures, including multigrid and additive domain decomposition [11, 12], which can easily be implemented on a parallel machine [13, 14]. Since the Laplacian is a (local) differential operator, a discretization point using such standard methods is coupled only to its nearest neighbors. This locality property has a two fold advantage in an iterative context. First, at each iteration, updating of a variable needs only information from a finite number of neighboring points, and the total cost of each sweep is linear in the number of grid points. Second, if a particular subregion has been allocated to a single processor, then only the interface points need inter-processor communication. Unfortunately, these standard methods are not completely robust when the source distribution has a complex structure, the grid is highly non-uniform, and high accuracy is required.

In order to overcome these difficulties, we solve the Poisson problem using a rather new approach which evaluates the solution u in the form of a volume integral

$$u(\mathbf{x}) = \frac{1}{2\pi} \int_{\mathbf{R}^2} \log |\mathbf{x} - \mathbf{y}| f(\mathbf{y})\, d\mathbf{y}. \tag{2}$$

There are many advantages to this approach and readers interested in a complete discussion of the sequential algorithm are referred to the paper by L. Greengard and J.-Y. Lee [5]. At first glance, this integral approach seems to be less attractive in terms of computational cost and spatial parallelism since direct evaluation of an integral operator with global dependency is quite expensive ($O(N^2)$ work, where N is the number of points in the domain, vs. $O(N)$ or $O(N \log N)$ for domain decomposition or multigrid) and the parallelism induced by local dependency is not obvious. Before describing the sequential method, we just remark that once the integral equation is discretized using our hierarchical data structure, we will recover data and computational locality. Only a small portion of interface data has to be transferred to neighbors. In fact, the present method shares many features with other numerical schemes for PDEs, such as domain decomposition methods, from the viewpoint of parallel computation.

We now briefly outline mathematical results on which our parallel implementation is based. Assume that the source distribution f is supported inside a square domain D embedded in a quad-tree structure with M leaf nodes D_i and f is smooth on the scale of each such small square D_i. The main result can be summarized in the following theorem :

Theorem 2.1 *Let the source distribution f be given as a K-th order Chebyshev polynomial f_i for each leaf node D_i for $i = 1, \ldots, M$ of the quad tree embedded on D. Then, for $\mathbf{x} \in D_i$, the solution to the Poisson equation (1) is given by*

$$u(\mathbf{x}) = u_i^s(\mathbf{x}) + \sum_{j=1}^{M} u_j^h(\mathbf{x}). \tag{3}$$

where $u_i^s(\mathbf{x})$ is a polynomial satisfying $\Delta u_i^s = f_i$ locally (inside D_i) and $u_j^h(\mathbf{x})$ is a harmonic function in D_i defined in terms of single and double layer potentials generated by the boundary values of $u_j^s(\mathbf{x})$ and $\frac{\partial}{\partial n} u_j^s(\mathbf{x})$ along the interfaces of subdomain D_j.

While $u_i^s(\mathbf{x})$ depends only on f_i, and is computed locally, the evaluation of the harmonic patches by direct summation over M boxes requires order $O(MN)$ work and communications (M source boxes to N target points), which is very expensive. The sequential algorithm described below uses the Fast Multipole Method (FMM) to achieve parallelism and to reduce the computation cost to order $O(N)$.

2.2 Adaptive Quad-tree Structure

Suppose a square box D contains the support of the right hand side f. Starting from $S_{0,0} = D$, a quad-tree structure is obtained by dividing a square subdomain $S_{l,k}$ into four equal size subdomains $S_{l+1,4k+d}$, for $d = 0, 1, 2, 3$. In order to achieve adaptivity, this process continues until the source term f is locally smooth enough on each of the leaf nodes $S_{l,k}$ aliased as D_i. We allow different level l of refinement under one condition which we call the refinement ratio 2 condition. That is two leaf nodes which share a boundary segment live at most one refinement level apart. This is merely for easy programming and could be relaxed easily.

Definition 2.1 *a) For each square $S_{l,k}$, the* neighbors $\mathcal{N}_{l,k}$ *consist of those squares at the same (or coarser, if none) refinement level with which it shares a boundary point. b) For each square $S_{l,k}$, the* interaction region *consists of the area covered by the neighbors of $S_{l,k}$'s parent, excluding the neighbors of $S_{l,k}$. The* interaction list $\mathcal{I}_{l,k}$ *consists of those squares in the interaction region which are at the same (or coarser, if none) refinement level.*

Using the notion we just defined, a domain D with respect to any node $S_{l,k}$ could be represented as a sum of itself, its neighbors, and the interaction list of ancestor starting from itself.

$$D \;=\; S_{l,k} \;\cup\; \mathcal{N}_{l,k} \;\cup_{p=0}^{l-1}\; \mathcal{I}_{l-p,floor(k/4^p)} \tag{4}$$

where $S_{l-p,floor(k/4^p)}$ is the p-th parent box of $S_{l,k}$. Therefore, if possible, a summation using the interaction list is cheaper than the direct summation over all leaf nodes, $O(Nlog_4 N)$ versus $O(N^2)$ for an optimally balanced quad-tree. However, data from far-away boxes (although few) is still needed.

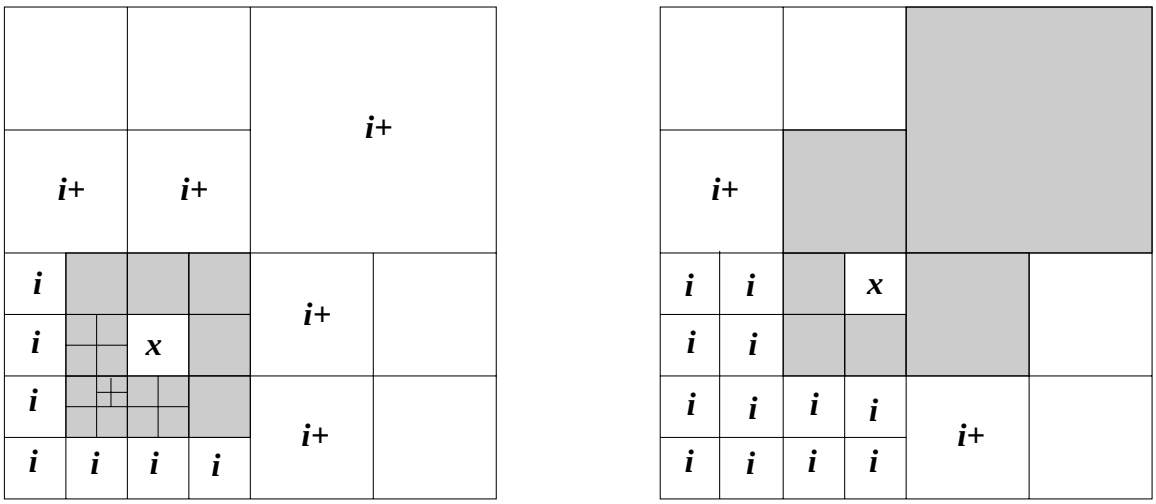

Figure 1: **Adaptive subdivision of a square domain D.** In the lefthand figure, all leaf nodes are visible and the neighbors of the square marked by an **x** are indicated by shading. The elements of the interaction list are indicated by an **i** or an **i**+, depending on whether they are at the same refinement level or at a coarser one. Note that some of the neighbors at the same refinement level are further subdivided, resulting in a somewhat complex local structure. In the righthand figure, the neighbors and interaction list of the square marked by an **x** are again indicated by shading or by the labels **i** and **i**+. We have omitted the refinements of some of the members of the interaction list on the right, since those refinements are of no consequence to the marked square under consideration.

2.3 Data Dependency and Parallelism of FMM

In order to further reduce the computational cost to $O(N)$ and simultaneously reduce the data dependency from far away, we need to take a look at the mathematical structure of the summation of harmonic patches $\sum_{j=1}^{M} u_j^h(\mathbf{x})$. Let us define a multipole expansion $\Phi_{l,k}$ and a local expansion $\Psi_{l,k}$ for each of the $S_{l,k}$,

$$\Phi_{l,k}(\mathbf{x}) = \sum_{D_j \subset S_{l,k}} u_j^h(\mathbf{x}) \text{ for } \mathbf{x} \in \mathcal{N}_{l,k}^c, \tag{5}$$

$$\Psi_{l,k}(\mathbf{x}) = \sum_{D_j \subset \mathcal{N}_{l,k}^c} u_j^h(\mathbf{x}) \text{ for } \mathbf{x} \in S_{l,k}. \tag{6}$$

where $\mathcal{N}_{l,k}^c$ denotes the outer region of $\mathcal{N}_{l,k}$ in R^2. In the FMM based sequential method[5], there are three kinds of data dependency among $\Phi_{l,k}$, $\Psi_{l,k}$, and $\sum_{j=1}^{M} u_j^h$:

1. The multipole expansion $\Phi_{l,k}$ of the node $S_{l,k}$ depends on the multipole expansions of the four children.
$$\Phi_{l,k} = \sum_{d=0}^{3} \Phi_{l+1,4k+d} \tag{7}$$

2. The local expansion $\Psi_{l,k}$ of the node $S_{l,k}$ depends on the local expansion of the parent and harmonic patches over the interaction list $\sum_{D_j \subset \mathcal{I}_{l,k}} u_j^h(\mathbf{x})$ which can be written in the form of the multipole expansions of all elements of the interaction list
$$\Psi_{l,k} = \Psi_{l-1,floor(k/4)} + \sum_{S_{l,k} \subset \mathcal{I}_{l,k}} \Phi_{l,k} \tag{8}$$

3. The harmonic patches $\sum_{j=1}^{M} u_i^h(\mathbf{x})$ for $\mathbf{x} \in D_i = S_{l,k}$ depends on u_i^h, $\sum_{D_j \subset \mathcal{N}_{l,k}} u_j^h$, and $\Psi_{l,k}$

$$\sum_{j=1}^{M} u_j^h(\mathbf{x}) = u_i^h(\mathbf{x}) + \sum_{D_j \subset \mathcal{N}_{l,k}} u_j^h(\mathbf{x}) + \Psi_{l,k}(\mathbf{x}) \text{ for } \mathbf{x} \in D_i = S_{l,k} \tag{9}$$

If an intermediate step needs the results of another step, then the former step is called *data-dependent* on the latter and can not start before the latter step finishes. Thus, data dependency prohibits parallel computation and requires communication and synchronization. In our application, dependency 1 forces us to compute multipole expansions of descendent nodes before those of ancestors; this phase of computation is called the *upward pass*. Dependency 2 requires local expansions of ancestor nodes to be computed before those of descendents; this phase of computation is called the *downward pass*. The downward pass also requires multipole expansions in the interaction list, thus, the upward pass must precede the downward pass; that is, there is no parallelism between the two passes. Furthermore, dependency 3 for leaf nodes $D_i = S_{l,k}$ requires local expansions of its parents, u_j^h for all D_j in the neighbor list $\mathcal{N}_{l,k}$ and u_i^h for D_i.

In spite of these three dependencies, the method still allows a substantial amount of parallelism. The major parallelism is that the multipole expansions in different subtrees can be computed concurrently. Likewise, we can use parallel processing for local expansions. This approach is particularly promising on networks of workstations because it allows coarse grained parallel computation.

2.4 The Numerical Method

We briefly summarize the implementation of the theorem which consists of four steps:

1. **First local solve:** Given any 2-dimensional Chebyshev polynomial $f_j(\mathbf{x})$, find a polynomial $u_j^s(\mathbf{x})$ such that $\Delta u_j^s(\mathbf{x}) = f_j(\mathbf{x})$ and then compute a multipole expansion $\Phi_j(\mathbf{x})$ at the center $\mathbf{y}_j$ representing the harmonic patch $u_j^h(\mathbf{x})$ for each of leaf node boxes D_j, $j = 1, \cdots, M$.

2. **FMM upward pass:** Once the multipole expansion Φ_j for each leaf node is obtained, multipole expansions $\Phi_{l,k}$ for internal nodes $S_{l,k}$ can be computed by collecting the information from their four children $S_{l+1,4k+d}$, $d = 0, 1, 2, 3$. The multipole for $S_{l,k}$ represents $\sum u_j^h(\mathbf{x})$ of all D_j inside of $S_{l,k}$ for $\mathbf{x} \notin \mathcal{N}_{l,k}$.

3. **FMM downward pass:** The local expansion $\Psi_{0,0}(\mathbf{x})$ for the root node is zero by definition since $\mathcal{N}_{0,0}^c$ is empty. The local expansion $\Psi_{l,k}(\mathbf{x})$ of a descendent $S_{l,k}$ at $\mathbf{x} \in S_{l,k}$ is a combination of its parent's $\Psi_{l-1,floor(k/4)}(\mathbf{x})$ and $\Phi_j(\mathbf{x})$ for all $D_j \in \mathcal{I}_{l,k}$ since $\mathcal{N}_{l,k}^c = \mathcal{N}_{l-1,floor(k/4)}^c \cup \mathcal{I}_{l,k}$. The hierarchical quad-tree data structure allows a recursive procedure of the summation from top to bottom.

4. **Final local solve:** Once the $\Psi_i(\mathbf{x})$ for all leaf nodes D_i are computed, the harmonic patches $\sum_{j=1}^{M} u_j^h(\mathbf{x})$ of the leaf node D_i is the sum of $\Psi_i(\mathbf{x})$, $u_i^h(\mathbf{x})$, and $\sum u_j^h(\mathbf{x})$ for $D_j \in \mathcal{N}_i$. To save computational time, instead of evaluating u_i^s and the harmonic patches at all of the desired points $\mathbf{x}$, we just evaluate them at the boundary points of D_i containing $\mathbf{x}$ and then solve the local Poisson equation again, but with the correct boundary data.

We end this section by estimating the CPU time required by the sequential method. Letting M be the number of leaf nodes and K be the desired order of accuracy, we construct a (scaled) $K \times K$ Chebyshev mesh on each leaf node D_i for $i = 1, \ldots, M$. The total number of discretization points is given by $N = MK^2$. The computational cost of the Poisson solver, to get the solution $u(x)$ at the

$N = MK^2$ grid points on M leaf nodes with $K \times K$ grid points each, is of order $N\left(4K + \frac{27p^2}{K^2} + K^2\right)$ where p is the number of terms in the multipole expansion (around 20 for single and 40 for double precision computation). The first term is the cost for the first local solvers, the second term for the multipole steps, and the third term for the final local solvers. For low order computation, the multipole steps dominate the computational cost and for high order computation, the final solver does. Break even points are at $K = 10$ for single precision computation with $p = 20$ and $K = 14$ for double precision with $p = 40$.

3 Parallel Implementation Issues

There are basically two approaches to parallel processing: shared memory and message passing[15]. The shared memory model is considered to be easier to use because of its intuitive approach, but it is less efficient than the message passing model on distributed computing environments such as NOW with no physical shared memory, because of an additional software layer required to simulate shared memory by network communication. We have chosen the shared memory model because the *ease of programming* issue is very important for developing software tools such as a Poisson solver since development of such a tool usually goes through a number of upgrade or modification phases.

Although our parallel solver is based on the shared memory model, its design focus is on reducing the use of shared memory as much as possible, in order to address the runtime performance issue. The runtime overhead due to shared memory is proportional to how much data shared memory maintains and how frequently shared memory is accessed and needs to be synchronized.

3.1 Data Distribution and Task Allocation

The sequential method maintains all of the data $\Phi_{l,k}$, $\Psi_{l,k}$, (and in addition u_i^s for each leaf node) in a simple and uniform quad-tree structure. The data sizes are $p * 16$, $p * 16$ and $8K * 8$ bytes, respectively. In our test example with 1M discretization points using $K = 8$, a single precision multipole method with $p = 20$ takes 32MB memory for the whole tree or 1.5KB for each of 16K boxes and spends 250 seconds for the full solve, averaging 16 $m$seconds per box. To handle real world problems that require large quad-trees and frequent access, the efficient management of the tree is very important for runtime performance. In this section, we discuss how to distribute and share the quad-tree data among processes participating in parallel computation.

We treat a multipole or local expansion of each node as a unit of data and a unit of computation, thus, the quad-tree distribution and task allocation are directly related. We considered two approaches to how to distribute the quad-tree in the parallel method:

- *Completely Shared quad-tree, Completely Dynamic task allocation* (CSCD). The whole tree is maintained as shared data. During execution, a process repeatedly grabs a node (or a tiny subtree) which is not computed yet, reads intermediate data required for the node or subtree, executes the computation, and then stores results back into shared memory.

- *Almost No Shared quad tree, almost Completely Static task allocation* (NSCS). At the beginning, the quad-tree is partitioned into some number of processes. During execution, the process performs computation only for the nodes in its partition. However, this strategy also requires processes to exchange boundary data with neighbor processes to make progress. For example, the multipole downward sweep needs data from the interaction list which might be allocated to other processes. They maintain in shared memory only the minimum interface data attached to nodes on boundaries between adjacent partitions.

We compare these two approaches with respect to ease of programming, load balancing, and shared memory access or network communication. First, CSCD allows the design of parallel code to be relatively simple because processes can access the entire tree in an uniform way whereas NSCS requires each process to be aware of the location of all the required data. CSCD also allows better load balancing because the faster the workstation a process runs on, the more work it performs. The strategy can handle the problem of dynamically changing load. By contrast, addressing this problem in NSCS is usually difficult because extra code is required. However, CSCD requires much more frequent access to shared memory because it accesses shared memory to read necessary data and to store a result for each node. By contrast, NSCS accesses shared memory to read and store only interface data.

Reducing data transfer among processes is crucial for the runtime performance of our Poisson solver, where over a couple of kilobytes of data are required for each box whose computation takes only a few mili-seconds. We have chosen the NSCS approach for runtime performance.

Let us now discuss how a process shares interface data. In figure 2, the outermost box represents a subtree whose adjacent subtrees have been allocated to other processes and we show only uniform refinements of the subtree for the sake of simplicity. The interface boxes illustrated by shading have to transfer their multipole expansions to the boxes allocated to other processes.

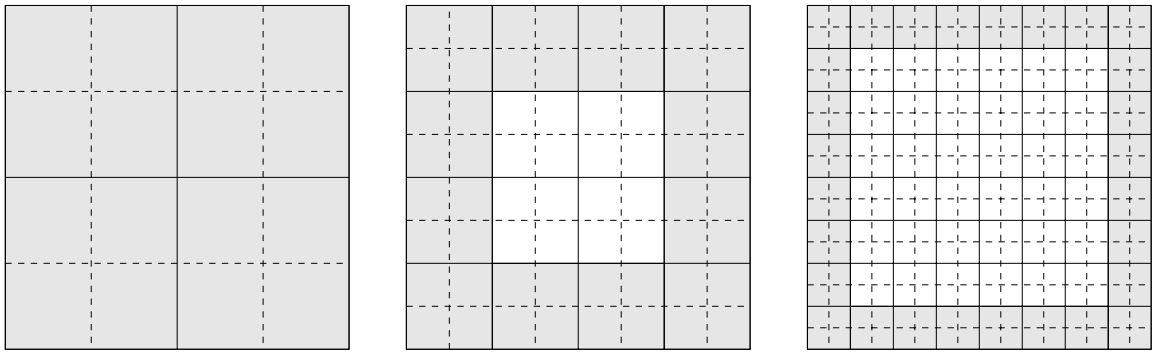

Figure 2: **Interface boxes of a subtree with various levels of uniform refinements.** In the leftmost figure, a subtree is uniformly refined up to level 2. All 16 leaves have parents which touch the subtree boundary. In the middle figure, the inner 4 parent boxes do not touch the subtree boundary, so that their 16 children nodes out of 64 leaf nodes on level 3 are not interface boxes. In the rightmost figure, only 112 boxes out of 256 leaf boxes are interface boxes.

In short, interface data are generated by boxes whose parents share a boundary point with a subtree allocated to different processes. This kind of simple data communication structure allows us to pack and ship all the interface data as a single message to its neighboring processes just after the upward pass. We create eight tuples, named as <INTERFACE>, on a share memory space. These tuples contains interface data for north, east, west, south and four diagonal direction neighbors, respectively. As shown in figure 2, a subtree with 2 level of uniform refinements has 16 leaf nodes, 4 parents, and 1 grandparent, in total 21 nodes and all of them are interface boxes. However, starting from refinement level 3, some nodes become excluded. As further refinements are undertaken, the number of interface boxes increases only as the square root of the total number of boxes. Table 3.1 summarizes the ratio of interface boxes to total boxes as a function of refinement level. As we can see, a subtree with many levels of refinement has a lower interface box ratio, therefore it is more efficient in terms of communication.

For runtime performance, load balancing is also crucial. Consider the case where a single process

Refinement level	1	2	3	4	5	6	7	8	9	10
Leaf nodes	4	16	64	256	1024	4K	16K	64K	256K	1M
Total nodes	5	21	85	341	1365	5461	21845	87381	349525	1398101
Interface nodes	5	21	69	181	421	917	1925	3957	8737	16213
Interface ratio(%)	100	100	81.2	53.1	30.8	16.8	8.8	4.5	2.3	1.2

Table 1: Number of leaf, total, and interface nodes, and ratios of interface to total nodes for a uniformly refined subtree.

holds a big portion of the tree. In that case, there is not much interface data, but the resulting performance will be poor. In order to address the problem, we modify the NSCS strategy as follows. The quad-tree is partitioned into subtrees that outnumber processes. Processes repeatedly grab a subtree and perform first local solves on the subtree until all the subtrees are taken and then each process computes only on the trees grabbed in the first stage. This scheme works fairly well even with machines with different computing capabilities (*static difference*). In section 4, we experimented with overall performance by varying the number of tasks for a given number of processes. A more challenging problem is how to deal with the situations where workload changes after the partitioning is finished (*dynamic difference*), or a machine stops after certain computations and communications with other processing units. This situation is discussed in subsection 3.3 and in our last example in section 4.

3.2 Parallel Algorithm

In this subsection, we present the algorithm underlying our parallel Poisson solver, based on a parallel programming technique called the *master/worker model*. In this model, there is one master process and multiple identical worker processes. The master creates tasks and workers repeatedly acquire tasks from the master and execute them. The programming model is effective for load balancing on distributed computing environments where workstations have different computing power and their workload can change dynamically.

The algorithm of the parallel Poisson solver is designed as follows:

1. *Quad-tree and task generation* **[Master]** • The master builds a skeleton quad-tree which does not contain multipole and local expansions. • Then, it partitions the tree into subtrees and stores each subtree as a single tuple named <SUBTREE>.

2. *Local solve and FMM upward pass* **[Workers]** • Each worker destructively retrieves a tuple <SUBTREE> for the first local solve. • Once all tuples are exhausted, a worker starts the upward pass for each subtree allocated. For each subtree, it generates the interface data of nodes on boundaries <INTERFACE> and the multipole coefficients of the top box <MULTIPOLE>. To minimize both the frequency and the amount of communication, each set of interface data to eight neighbors is packed into a single tuple.

3. *Final upward pass and first downward pass* **[Master]** • The master collects all multipoles <MULTIPOLE> of the top nodes of the subtrees from workers and finalizes the FMM upward pass for the upper quad-tree. • Then it starts the FMM downward pass to generate local expansions <TAYLOR> of each of the top boxes of the subtrees for the workers.

4. *FMM downward pass and second local solve* **[Workers]** • Workers perform the downward pass computation using <TAYLOR> from the master and <INTERFACE> on subtrees allocated to

other workers. • For each leaf node, they perform the final local solve using the correct boundary conditions and report the <RESULTS> for each allocated subtree to the master.

5. *Termination* **[Master]** • The program terminates when the master gets the <RESULTS> from all subtrees.

The parallel method executes these steps sequentially, but uses parallel processing for the second and fourth steps, which involve most of computation.

3.3 A Fault Tolerant Computing System using PLinda

Fault tolerance is very important for parallel processing on NOW, not just because a system with many computing units has higher chances of computer or network failure (hard failure) but also because a computer may become very slow during a computation (soft failure) or a job may be stopped when the owner of a private computer touches a mouse or a keyboard (policy failure). However, designing a fault tolerant scheme for arbitrary programs is challenging because a failure may happen at any point and even a very minor fault can stop the execution of computation. Even worse, the whole process may continue and result in incorrect outputs without any warning, since centralized management or monitoring of parallel computation is very difficult.

In this subsection, we explain how to extend the parallel algorithm in subsection 3.2. Since it is not easy to support fault tolerance through application code alone, we use a fault-tolerant computing system called "Persistent Linda" or PLinda[16, 17, 18] developed at New York University. PLinda is based on the shared memory Linda model[19, 15] and guarantees that shared memory called tuple space survives failure[1]. In the PLinda model, each process executes a series of steps (each step called an atomic action or transaction) that are guaranteed to run in the "all or nothing" manner. The process saves critical data which needs to survive failure to shared memory at the end of each step. On failure, the PLinda runtime system automatically restarts the failed process, called the "backup" process. The backup process retrieves critical data from shared memory and resumes execution from the completion point of the last step.

Since the parallel algorithm maintains all the computation results in a quad-tree, protecting the quad-tree from failure is most crucial for fault tolerance of the algorithm. In the algorithm, the quad-tree is distributed over processes' local states: data of the top portion on the master and data of each of the subtrees on the workers. Although local data allocated to a failed process would be lost, the PLinda system guarantees that data in fault-tolerant tuple space would be safely recovered back at the beginning of the failed step.

There are two extreme approaches to failure recovery. One is saving the image of all local states at the end of each transaction, which allows a failed process to recover quickly but requires saving data to the memory space of other machines or local stable storage such as disks. The other one is saving only the minimal information needed for recovery, where the backup process restores the local status of the failed process by restarting the computation from scratch. In many scientific computations such as our Poisson solver, a small portion of initial data generates huge amounts of intermediate data, which is explicitly reproducible. We have taken the latter approach because this approach incurs less overhead during normal execution, and accept the relatively high cost of failure which is in fact rare.

In the algorithm described in subsection 3.2, a master performs steps 1, 3, 5 and workers perform steps 2, 4. The master generates a quad-tree and task tuples <SUBTREE> in step 1

[1] In fact, tuple space itself may loses data on failure, but PLinda maintains the global consistency among processes and tuple space regardless of failure.

and failure in step 1 results in automatic restarting at the very beginning. Workers start step 2 by grabbing <SUBTREE> and solving local equations after successful completion of step 1 and generate <INTERFACE> and <MULTIPOLE> after the FMM upward pass. The Master starts the final upward pass and the first downward pass once all multipole moments <MULTIPOLE> of subtree top boxes are received and generates <TAYLOR> for workers. If the master fails in step 3, it can safely resume its computation at the beginning of step 3 instead of going back to step 1, assuming it is saved at the end of step 1. Workers need not just <TAYLOR> from the master and <INTERFACE> from the other workers to successfully pass step 2 through tuple space but also local data which have been generated in step 2. A failed worker recomputes local solutions to regenerate the local data and restarts step 4, but other successful workers can continue their jobs to make <RESULTS> since <INTERFACE> even from the failed process is safely stored in tuple space. In step 5, the master just collects <RESULTS> so any previous local status is not needed.

4 Experimental Results

The algorithm described in section 3.2 has been implemented in double precision floating point arithmetic using C/C++ with aid of mathematical subroutines written in Fortran and has been ported for machines running on various operating systems such as SunOS 4, Solaris, IRIX, and HPUX. Most parallel implementation codes are identical with that of the sequential version except for two main control programs: one for the master and one for the workers, thus provide the identical results on adaptive grids. We demonstrate only the case of a uniform grid on Sun workstations in our examples below for the sake of clarity.

EXAMPLE 1 **(Speedup)** We first consider a Poisson equation on a uniform quad-tree to test speedup using 8 identical Sun SPARC 5 workstations with 85MHz CPU clock speed, 32 MB memory, 128 MB swap space, and one shared 1GB NFS hard disk attached to our PLinda server. We did not run the master on a separate machine, therefore, one machine runs both the master and a worker. However, the computations of the master and workers do not overlap so that does not cause a problem except for swapping.

Box	K	Points	p	Sequential	W1	W2	W4	W6	W8
16K	8	1M	22	336.4	367.0	152.7	71.4	62.9	41.6
16K	8	1M	45	571.7	601.9	275.4	118.2	91.9	66.6
16K	16	4M	22	1223.0	1248.8	591.9	282.7	210.8	151.3
16K	16	4M	45	1453.7	1461.1	719.6	332.3	247.5	191.1
64K	8	4M	22	N. A.	N. A.	861.6	432.2	292.2	212.5
64K	8	4M	45	N. A.	N. A.	1258.5	629.6	464.8	335.8
64K	16	16M	22	N. A.	N. A.	2574.4	1280.6	957.4	654.8

Table 2: Wall clock run time in seconds (Example 1). The data in the column marked W1 correspond to 1 worker, the data in the column marked W2 correspond to 2 workers, etc.

We tested various combinations of parameters: number of Boxes (16384 boxes at 7 levels of uniform refinment, 65536 boxes at 8 levels), discretization order for local solves ($K = 8$ for 8 by 8 points per box, $K = 16$ for 16 by 16), total number of discretization Points (Box * K^2), and FMM accuracy ($p = 22$ for single precision, $p = 45$ for double precision) and partitioned the quad-tree into 16 equal subtrees to various number of workers.

To check the speedup, we normalize computational time of the parallel implementation based

on that of a sequential one with the same input parameters. The plotted values in the left graph in Figure 3 are the ratio between sequential time and parallel time for the four examples with 16K boxes. However, we cannot run the examples with 65536 boxes for 4M or 16M discretization points using our sequential solver (or our parallel solver with 1 worker) because of memory limitations, so we use the time for 2 workers for normalization.

Figure 3 shows almost linear speedup, with occasional superlinear performance due to a reduction in memory swapping.

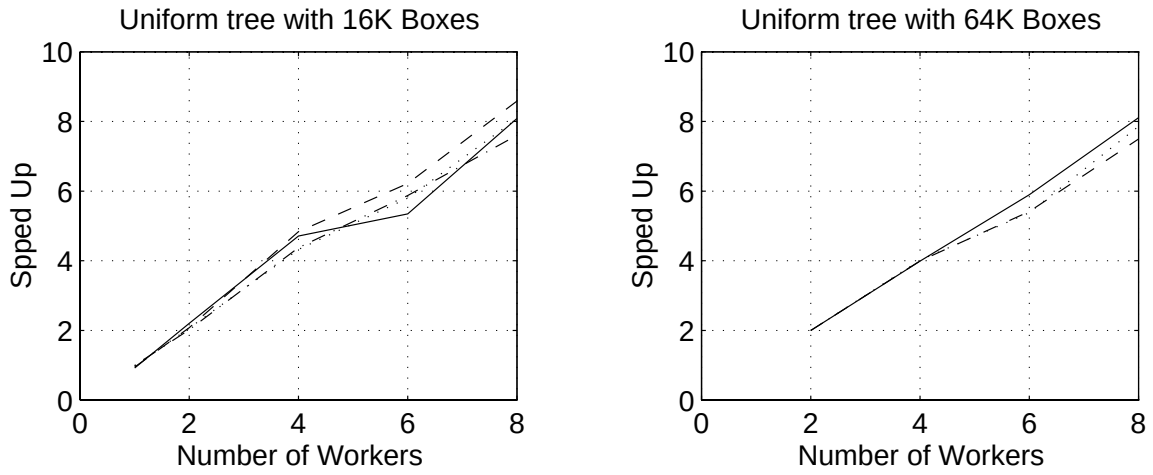

Figure 3: **Speedup using multiple workers (Example 1)**
In the lefthand figure, we plot four results with 16K boxes and in the righthand figure, three results with 64K boxes. The results for $K = 8/p = 22, K = 8/p = 45, K = 16/p = 22$, and $K = 16/p = 45$ are plotted using solid, dash, dotted, and dashdot lines, respectively.

Example 2 **(Scalability)** We perform the identical experiments as in the first four cases in Example 1 with 16384 boxes but examine the memory requirements instead of run time. The sequential code using about 34.7 to 64.0 MB memory works fairly well on a machine with 32MB memory and 128MB swap space but fails to solve bigger problems with 65536 boxes. By contrast, the parallel code distributes memory requirements over worker processes (in other words, the more machines the less memory requirement for each worker) and the memory requirements for the master increase slowly with the number of boxes.

Box	K	Points	p	Sequential	Master	W1	W2	W4	W6	W8
16K	8	1M	22	34.7	5.6	37.1	21.1	12.8	8.6	7.8
16K	8	1M	45	50.5	5.6	52.7	29.1	17.1	14.0	10.7
16K	16	4M	22	48.3	8.1	50.6	28.7	17.3	11.9	11.4
16K	16	4M	45	64.0	8.1	66.3	37.0	21.6	14.4	13.6

Table 3: Memory used by sequential solver, master, and worker in Mbytes (Example 2)

Our major concern is how much memory is required for the master and for each worker and how much additional memory is, overall, required for the parallel code. As it can be seen in Table 3, the parallel code requires only a modest amount of additional memory which is well distributed over workers. For example, the overall memory requirement with a master and 8 workers for the parallel code is only twice as big as that for the sequential code. Also, the memory requirements

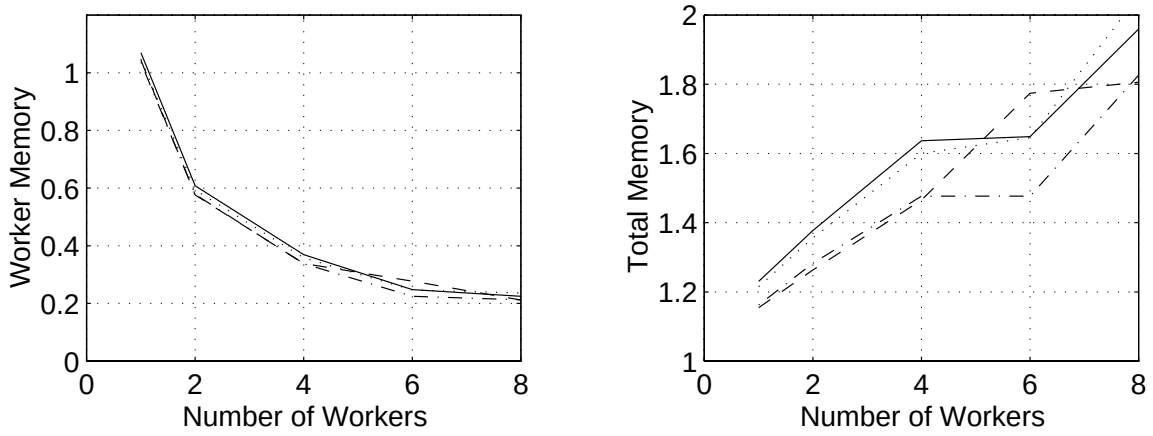

Figure 4: **Memory usage by each worker and by all (Example 2)**
In the leftmost figure, we normalize memory requirements of workers by memory used by the sequential solver. The one worker case is more expensive than that for the sequential one but less memory is used for more workers. In the rightmost figure, total memory used by a master and the workers used is plotted which is also relative ratio to the sequential solver. Four different line styles used here represent the four examples as in the Figure 3.

for the master increase slowly with the number of boxes. Therefore, we conclude that the parallel code is scalable with respect to memory requirements.

EXAMPLE 3 **(Load balancing and data partitioning)** In this example, we fix the problem size to 7 level uniform refinement with $K = 16$ and $p = 22$ and change the subtree size and the way we distribute subtrees over the workstations. We used either 16 or 64 subtrees, with workers getting subtrees at random locations versus workers trying to get subtrees in a clustered fashion. Table 4 summarizes the run time on identical Sun SPARC 5 workstations.

Subtree	Clustering	W1	W2	W4	W6	W10
16	YES	1248.8	591.9	282.7	210.8	154.8
64	YES	1250.7	606.9	280.3	197.5	140.7
16	NO	N. A.	614.3	289.1	217.2	163.5
64	NO	N. A.	631.6	302.6	214.5	149.5

Table 4: Wall clock run time for different task allocation (Example 3)

The results with clustering are a little bit better, except in the one worker case in which clustering makes no difference, but not significantly better. Note also that 16 subtree partitioning shows slightly better performance for 1 to 4 workstations, while 64 subtree partitioning is better for around 10 workstations.

For our next experiments, we use machines with different computing power to reflect the real world situation where machines are heterogeneous. We choose 4 types of workstations: SUN SPARC station 1 (SS-1) to SUN Ultra station 2 (Ultra-2) which is about 20 times faster than SS-1. One of the six workers is our department server and has a relatively heavy load (average work load 2.5), so it could perform only 1 job while idle machines of the same type finished 3.5 jobs. The results are listed in Table 5 and show the effectiveness of our scheme using master-worker models for load balancing even with significant differences in computing power. Note that the ratio of run time is

1.4 to 1.5 which indicates that load balancing worked pretty well.

Machine Type	SS-5	SS-1	SS-2	SS-2	SS-5	SS-5	Ultra-2
Operating System	SunOS	SunOS	SunOS	SunOS	SunOS	Solaris	Solaris
CPU clock speed (MHz)	85	20	40	40	85	85	400*
Work Load (last 5 min)	1.48	0.05	0.32	0.02	2.59	0.02	0.03
Subtree allocated	master	2	3	4	4	10	41
Upward time	195.9	173.5	120.8	169.1	136.8	146.4	158.1
Downward time	721.5	721.4	483.9	561.5	582.1	668.3	500.1

Table 5: Performance result of various machines (Example 3)
*A Ultra has a different CPU architecture so the clock speed is scaled relative to the SS.

Example 4 **(Idle workstation utilization)** In these experiments, we examine not only speedup but also the effectiveness of networks of workstations (NOW) in real world situations. We performed our experiments at Wednesday 3 to 4 PM which is one of busiest time slots in the department. The PLinda system automatically checked 40 listed workstations and found only 10 machines were busy[2] in the last 60 seconds. We ran an example on 25 workers for 16M grid points with $K = 16$ and $p = 45$ for double precision accuracy, which requires about 256 Mbytes memory for a sequential solver. About half of the 25 workers we used are at the SS-1 to SS-2 level and half at the SS-5 level or above. They include a mixture of machines for private use, public use, and a department server. Figure 5 summarizes the results. It shows that two failures happened in the subtree allocation stage, one in the upward pass, and four in the downward pass. The time plotted of deleted process is time spent only by the successful pass, not the sum of the deleted and the backup process.

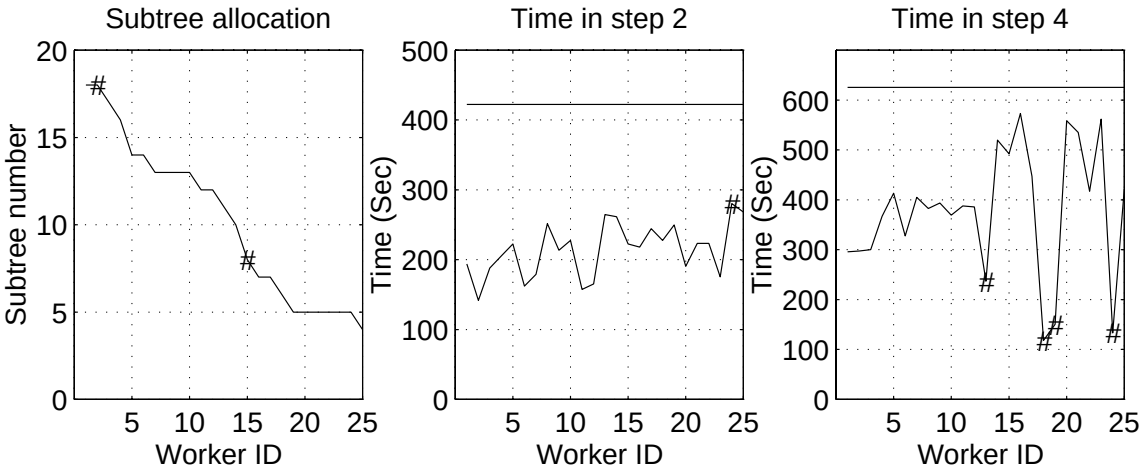

Figure 5: **Performance result using 25 workers (Example 4)**
Worker ID of 25 workers is assigned in order of number of subtrees allocated. Machines which failed at some stage are marked with #. In timing figures, top horizontal lines shows time spent until the master collects all results.

It is interesting to note the big gap between the time spent by worker 24 (the slowest) and by the master in the middle of Figure 5. The time spent by the master is counted from step 1 so

[2]The PLinda system are set to check only keyboard or mouse events

it takes more time than that used by the slowest worker for step 2; however, the gap is mainly caused by the failure of worker 24. The PLinda system automatically allocates worker processes to machines which respond faster. 25 faster machines participated in computation and 2 machines failed in the subtree allocation step, therefore, 3 slower machines were not used. Failure of a worker at the end of step 2 may double the overall run time of step 2. By contrast, the failed processes in step 4 on slower machines cause less damage, since their remaining tasks are allocated to faster machines which have already finished their own computation.

Overall we finished our job in 1050 seconds which is about 6 times (12 times) faster than projected computational time on a dedicated SS-5 (SS-2) with 256 MB memory.

5 Conclusions and Future Work

Our parallel Poisson solver using FMM localizes most of the computation and memory usage to a process in order to reduce the amount of data communication between adjacent processes. It also minimizes the number of data transfers and synchronizations. Our examples show linear speedup under a controlled environment with 8 workstations and demonstrate the effectiveness of NOW for problems requiring intensive computation and large memory. The PLinda system allows us to utilize more than half of the machines for our computation even in the busiest time slot of a week by automatically detecting the idle status of each workstation and migrating processes in machines which become in active use to idle machines.

We did not try to optimize the solver in our computing environment or use a clever strategy for task allocation. There are many research questions concerning the effective use of NOW in a general environment. These include a task allocation strategy using a statistical model to predict which machines will be idle, a failure detection mechanism, a policy for setting priorities, and an automatic job migration method for load balancing, where a fast idle machine which finished its own tasks duplicates tasks for other machines as a backup.

Despite these unanswered questions, developing parallel algorithms which minimize the amount and the frequency of data transfer is certainly one of the keys to obtaining good results with NOW. We believe that NOW will be an effective platform for large scale scientific computation, and plan to continue work on parallelization in this environment.

Acknowledgements: We thank Surendranath Talla for his PLinda coding help in early stage of our work, and Frank Ethridge, Leslie Greengard, and Dennis Shasha for reading our paper and giving many valuable comments. The experiments have been done at the Courant Institute of Mathematical Sciences (CIMS) of New York University with the support of the Courant Mathematics and Computing Laboratory (CMCL) and the Department of Computer Science.

References

[1] T. F. Chan and D. C. Resasco, A domain-decomposed fast poisson solver on a rectangle, *SIAM J. Sci. Stat. Comput.* **8**(1) S14–26, (1987).

[2] D. Lee, Fast parallel solution of the poisson equation on irregular domains, *Numer. Algorithms* **8**(2-4) 347–362, (1994).

[3] U. Schumann and M. Strietzel, Parallel solution of tridiagonal systems for the poisson equation, *J. Sci. Comput.* **10**(2) 181–190, (1995).

[4] P. N. Swarztrauber and R. A. Sweet, Vector and parallel methods for the direct solution of poisson's equation, *J. Comput. Appl. Math.* **27**(1-2) 241–263, (1989).

[5] L. Greengard and J.-Y. Lee, A direct adaptive poisson solver of arbitrary order accuracy, *J. Comput. Phys.* **125** 415–424, (1996).

[6] Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang, *Spectral Methods in Fluid Dynamics*, Society for Industrial and Applied Mathematics, Philadelphia, (1988).

[7] A. T. Patera, A spectral element method for fluid dynamics: laminar flow in a fluid expansion, *J. Comput. Phys.* **54** 468–488, (1984).

[8] L. Greengard and V. Rokhlin, A fast algorithm for particle simulations, *J. Comput. Phys.* **73** 325–348, (1987).

[9] J. Carrier, L. Greegard, and V. Rokhlin, A fast adaptive multipole algorithm for particle simulation, *SIAM J. Sci. Stat. Comput.* **9**(4) 669-686 (1987).

[10] F. W. Dorr, The direct solution of the discrete poisson equation on a rectangle, *SIAM Rev.* **12** 248–263, (1970).

[11] C. Anderson, Domain decomposition techniques and the solution of poisson's equation in infinite domains, In *the Second International Symposium on Domain Decomposition methods* pages 129–139, (1987).

[12] A. Brandt, Multi-level adaptive solutions to boundary value problems, *Math. Comp.* **31** 330–390, (1977).

[13] M. Griebel, Parallel domain-oriented multilevel methods, *SIAM J. Sci. Comput.* **16**(5) 1105–1125, September (1995).

[14] S. Kim, Parallel multidomain iterative algorithms for the helmbholtz wave equation, *Appl. Numer. Math.* **17** 411–429, (1995).

[15] N. Carriero and D. Gelernter, *How to write parallel programs : a first course*, MIT Press, Cambridge, (1992).

[16] K. Jeong, *Fault-tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions*, PhD thesis, New York University, (1996).

[17] K. Jeong and D. Shasha, Plinda 2.0: A transactional/checkpointing approach to fault tolerant linda, In *Proc. of the 13th International Symposium on Reliable Distributed Systems*, October (1994).

[18] K. Jeong, D. Shasha, S. Talla, and P. Wyckoff, An approach to fault-tolerant parallel processing on intermittently idle, heterogeneous workstations, In *Proc. the 27th International Symposium on Fault Tolerant Computing*, June (1997).

[19] N. Carriero, *Implementing Tuple Space Machines*, PhD thesis, Yale University, Department of Computer Science, (1987).