1992

Macromolecular Electron Density Averaging on Distributed Memory MIMD Systems

Dan C. Marinescu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Marius A. Cornea-Hasegan

Robert E. Lynch
Purdue University, rel@cs.purdue.edu

Michael G. Rossmann

Report Number:
92-019

MACROMOLECULAR ELECTRON DENSITY AVERAGING ON DISTRIBUTED MEMORY MIMD SYSTEMS

Dan C. Marinescu
John R. Rice
Marius A. Cornea-Hasegan
Robert E. Lynch
Michael G. Rossmann

CSD-TR-92-019
March 1992

Macromolecular Electron Density Averaging on Distributed Memory MIMD Systems

Dan C. Marinescu, John R. Rice, Marius A. Cornea-Hasegan
Department of Computer Sciences

Robert E. Lynch
Departments of Computer Sciences and Mathematics

and
Michael G. Rossmann
Department of Biological Sciences

Purdue University
April 2, 1992

Abstract

The paper discusses algorithms and programs for electron density averaging using a distributed memory MIMD system. Electron density averaging is a computationally intensive step needed for phase refinement and extension in the computation of the 3-D structure of macromolecules like proteins and viruses. The determination of a single structure may require thousands of hours of CPU time for traditional supercomputers. The approach discussed in this paper leads to a reduction by one to two orders of magnitude of the computing time. The program runs on an Intel iPSC/860 and on the Touchstone Delta system and uses a user controlled shared virtual memory and a dynamic load balancing mechanism.

Contents

1 Introduction 3

2 Electron Density Averaging in Macromolecular Structure Determination 4

3 A Model of the Computation and Parallel Algorithms for Electron Density Averaging 8

3.1 The model 8

3.2 Parallel algorithms to average the electron density 9

3.2.1 A data parallel algorithm. 10

3.2.2 A control parallel algorithm. 11

4 A Shared Virtual Memory for Electron Density Averaging 14

4.1 Memory constraints 14

4.2 Implementation of a shared virtual memory for a distributed memory MIMD system 15

5 Load Balancing 17

6 Experimental Results 18

7 Conclusions 25

1 Introduction

X-ray crystallography is a major tool for the structure determination of proteins and viruses. The determination of the 3-dimensional atomic structure of biological macromolecules requires a huge amount of computing resources. Currently, processing of experimental data, phase determination, and model building are being done by a series of complicated *sequential* programs. The memory and time requirements for the analysis of virus and enzyme systems are at the limit of existing vector computers such as the Cyber 205. For example, a single iteration required to average a 3.0Å resolution electron density map with 120-fold redundancy (see Section 2) for the Coxsackie B_3 virus crystals takes in excess of 64,000 seconds (18 hours) of CPU time, and the memory requirements are in the 100 Mbyte range. Soon new sources of intense X-ray radiation will become available. The resulting increased accuracy and rate of data acquisition will allow the analysis of more complex molecular assemblies (e.g., virus receptor complexes) and will therefore require much greater computing resources than are or will be available using sequential computers.

For macromolecular structure determination, parallel algorithms and data partitioning schemes are required to exploit fully the distributed memory multiprocessor systems like the Intel iPSC/860 hypercube or the Touchstone Delta 2-D mesh. These provide the computing resources necessary for such computationally intensive tasks. For example, a 64 node Intel iPSC/860 with 16 Mbytes per node has a peak speed of about 4 Gflops and 1 Gbyte of memory. The Touchstone Delta system has 517 computational nodes, a peak performance of more than 30 Gflops, and more than 8 Gbytes of memory. We expect that within 2 to 4 years, machines of this class will have peak performances of 10 to 30 times these rates.

In addition to the conceptual difficulties of designing and implementing concurrent methods, there are also problems due to the rather primitive program development and run-time support environments. For example, in our Intel iPSC/860 implementation, a shared virtual memory had to be designed and implemented because the memory requirements exceed the amount of real storage available in each node.

Some of the most difficult issues are related to performance tuning. Achieving load balance among a collection of processors is always essential in obtaining peak performance on MIMD (Multiple Instruction Multiple Data) systems. Yet achieving a balanced load scheme together with an efficient shared virtual memory system is a very difficult task.

In this paper we describe our methods and experience in redesigning for MIMD systems the sequential algorithms which carry out one of the extremely time consuming parts of macromolecular structure determination. The calculation is explained in physical terms in Section 2. Section 3 presents an abstract model of the computation as well as the parallel algorithms for the computation. For this problem, a key to reducing significantly computation and elapsed time was the implementation of a program controlled virtual shared memory, as explained in Section 4. Some important and practical considerations regarding load balancing policies are discussed in Section 5. Experimental performance results are presented

Vector processor	MIMD systems		
Cyber 205	16 Node iPSC/860	64 Node iPSC/860	128 Nodes of Delta
12,000†	3,000‡	680‡	550‡

Table 1: Time in seconds for one iteration of electron averaging of $130 \times 130 \times 400 = 6,760,000$ grid points. †: CPU time; ‡: elapsed time.

in Section 6.

2 Electron Density Averaging in Macromolecular Structure Determination

In this section we describe in physical terms one of the steps of the computer determination of a macromolecular structure which we have successfully implemented on distributed memory MIMD systems. The technical aspects of the implementation are presented in Sections 3, 4, and 5.

The process which was made to run efficiently on a parallel processor is the 'electron density averaging'. Although this is just one of many steps leading to the determination of a structure, it is an iterative procedure and represents a large proportion of the total computer time required to determine and analyze a virus structure. Typically hundreds of iterations are carried out. Table 1 lists times for a *single* iteration for a sample medium sized virus problem (see Section 6). In contrast to the *elapsed times* listed for the MIMD systems, the 3.33 hours of Cyber 205 is *CPU time* and to this CPU time must be added 'turn-around-time' which might be several *days* with a typical load when a job is run that carries out 10 iterations.

X-ray diffraction in crystals occurs because of their constructive reflection caused by scattering from electrons in planes of atoms. An X-ray diffraction pattern is characteristic of the specific configuration of the regular arrangement of the atoms in the *unit cell*, whose translates in 3-dimensions make up the crystal. Each unit cell contains the same atoms arranged in the same way. A crystalline form of a macromolecule, such as a virus having millions of atoms, has one or more macromolecules at specific locations and in specific orientations in each unit cell.

When a macromolecule (together with solvent, such as water) can be crystallized, data from X-ray diffraction experiments can be collected and results from millions of reflections are recorded. These measurements lead to observed values of the structure amplitudes $|F_{h,k,\ell}|$ which can be used as moduli of the Fourier coefficients of the periodic electron density, ρ. The accuracy depends on the *resolution*, which indicates the spacing between planes of atoms which the experimental data can resolve; the higher the resolution, the more structure amplitudes can be measured (but the units of resolution are angstroms, so an experiment with a 3Å resolution has 'higher' resolution than one at 10Å).

In general for such large molecules, the phases of the Fourier coefficients $F_{h,k,\ell}$ cannot be determined experimentally. But, initial estimates of phases for some low resolution coefficients can be computed approximately from crude models of the macromolecules (see [Ross 72]). Given the phases and the experimentally observed structure amplitudes, Fast Fourier Transforms (FFTs) can be used to obtain an approximate electron density distribution at points of a grid $\mathbf{x}_{i,j,k} = (x_i, y_j, z_k)$, $1 \leq i \leq n_x$, $1 \leq j \leq n_y$, $1 \leq k \leq n_z$. The number $N = n_x \times n_y \times n_z$ of grid points increases as the resolution of the Fourier coefficients increases and might reach 10^9 for some large problems.

There are two types of symmetries that can be exploited in determining the electron density distribution. First, there is the symmetry that is present in the 3-dimensional crystal lattice. Any symmetry operator of this kind ("crystallographic symmetry") is true throughout the extent of the crystal. Second, there is the *local* symmetry ("noncrystallographic symmetry") which pertains only to a limited volume such as a single virus particle. The electron density at points related by this local symmetry should be equal, which permits electron density averaging and modification. The altered density should then be more accurate and this allows for iterative refinement of the phases of the Fourier coefficients.

Symmetries can be expressed in terms of transformations S_m which carry a point into another with the same physical properties (e.g., same electron density). Suppose, for example, that the molecule has the noncrystallographic symmetry of an icosahedron — i.e., 6 five-fold, 10 three-fold, and 30 two-fold axes. Then 1/60-th of the virus is repeated 60 times to make up the entire virus. Thus, there are $M = 60$ transformations S_m which carry a point $\mathbf{x}_{i,j,k}$ into $\mathbf{x}^{(m)}_{i,j,k} = S_m[\mathbf{x}_{i,j,k}]$ where the electron density, ρ, is the same: $\rho(\mathbf{x}^{(m)}_{i,j,k}) = \rho(\mathbf{x}_{i,j,k})$.

An improved value of the electron density is obtained by averaging:

$$\overline{\rho_{i,j,k}} = \frac{1}{M} \sum_{m=1}^{M} \rho(S_m[\mathbf{x}_{i,j,k}]);$$

the original estimate of ρ is replaced with $\overline{\rho_{i,j,k}}$ at each of the M points $\mathbf{x}^{(m)}_{i,j,k}$. To carry out such averaging, one needs to know the symmetries involved, to have approximate values of electron density at grid points, and to have an identifier, called a *mask*, whose value specifies that the grid point is in a particular molecule or in the solvent. The electron density at points in the solvent are *flattened*, that is, set equal to a constant. A detailed discussion of the averaging is described in [Ross 92].

Sections 3, 4, and 5 of this paper describe some of the details of our implementation of this process on an MIMD system. We outline subsequent procedures, which we also intend to implement on an MIMD system.

After computing a new approximation of ρ by averaging, its Fourier coefficients $\hat{F}_{h,k,\ell}$ are computed by using FFTs. The phases of these should be more accurate than the phases

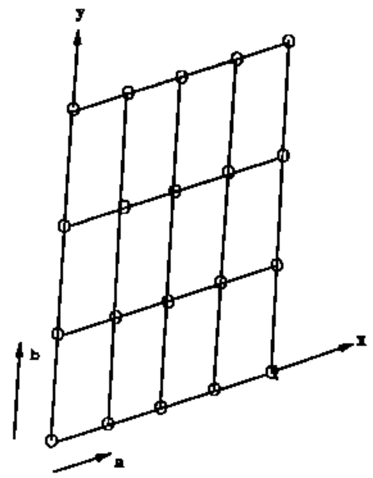

Figure 1: Simplified schematic of a planar crystal showing 12 unit cells. The circles indicate lattice points.

used before the averaging. The new phases are combined with the experimentally observed moduli $|F_{h,k,\ell}|$ to obtain new Fourier coefficients. Then the process is repeated: electron density is computed at grid points, it is averaged, and so on, until the phases converge. At this point, the resolution can be extended.

A Simplified Example. We present a simplified, example as an introduction to the material discussed in Sections 3, 4, and 5.

To indicate how the calculation is carried out, and, most importantly, how data flows into and out of memory during the averaging, we discuss the simplified situation of a planar crystal (Figure 1). The electron density $\rho(\mathbf{x})$ at any point in any unit cell is identical to the density at all points $\mathbf{x} + j\mathbf{a} + k\mathbf{b}$ where j, k are integers and $\mathbf{a}$, $\mathbf{b}$ are basis vectors: the edges of a specific unit cell.

Figure 2 shows a single unit cell and four hypothetical triangular molecules. The unit cell is the parallelogram with boundaries indicated by the solid lines. Only a portion of each of the molecules at the four lattice points is inside the unit cell, giving a total of one molecule per unit cell. We assume that each molecule has a 3-fold axis of symmetry perpendicular to its centroid — that is, rotations of 120° leaves the molecule unchanged. This is an example of a *noncrystallographic* symmetry which is local, confined within the envelope that defines the molecular boundary. Thus the collection of unit cells (the crystal lattice) is not invariant under such a transformation. The *crystallographic* operators of this particular example consist only of integral numbers of translates by the basis vectors $\mathbf{a}$ and $\mathbf{b}$.

The electron density is given at grid points only in the unit cell in Figure 2. Because of the noncrystallographic symmetry, the electron density is equal at the points labeled A, B,

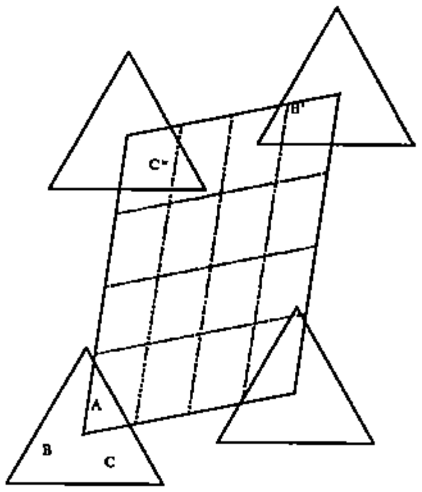

Figure 2: Unit cell and molecules with 3-fold axes.

and C and only point A is in the unit cell. The crystallographic symmetry implies that the electron density is the same at the points A, B′, and C″. Consequently, the electron density at points B′ and C″ can be used with that at A to improve the accuracy of the density at point A. The densities at these three points are averaged; similarly for other grid points in the unit cell. The points A, B' and C'' are in different molecules and their respective neighbor grid points have different mask values.

This averaging is done over a grid of points in the unit cell and the computation processes the grid points in order (in some real cases, a volume smaller than the unit cell, called the *asymmetric unit*, can be used instead of the full unit cell). Suppose the point A is a grid point; then the average

$$\overline{\rho_{\rm A}} = [\rho_{\rm A} + \rho_{\rm B'} + \rho_{\rm C''}]/3$$

is formed, where $\rho_{\rm A}$, $\rho_{\rm B'}$, $\rho_{\rm C''}$, denote the electron densities at A, B′, C″. This value is used at the grid point A during subsequent parts of the analysis (i.e., during the FFT computations).

However, two problems arise. First, usually the different points B', C'', associated with A by noncrystallographic and crystallographic symmetry are not grid points and so values of $\rho_{\rm B'}$, and $\rho_{C''}$ are not directly available. Bilinear interpolation to values at four nearest neighbor grid points is used to obtain an interpolated value of, say, $\rho_{B'}$, and $\rho_{C''}$ to be used in the averaging. (In 3-dimensions, trilinear interpolation to eight nearest neighbor grid points is used.)

The second problem is the topic of the remainder of this paper. It arises because in real cases the amount of data does not fit into the main memory of the computer whether this is a vector processor (as now used for production) or the 16 Mbyte memory modules of current MIMD machines. A solution to this problem is the creation of a data structure and an efficient data management scheme which allow rapid access to those groups of grid values which have to be combined in averages. Although this is like a virtual memory system, the fetching of data is controlled by the program rather than the operating system. We indicate how it works by considering the simplified 2-dimensional example.

The grid points are partitioned into "bricks" (they are solids in 3-dimensions) as indicated by the dotted lines in Figure 2. To do the averaging for point A, the bricks which contain the other 2 points, B', C'', related to A by symmetry, must be in the main memory. If they are in the main memory, then the averaging takes place; if some are not, then the required bricks are fetched from an external storage device, such as a disk. If the main memory is already full of bricks, then a brick must be discarded (overwritten) to make room for another one, resulting in a 'brick fault'.

3 A Model of the Computation and Parallel Algorithms for Electron Density Averaging

The electron density averaging, described in Section 2, is just one problem in a large class, which requires similar data management in order that its calculation be done efficiently on an MIMD system. In Section 3.1 we present a high level model of such problems. This model represents a class of computations for 3-D grids which correlate physical information (temperature, density, electron density, etc.) of subsets of points related by a group of transformations. The points in each subset are scattered throughout the entire grid and the amount of storage and computing time for averaging at a grid point varies significantly.

The model is used to understand the requirements for a shared virtual memory system and for load balancing. Then, in Section 3.2, we discuss parallel algorithms for a specific problem in this class, namely, the electron density averaging.

3.1 The model

Consider a 3-D grid G with $N = n_x \times n_y \times n_z$ nodes and a collection of objects

$$A = \{a_1, a_2, \ldots, a_N\}$$

Each object $a_i \in A$ consists of geometric location information $\gamma_i = (x_i, y_i, z_i)$ and physical information $\{\rho_i^{(1)}, \ldots, \rho_i^{(q)}\}$ associated with grid point Q_i of G. The geometric information consists of the coordinates of the node.

The discussion is restricted to the case where the physical information has only two components (μ_i, ρ_i) and where $\mu_i = \rho_i^{(1)}$ is a binary indicator which partitions A into two disjoint subsets, $A = B \cup C$ as follows

$$a_i \in B \quad \text{iff} \quad \mu_i \neq 0, \qquad \text{and} \qquad a_i \in C \quad \text{iff} \quad \mu_i = 0$$

The model includes a group of m transformations,

$$S = \{S_1, S_2, \ldots, S_m\}$$

each of which transforms the space into itself. The nature of these transformations is illustrated in Figure 2. We see that the set B consists of many physical copies of the same object, these objects are irregularly distributed and oriented. Thus the physical quantities to be determined (the ρ_i) are measured at many different places in the grid. A transformation maps a copy of the object onto another. Or, it uses known symmetry of the objects to map one part of an object onto a corresponding part. Once these correspondences are known from the map S, then one can average the multiple measures of the same quantities to obtain more accurate estimates of their values.

In more mathematical terms, every $S_j \in S$, associates with each $a_i \in A$ a set

$$S_j(a_i) = \{a_{i,1}, a_{i,2}, \ldots, a_{i,n_j}\}$$

of elements of $a_{i,k} \in A$. Recall that $a_{i,k} = (\mu_{i,k}, \rho_{i,k})$. Here n_j is the number of nodes mapped into the ith node by the jth transformation. This number is nearly a constant which is denoted by n. Call

$$S(a_i) = \{S_1(a_i), S_2(a_i), \ldots S_m(a_i)\}$$

and denote by $|S(a_i)|$ the cardinality of $S(a_i)$. The associations produced by S are such that when $a_i \in B$ then $|S(a_i)| = n \times m$ and when $a_i \in C$ then $|S(a_i)| = 1$.

Given A and S, consider a computation $\mathcal{C}$ which transforms A into $A' = \mathcal{C}(A)$. For every $a_i \in A$, $\mathcal{C}$ first determines $S(a_i)$ and then computes ρ'_i, an improved value of ρ_i. The transformation of $a_i = (\mu_i, \rho_i)$, $a_i \in A$ into $a'_i = (\mu_i, \rho'_i)$, $a'_i \in A'$ preserves the membership of a_i, namely, if $a_i \in B$ then $a'_i \in B'$ and if $a_i \in C$ then $a'_i \in C'$.

The time and space complexity to compute ρ'_i for each a_i is $\mathcal{O}(|S(a_i)|)$. It follows that the time and space complexities of the computation $\mathcal{C}$ for different grid points are very different, namely

$$\mathcal{O}(n \times m) \quad \text{if} \quad a_i \in B, \qquad \text{and} \qquad \mathcal{O}(1) \quad \text{if} \quad a_i \in C.$$

The time and space complexity for the entire computation $\mathcal{C}$ is $\mathcal{O}(N)$, where N is the number of grid points.

In typical problems of electron density averaging in viruses, values of interest are $N \simeq 10^9$, $m \simeq 10^2$, $n \simeq 10$, $|B|/N \simeq 0.5$.

3.2 Parallel algorithms to average the electron density

Consider a parallel system with P processing elements, PEs, communicating with each other through an interconnection network. A *processing element* consists of a processor, memory, and possibly other elements such as co-processors, I/O interfaces, network interfaces, etc. Our objective is to use such systems to reduce the time for tasks like the electron density averaging computation. Parallel algorithms must be developed and these often have little resemblance to the sequential algorithms used to solve the same problem. These algorithms are usually specific for a particular computer architecture. There are two basic classes of parallel computers, the Single Instruction Multiple Data, SIMD systems, and the Multiple Instruction Multiple Data, MIMD systems. Two parallel algorithms, one for each type of system, for computing the average electron density at grid points in the crystallographic asymmetric unit are presented below.

The purpose of parallel computing is to reduce the time required to perform a computation. The performance of a parallel algorithm and of its implementation on a parallel system is measured by the *speedup*. Let $T(P)$ denote the time to finish the computation when P processing elements are used. The speedup, $\mathbf{S}(P)$, with P processing elements is defined as $\mathbf{S}(P) = T(1)/T(P)$.

The *PE utilization*, η, is defined as the ratio of the time a PE performs useful computations to the total time the PE is allocated to that particular task. For example if a computation takes 10 seconds and the PE is idle for 30 seconds, waiting for results from other PEs, then the processor utilization is 0.25. If the η_P denotes the *average processor utilization* with P processing elements, then $\mathbf{S}(P) = P \times \eta_P$; see for example [Fox 87]. This corresponds to the intuition that the largest possible speedup with P processing elements is P and can be achieved if all PEs perform only useful computations.

The basic algorithm for computing the average electron density is described in Section 2. In §3.2.1 a data parallel algorithm for averaging on SIMD systems is presented. This is followed, in §3.3.2, with a discussion of parallel computations on distributed memory MIMD systems and a control parallel algorithm is described in more detail.

3.2.1 A data parallel algorithm.

The data parallel approach is best suited for computations in which *identical transformations* are applied to a collection of similar objects, and when the transformations can be carried out *independently* for all the objects.

A SIMD system can be used to carry out a data parallel computation as follows. A *control unit*, CU, executes a program stored in its local memory. At each step it broadcasts a single instruction or a block of instructions to the PEs it controls. At each step a PE can either be enabled or disabled. If the PE is *enabled* it executes the current instruction using data from its local memory and sets its condition code accordingly. If the PE is *disabled* it does not execute the current instruction. The execution is *synchronous*: all PEs simultaneously execute the same instructions using their own data. A significant speedup can be obtained only if most of the PEs are enabled most of the time.

A data parallel algorithm for computing the average electron density transforms all grid points in parallel in the following way. It assigns every grid point to a virtual processor and then it maps virtual processors to the PEs of an SIMD system according to some strategy. The control unit has all PEs it controls first fetch the data associated with a grid point and then check its mask. All PEs with grid points in the subset C (solvent) are disabled at this stage. Then the control unit executes the program for averaging on all enabled PEs (those of B with grid points in the virus). Finally, the PEs with grid points in the subset B are disabled, those in set C are enabled and the simple computation, flattening, is carried out. Call c_{ave} the time it takes to perform electron density averaging and c_{flat} the time it takes to perform the flattening algorithm. For the problem of electron density averaging, $c_{ave} \gg c_{flat}$.

We now give an approximate analysis of the utilization and speedup of this approach. Assume first that there is a one to one mapping of virtual processors to real PEs. In this unrealistic case the number of PEs is equal to the number of grid points, $P = N$. The total execution time is $c_{ave} + c_{flat} \simeq c_{ave}$. The average computation time is $(|B| \times c_{ave} + |C| \times c_{flat})/N \simeq |B| \times c_{ave}/N$. Thus the expected PE utilization is $\eta_N \simeq |B|/N$ and the speedup is $S(N) = |B|$. This simplified analysis is consistent with the intuition that only $|B|$ processing elements do a significant amount of work, these are the ones assigned to grid points in the virus and the remaining PEs are idle most of the time.

3.2.2 A control parallel algorithm.

A control parallel algorithm has multiple threads of control running asynchronously on a MIMD system. Each processor executes a program stored in its local memory, uses locally stored data, and communicates with other PEs via messages. If all PEs run the same program, but use different data, then the computation follows the Same Program Multiple Data, SPMD, paradigm. Even in case of the SPMD execution, the PEs might complete their computations at different times.

Our control parallel algorithm for averaging the electron density follows closely the SPMD paradigm. Given P processing elements, the data is partitioned into $P-1$ sub-domains. One PE is used to control the entire computation while each of the remaining $P-1$ processing elements execute the same program and perform the same computation for each of the grid points in the sub-domain assigned to it. For each grid point a processor determines if the point belongs to the subset B by checking its mask. If it does, then the coordinates of the other $M-1$ points related by noncrystallographic symmetry are computed and, when necessary, the corresponding points in the unit cell are obtain by crystallographic symmetry. Such points do not coincide with grid points, and an 8-point trilinear interpolant is used to get an approximate electron density at this point. Finally, the average electron density is computed and it replaces the density at the original grid point. On the other hand, if the gird point belongs to the subset C then its electron density is flattened (set to a constant).

A data partitioning strategy. The partitioning strategy profoundly affects the overall performance. It should be tailored according to the computational algorithm and the computing resources available, in particular the amount of local memory in each PE. Different strategies to partition the data can be considered. For example, each partition may consist of a number of grid points picked at random from the grid. One may attempt to sort the data and then to assign to each partition an equal number of grid points in sets B and C to ensure that the computation requirements for each partition are balanced. Another approach is to use some sort of geometric partitioning.

One geometric strategy is the following. Partition the 3-D grid with $N = n_x \times n_y \times n_z$ points into N_B small volumes called *bricks*, denoted by $B_1, B_2, \ldots, B_{N_B}$. Each brick consists of $b = b_x \times b_y \times b_z$ contiguous grid points. For the sake of convenience, assume that n_x, n_y and n_z are multiples of b_x, b_y, and b_z, respectively. The grid G is then a 3-D brick structure with N_x, N_y and N_z bricks in each dimension where

$$N_x = \frac{n_x}{b_x}, N_y = \frac{n_y}{b_y}, N_z = \frac{n_z}{b_z} \qquad \text{and} \qquad N_B = N_x \times N_y \times N_z.$$

The bricks are numbered consecutively from 1 to N_B. The brick numbered nbrk has the 3-D coordinates, xbrk, ybrk and zbrk such that

$$\mathtt{nbrk} = \mathtt{xbrk} + (\mathtt{ybrk} - 1) \times N_x + (\mathtt{zbrk} - 1) \times N_x \times N_y.$$

and $\mathtt{nbrk} \leq N_B$, $\mathtt{xbrk} \leq N_x$, $\mathtt{ybrk} \leq N_y$ and $\mathtt{zbrk} \leq N_z$. Any particular brick may consist of grid points belonging only to the subset B, only to the subset C, or to both.

In general, the partitioning strategy for computing the average electron density should attempt to satisfy the following criteria.

(a) It should preserve geometric proximity as much as possible: two grid points close to one another should be assigned to the same partition. This is because transformations of the coordinates of points related by noncrystallographic symmetry preserve geometric distances. Thus this type of partitioning ensures some *locality of reference* for the electron averaging.

(b) Each partition should have the same number of points belonging to the subset B so as to achieve some form of balancing of the computational load for each partition.

(c) The strategy should be independent of the actual problem size, the size of the memory available in each node, and the number of PEs used to solve the problem.

A data mapping strategy. The bricks can be assigned to the PEs following different mapping strategies. The main goal is to have an assignment which results in the load being balanced among the PEs. Load balancing is critical for obtaining a large speedup. For example if one PE takes twice as much time to complete its execution as the remaining $P - 1$, then the speedup cannot be larger than $P/2$ if P is large. We consider three basic data mapping strategies:

1: *Static mapping*: all bricks are assigned at the beginning of the computation,

2: *Dynamic mapping*: a dispatcher distributes the bricks to PEs upon request,

3: *Hybrid mapping*: a fraction of the bricks are assigned statically and the remaining bricks are assigned by a dispatcher upon request.

Given P processing elements and a problem with N_B bricks, a *static mapping* assigns to each PE an equal number of bricks, N_B/P. For simplicity, suppose that N_B is a multiple of P. The static assignment can preserve some of the geometric proximity of the bricks. For example, bricks 1 to N_B/P are assigned to PE_1, bricks $N_B/P+1$ to $2 \times N_B/P$ are assigned to PE_2, and so on. A static mapping is very likely to lead to load imbalance because the computing time per brick can be very different. An alternative is to have small bricks and assign to each PE many bricks scattered throughout the entire grid G. The hope is that eventually the total number of grid points in B are uniformly distributed among all PEs. Such a random scattered mapping is likely to nullify the locality of reference.

Dynamic or hybrid mapping schemes are potentially much better and such schemes are discussed in Section 5.

The storage management. The space requirements for computing the average electron density are considerable. Assuming that the data are densely packed, at least $2N$ bytes are needed to store the electron density and the mask associated with the N grid points. For example, the space required for a grid with 500 grid points in each direction is at least 125 Mbytes. The amount of physical memory available in each PE of existing MIMD systems is in the 16 Mbyte range; this is considerably smaller than the size of the address space required by such computations. The function of a storage management scheme is to fold a large address space into the smaller physical space actually available in each PE.

We introduce more precise terminology as follows. A *master brick* is that brick whose electron density is being averaged/flattened by a PE. All the bricks needed in averaging for a particular master brick, B_j, form its *working set*, denoted by W_{B_j}. The data mapping assigns master bricks to PEs for processing while the storage management fetches the bricks in the working set W_{B_j}. Every time a brick in W_{B_j} is not present in the local memory of the PE processing B_j, a *brick fault* occurs. In case of a brick fault the averaging process stops, the brick is located by the storage management scheme, it is brought into the local memory of the PE, and then the averaging resumes. The time to fetch a brick is much larger than the time needed to compute the average electron density for one grid point; therefore brick faults greatly increase the processing time per master brick.

If n_b bricks are needed to perform the averaging for one grid point then the *locality of reference* implies that the working set of a brick is considerably smaller than $n_b \times b$ with b the number of grid points per brick. Recall that grid points related by noncrystallographic symmetry are scattered throughout the entire virus therefore a good storage management scheme is one which exploits the locality of reference to minimize the number of brick faults.

If, when the processing of brick B_j begins, a large fraction of its working set W_{B_j} is already in the local memory, then the total processing time for B_j is much less than if all of W_{B_j} had to be fetched into local memory.

Several methods for reducing fetch-time can be contemplated. For example, one may attempt to distribute all the bricks to the PEs. This approach is feasible if the memory available in each PE is larger than $2b \times N_B/P$ bytes, assuming 2 bytes per datum are required. Another approach is to keep all bricks on an external storage device (disk system) and fetch them as needed. The approach exploits the locality of reference and keeps in the local memory the most recently used bricks. This virtual memory approach is discussed in the next section.

4 A Shared Virtual Memory for Electron Density Averaging

4.1 Memory constraints

The electron density averaging computation has the potential of achieving a linear speedup because averaging for all bricks can be done independently with little or no communication among the PEs. But the amount of memory available in each PE of a distributed memory MIMD system limits the actual speedup that can be achieved on a real system. If the local memory of a PE could hold a replica of the entire data structure containing A, then a PE could run without communication from the instant the data structure is loaded to the end of the computation. Systems like the Intel iPSC/860 and NCUBE hypercubes or like the Touchstone Delta system currently have at most 16 Mbyte of memory per PE and the user's address space is about 12 Mbytes. To achieve the linear speedup, the electron density averaging computation requires 10 to 1000 times more memory per PE than is currently available.

Due to this memory constraint, the speedup observed on existing machines for averaging the electron density is significantly less than linear. The processing time per master brick increases as the number of brick faults increases. A simplified analysis of the expected speedup in the presence of brick faults follows. Call τ the expected time necessary to fetch to the local memory of a PE all the bricks in the working set of a given master brick and call c the expected computing time per master brick if no brick faults occur. Then the expected PE utilization in the presence of brick faults is

$$\eta = \frac{c}{c+\tau} = \frac{1}{1+\beta},$$

with $\beta = \tau/c$. The expected speedup with P processing elements is $\mathrm{S}(P) = P \times \eta$.

This shows that the larger is the number of brick faults when a master brick is processed (τ large and β large) the lower is the expected PE utilization and, therefore, the speedup. For this reason, a large speedup can only be obtained when there is a good locality of reference. In a typical case, for a single PE we have $\tau \approx 200$ seconds and $c \approx 25$ seconds which results in a utilization of $\eta \simeq 0.11$ or, for example, a speedup estimate of 7 for 64 processors. The data of Table 3 shows a measured speedup of 27. This suggests that the change in working sets for successive master bricks is modest because of the locality of reference.

4.2 Implementation of a shared virtual memory for a distributed memory MIMD system

This section presents an implementation of a user controlled *Shared Virtual Memory*, SVM, for distributed memory MIMD systems like the Intel iPSC/860 hypercube or for the Touchstone Delta System. Such MIMD systems can be partitioned into groups of PEs which can be assigned to different users. The SVM discussed in this paper allows all PEs of a partition to share a common address space larger than the space available in each PE. This SVM is implemented at the user level, it does not require any modifications of the operating system, and it was specifically designed for computations which correspond to the model described in Section 3.1. The SVM maps a large *Virtual Brick Area* located on an external storage device into a much smaller *Real Brick Area* available in the local memory of each PE.

The SVM allows all PEs of a given partition to access a Virtual Brick Area, VBA which consists of a collection of bricks. The brick size, b, and the total number of bricks, N_B, are specified by the user. The VBA for the electron density averaging program uses two direct access files, a read-only file for the original data (electron density and the mask for all grid points), and a write-only file for the averaged electron density. The two files are accessed using the *Concurrent File System*, [Intel 90]. Each file consists of $(N_B + 1)$ records: record 0 is a file descriptor and each subsequent record consists of one brick. The total size of the VBA is $2b \times (N_B + 1)$ bytes.

The Real Brick Area, RBA, is the local storage dedicated by each PE for storing bricks in *brick frames*. The size of a brick frame equals the size of a brick. The size of RBA is determined as soon as the size of the program is known. It is the difference between the size of the PE memory available and the amount used by the PE operating system, the PE program and the user's data.

The shared virtual memory is managed by an *SVM manager*, which performs a *dynamic address translation*. It maps *virtual brick numbers* used to identify bricks in VBA to real brick numbers used to access bricks in RBA. To perform the translation, the SVM manager uses a *translation table* with N_B entries, one for each virtual brick. The dynamic address translation works as follows. Given a virtual brick number `nbrk`, the `nbrk`-th entry of the translation table is examined. If the entry is a valid memory address, then the brick is present in RBA and the translation is complete. If the entry is invalid, then a *brick fault* occurs and the brick is fetched from the VBA.

The SVM node manager fetches bricks *on demand*, namely when the PE processing a master brick needs the corresponding data. An alternative is to use an anticipatory strategy. The bricks in the working set of a master brick could be brought in before the actual averaging begins. This approach is possible because it is known how the eight vertices of each brick are transformed by the noncrystallographic symmetry, thus all the bricks in the working set can be determined with little overhead before the actual averaging begins.

When the RBA is exhausted and an additional brick is needed, the SVM manager uses a *least-recently-used* (LRU) replacement algorithm to find a brick frame for the new brick. Each brick frame has a *reference counter* incremented every time the brick in that frame is referenced. To find a brick frame when none is available, the LRU algorithm first attempts to find a brick frame with a zero reference counter. If none exists, then the frame with the lowest reference counter is determined. The brick frame holding a master brick cannot be released until the processing of the master brick is completed.

The SVM manager has a rather simple user interface. It provides a function called `getentry(i,j,k)` which

(a) maps the global coordinates of a grid point, `(i,j,k)`, into a brick number **`nbrk`** and into local coordinates of the grid point, **`(ibrk, jbrk, kbrk)`**, in that brick,

(b) checks if `nbrk` is a valid brick number and if it is the last brick referenced. If so, it uses the known brick address in real memory to fetch the grid point information and then updates the brick reference counter,

(c) else it uses the dynamic address translation mechanism to get the memory location where the brick is stored.

(d) returns the electron density and the mask at `(i,j,k)`

The SVA manager provides several other functions like

`getmbrick(mbrk)`: fetches a new master brick from A into VBA,

`putmbrick(mbrk)`: stores a master brick from VBA into A',

`open` VBA, `close` VBA: to open and close the VBA,

`update`: updates all brick frame reference counts,

`report`: reports the number of brick faults and related data.

Though this type of storage management allows problems with different sizes to run on the same system, the actual performance depends upon the computing resources available, namely upon the amount of PE memory, upon the I/O bandwidth of the system, and upon the number of PEs used. The ratio size(VBA)/size(RBA) does not affect the performance directly. Recall that size(VBA) is determined by the problem size, by the number of bricks in the virus, and size(RBA) is determined by the amount of memory available. The measurements reported in this paper correspond to the case when this ratio is about 4.

The execution time increases significantly when size(RBA) is less than the working set of a brick. If size(RBA) is sufficient to accommodate the working set of a brick then the problem size does not directly affect the speedup. The I/O bandwidth is a major factor in deciding the number P of PEs used and it determines the actual start-up time. Let the brick size be $2b$ bytes, the average number of bricks in the working set of a brick be W, and the I/O transfer rate R bytes per second. The *start-up time*, t_s, is the time it takes for all PEs to fetch the brick working set of the master brick in the absence of *hot spot contention*:

$$t_s = \frac{P \times W \times 2b}{R}.$$

For example, a Delta submesh with 256 PEs, a working set of $W = 500$ bricks each of $2b = 8192$ bytes, and a transfer rate of $R = 5$ Mbytes/sec, results in $t_s \simeq 200$ seconds.

5 Load Balancing

In this section, two strategies for load balancing are discussed: a static one and a hybrid one, combining static and dynamic allocation.

In the *static load balancing approach*, the bricks are divided evenly among the $P - 1$ processing elements. Note that for a hypercube, P is determined by the dimension d of the sub-cube allocated to the problem: $P = 2^d$. In case of an MIMD systems with a 2-D mesh interconnection network, P can be chosen to match the geometry of the problem. For example, when $N_B = N_x \times N_y \times N_z$, a $N_x \times N_y$ mesh could be used and each PE could be allocated N_z bricks. If $N_z >> (N_x, N_y)$ and $N_z = a \times b$, an alternative method is to use an $(a \times b)$ mesh and to allocate to each PE an entire slab of $N_x \times N_y$ bricks.

Neither of these schemes guarantees that the load is perfectly balanced because geometric equipartition does not imply equal computational load. Call n_B the number of grid points of brick B_j in the subset B and note that $0 \leq n_B \leq b$ and further that b can be large, for example $b = 16 \times 16 \times 16 = 4096$ for the experiments reported in this paper. The value of n_B determines the size of the working set of B_j, the larger n_B, the larger W_{B_j} is likely to be, therefore the more time is needed to fetch all the bricks in W_{B_j}. The actual computing time to transform a master brick is also dependent upon n_B as shown in Section 3.1. For these reasons, the processing time for different master bricks can be very different; values in the range 10^{-3} to 300 seconds were observed for the test cases reported in the next section.

A better static load balancing approach which attempts to assign to every PE an equal number of grid points in B is under investigation. In a preprocessing phase, $n_B(j)$, the values

n_B for all B_j, $1 \leq j \leq N_B$, are determined. The first step of the mapping algorithm uses one of the static allocation schemes described in Section 3.2.2 and then determines for every PE$_k$ the value $r_k = \sum n_B(j)$ for all B_j assigned to PE$_k$. It also computes

$$r_{opt} = \frac{\sum_{j=1}^{N_B} n_B(j)}{P}$$

The algorithm then performs a number of iterations to smooth out the difference in r_k values. At each step it computes $r_{\max} = \max(r_k)$, $r_{\min} = \min(r_k)$ for $1 \leq k \leq P$, and identifies the two PEs, PE$_{\max}$ and PE$_{\min}$, which were assigned these extreme values. Then the list of bricks B_j assigned to PE$_{\max}$ is searched to find a brick with n_j closest to a value $\delta = \min((r_{\max} - r_{\text{opt}}), (r_{\text{opt}} - r_{\min}))$. Then brick B_j is reassigned to PE$_{\min}$. The algorithm terminates when δ becomes smaller than a given Δ or when at most N_B/z steps have been performed. If $\delta > b$ a group of bricks is reassigned rather than a single brick.

In the *hybrid load balancing approach*, the user specifies a percentage of bricks to be assigned dynamically and the set of bricks is divided into two subsets, one assigned statically and one assigned dynamically. One of the PEs, called the *dispatcher*, manages the dynamic subset. When a PE finishes processing the bricks in the static subset assigned to it, it requests additional load which is distributed by the dispatcher, on a first come first served basis. The dynamic load balance approach has a major difficulty due to the interaction with the virtual brick management system. If brick B_{j+1} is assigned to a PE different from the one which processed B_j, then there is a high probability that more brick faults occur than when B_j and B_{j+1} are processed by the same PE. Therefore this dynamic load approach increases the total number of brick faults.

6 Experimental Results

A program for the CDC Cyber 205, which implements a data parallel algorithm for computing the average electron density at points in the virus, was the basis for the present implementation of a control parallel algorithm for a distributed memory MIMD system on the Intel iPSC/860 and the Touchstone Delta.

The first version of the MIMD program used a shared virtual memory and a dynamic load balancing scheme and required less time than when executed sequentially on the CDC Cyber 205. An iteration of the electron density averaging for the structure determination of the tetragonal Canine Parvovirus (CPV) at 3Å resolution, requires about 12,000 seconds (of CPU time) on the CDC Cyber 205, about 32,000 seconds (CPU time) on an IBM RS 6000 model 550 workstation, and it runs in about 3,000 seconds (elapsed time) on the 16 node Intel iPSC/860 system at Purdue. At Caltech, the same program runs in about 864 seconds of elapsed time on a 64 node Intel iPSC/860 and in 550 seconds on a 128 node submesh of the Delta System. An iteration in the computation of the average electron

Number of PEs	1	2	4	8	16
Execution Time	21,160	10,990	6,069	3,842	3,090
Speedup	1.0	1.92	3.48	5.5	6.84

Table 2: Execution time (seconds) and speedup for an iPSC/860 with 16 PEs and one I/O node.

Number of PEs	1	2	8	16	32	64
Execution Time	23,364	11,855	3,473	1,977	1,323	864
Speedup	1.0	1.97	6.7	11.8	17.6	27

Table 3: Execution time (seconds) and speedup for an iPSC/860 with 64 PEs and 6 I/O nodes as the number of PEs varies.

density for the Coxsackie B_3 virus at 3Å resolution takes about 64,000 seconds CPU time on the CDC Cyber 205 and about 12,000 seconds (elapsed time) on the 16 node Intel iPSC/860 at Purdue. These examples show the advantage of using distributed memory MIMD systems for the determination of macromolecular structures.

An important characteristic of parallel computing on MIMD systems is the speedup and its relationship with the problem size and the number of PEs used. The problem size for applications described by the model in Section 3.1 is determined by the resolution (which results in a specific number of grid points, N) and by the number of grid points in the subset B. The measurements reported in this section all use the same set of data for the structure determination of the CPV at 3Å. This is a medium size problem, the number of grid points is $N = 130 \times 130 \times 400 \approx 7 \times 10^6$ and there are about 4×10^6 grid points in the virus, the B subset. The execution time on the CDC Cyber 205 is about 12,000 seconds (CPU time). The actual execution time and the speedup using the optimal load balancing factor are reported in Table 2 and Table 3 for two systems:

System 1: an iPSC/860 with 16 PEs and one I/O node,

System 2: an iPSC/860 with 64 PEs and six I/O nodes.

The speedup for System 2 is shown in Figure 3.

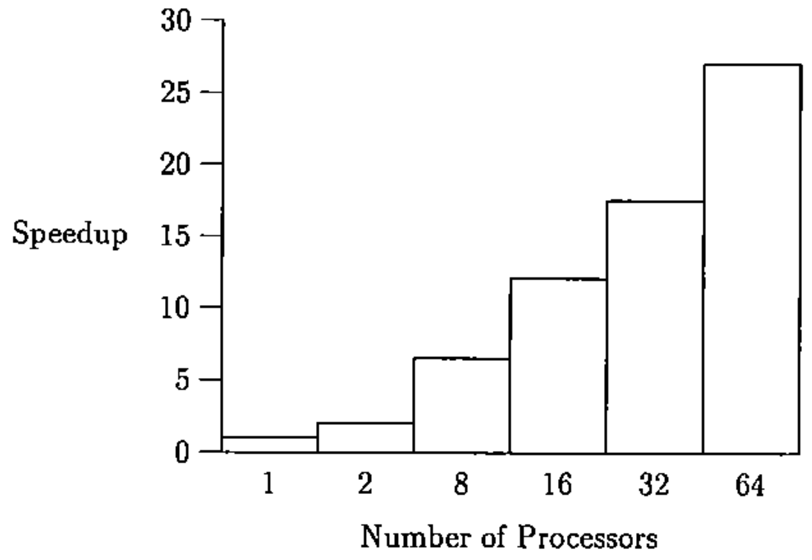

Figure 3: The speedup for an iPSC/860 as a function of the number of PEs for the CPV.

The I/O bandwidth can limit severely the performance and the previous measurements confirm this. The I/O bandwidth determines the data transfer rate to and from the external I/O devices. The two systems studied have different I/O bandwidths: System 1 had only one I/O node while System 2 had six I/O nodes. For example, in the 16 PE case, the execution time on System 1 is about 50% larger than on System 2: 3000 seconds versus 2000 seconds (elapsed time). This clearly suggests that the smaller is the I/O bandwidth, the larger is the execution time. The execution times with 1 and 2 PEs are larger on System 2, possibly due to contention with other jobs which run concurrently. No other jobs were running on System 1 when the experiments were done.

The effect of the I/O bandwidth upon the performance deserves further investigation. It is reasonable to expect that the virtual shared memory increases the data rates required by the application; the effects of the number of PEs and of the locality of reference upon the performance need to be studied. As expected, the larger is the number of PEs used for computation, the larger is the need for I/O bandwidth. The total number of brick faults increases when the number of PEs increases as the results reported in Table 4 indicate. Also, as predicted by the analysis in Section 4.2, the start-up time t_s increases with the number of PEs, because each PE has to fetch all the bricks in the working set of the first master brick assigned to it.

Other effects which deserve further investigation are related to the load balancing. The total processing time per brick depends upon the number of grid points in the brick which are also in the virus. Processing times per brick in the range of 0.001 to 30.0 seconds are observed, in the experiment which uses only one PE; with a single PE, there is no contention for I/O resources from other PEs.

Number of PEs	1	2	4	8	16	64
Brick faults	42,293	47,743	51,087	54,258	60,585	98,090

Table 4: The number of brick faults for 20% dynamic load.

Load balance factor	2%	10%	20%	30%	50%	70%	90%
Brick faults	54,138	57,024	60,585	68,095	76,629	87,138	101,330
Execution time (sec)	3,072	3,102	3,090	3,438	3,838	4,311	4,851

Table 5: The total number of brick faults and the execution time in seconds for System 1 and 16 PEs running.

Even for a specific brick, the processing time may vary widely depending upon the total number of PEs used, the I/O bandwidth used, and the mapping strategy used. For example, when only one PE is used, the processing time for one example brick is about 26 seconds and only one brick fault is generated. The processing time for this same brick increases to about 28 seconds for 2 PEs, 29 seconds for 4 PEs, 45 seconds for 8 PEs, and 291 seconds for 16 PEs. In the 16 PE case, this brick is the first one assigned to a PE and its entire working set of 408 bricks has to be fetched from VBA; this results in many brick faults. Table 5 and Table 6 illustrate the effect of the dynamic load balance factor upon the total number of brick faults and upon the total execution time.

The relationship between the mapping strategy and the shared virtual memory is examined next. Consider a hybrid load balancing strategy where a fraction of the total number of bricks is assigned statically and the remaining bricks are assigned dynamically. The fraction of the total number of bricks assigned dynamically is called the *dynamic load factor*. As one would expect, a higher dynamic load factor leads to a larger number of brick faults. For example in the 16 PE cases, with 90% dynamic load, the number of brick faults is nearly double that with the 2% dynamic load. Indeed, each PE processes an average of 175 bricks

Load balance factor	2%	10%	20%	25%	30%	35%	40%	45%	100%
Brick faults	89,792	95,310	98,090	100,447	101,101	102,961	105,408	106,009	114,811
Execution time	973	959	939	909	905	893	864	870	995

Table 6: The total number of brick faults and the execution time in seconds for System 2 and 64 PEs running.

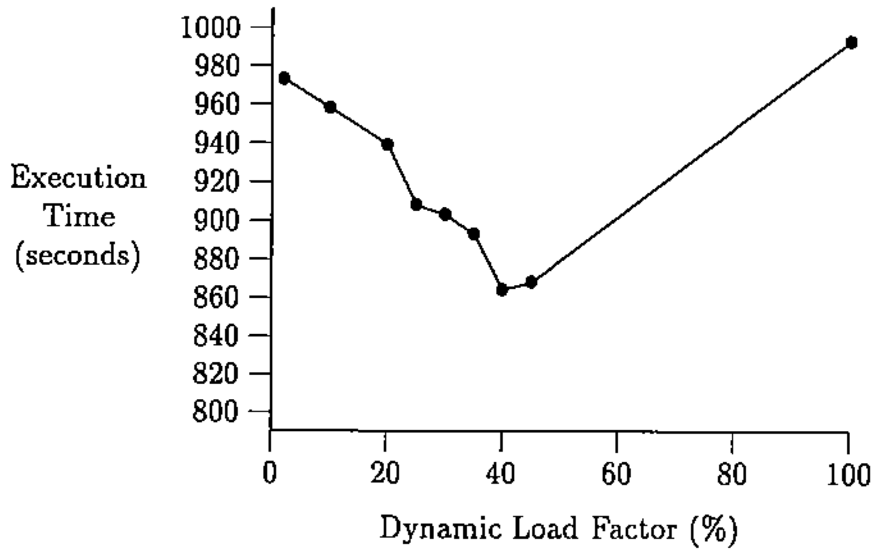

Figure 4: The effect of the dynamic load factor upon the execution time.

(N_B = 2806) and, due to the virtually random assignment of bricks to PEs, each time a PE starts processing a new master brick, only a few bricks in its working set are already in real storage. Due to the limited I/O bandwidth of System 1, the doubling of the number of brick faults increases the actual execution time by about 60% and seems to cancel out any beneficial effect of the dynamic load balancing.

In System 2, when 64 PEs were running the number of brick faults increases by about 25% when the dynamic load factor increases from 2% to 100%, yet the execution time is practically the same in these extreme cases. This suggests that other effects, for example *hot spot* contention (several PEs try to access the same brick at the same time), might play some role. The dynamic load balancing produces a 10% decrease in the total execution time. The effects of the dynamic load upon the execution time (Figure 4) and upon the number of brick faults (Figure 5) illustrate graphically the results for the 64 PE experiment.

Experiments with smaller problems indicate that when size(RBA) > size(VBA), the dynamic load balancing decreases the execution time by 10–20%. Indeed, in this case all bricks are eventually brought into the RBA of all PEs and no brick faults occur. Therefore the benefits of a dynamic load balancing are no longer canceled out by the limited I/O bandwidth.

Figures 6, 7, and 8 show the fraction of the total execution time used for computing, for I/O, and for waiting. The fractions are plotted for each of the PEs and for three different dynamic load factors: 0%, 20%, and 80% for a 16 processor run. The data were gathered using the Parallel Analysis Tool (PAT) provided by INTEL. Figure 6 shows that the amount of computation per node varies widely when there is no load balance strategy (0% dynamic load factor). There is a visible improvement when the dynamic load factor is 20% (see Figure

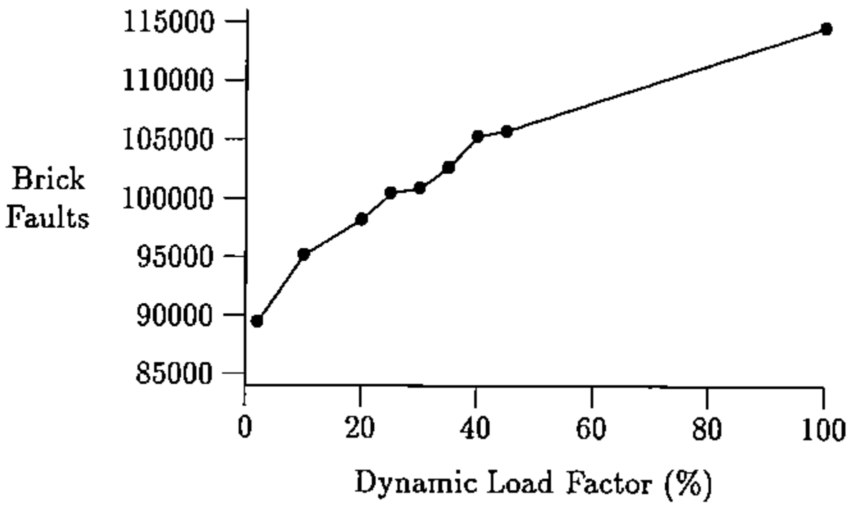

Figure 5: The effect of the dynamic load factor upon the number of brick faults.

7) but the overall processor utilization decreases because of the increased I/O activity. This trend continues as seen in Figure 8 (dynamic load factor 80%), all PEs have essentially equal computation loads but the computing time represents a smaller fraction of the total time due to an increase in the I/O time.

Ultimately the performance of the program depends upon

(a) the characteristics of the problem to be solved, namely the problem size (N_B, the number of bricks) and the grid geometry,

(b) the computing resources: number of PEs, the I/O bandwidth, the amount of memory of each PE, and

(c) the parameters of the algorithm. To obtain the best performance, an optimal selection of parameters like the brick size, the dynamic load factor, the frequency of updates of the brick reference counters, and so on, must be made.

The brick LRU replacement algorithm works well. In the experiments reported here, the brick reference counts are reset after the processing of every master brick. A less frequent reset, say after every k bricks, should be explored. Yet the expected number of brick faults per master brick processed is about 32 for the 64 PEs and 2% load balance case. One can expect up to 480 bricks in the working set for a given grid point when $M = 60$ and an 8 point interpolation scheme is used. A brick in these experiments has 4096 grid points, therefore the observed average brick working set size of 32 is considerably smaller than the maximum value and shows that the computation has a very good locality of reference, and the brick replacement algorithm works well. The current brick size is 8,192 bytes and experiments

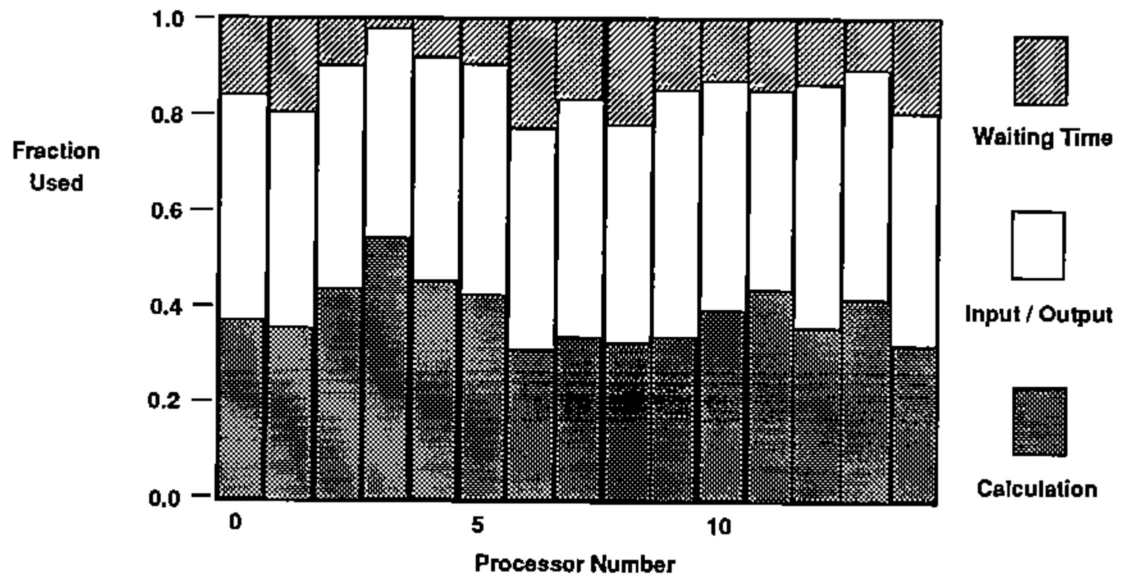

Figure 6: The effect of dynamic load balance on utilization: 0% load balance

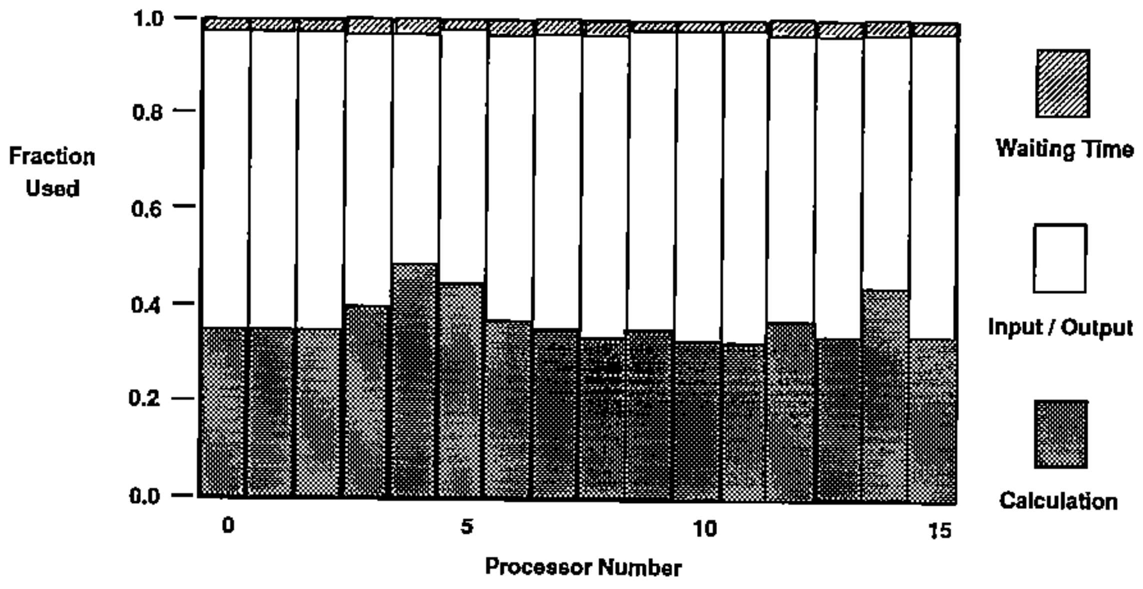

Figure 7: The effect of dynamic load balance on utilization: 20% load balance

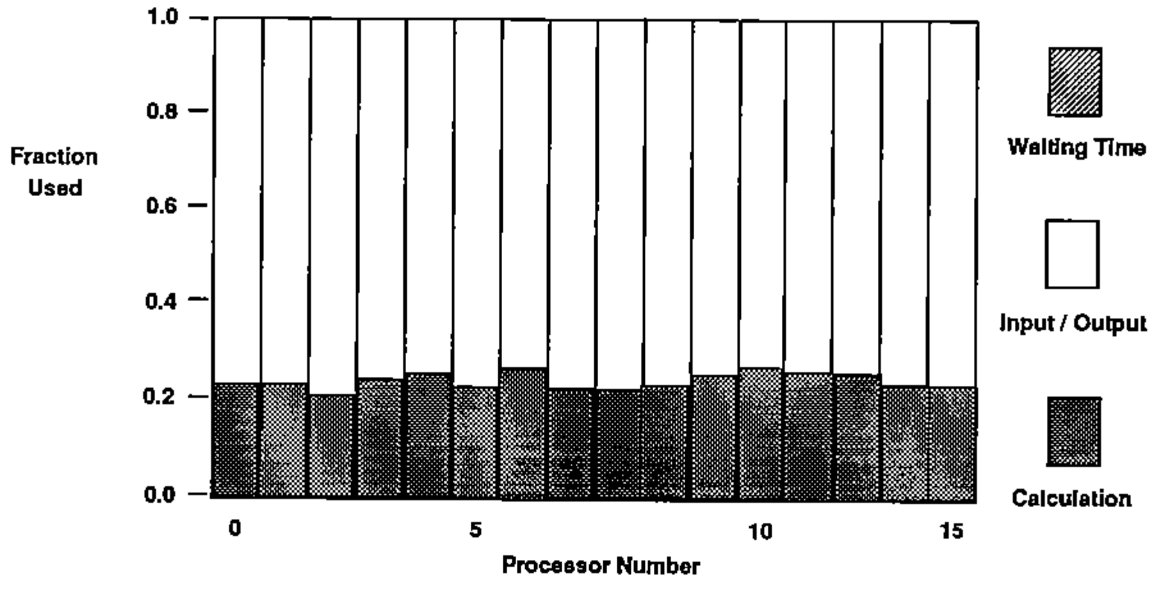

Figure 8: The effect of dynamic load balance on utilization: 80% load balance

to determine if a different brick size, (e.g., 2048 grid points or 4096 bytes) leads to better performance are planned.

A careful tuning of an application is always necessary. For example, eliminating messages sent by every PE upon completion of the computation for a master brick reduces the execution time by about 3%.

7 Conclusions

For many applications, distributed memory MIMD systems provide an increasingly attractive alternative to more traditional vector supercomputing. For example, the computation of the averaged electron density ran 15 times faster on a 64 node Intel iPSC/860 hypercube than on a CDC Cyber 205. Yet the development of an MIMD application is considerably more difficult than that of a SIMD one. In addition to the inherent difficulties of handling several threads of control in parallel, the application presented in this paper relies on a user controlled shared virtual memory. This expands the user's address space beyond the limits imposed by the memory available in each PE of a distributed memory MIMD system in order to handle large problems.

It is reasonable to expect that future generations of MIMD systems will provide such virtual memory, but the I/O bandwidth of such systems must be increased accordingly. Our experiments show that low I/O bandwidth can limit severely the performance of an MIMD system with virtual memory even when the application exhibits a fairly good locality of reference. In the past, the processor bandwidth has increased at a faster pace than the I/O and communication bandwidths. It is plausible that in order to support efficiently virtual memory, future systems will need a dedicated I/O network for linking the computational and I/O nodes.

The tuning of virtual memory applications is quite difficult. The use of a dynamic load balance mechanism did not produce the theoretical optimal results because of the diminished locality of reference. Assigning bricks randomly to the PEs increases the number of brick faults. Load balancing schemes which simultaneously partition the data well and at the same time preserve the locality of reference are yet to be found.

Several improvements to the implementation described in this paper are under consideration. First, one could implement a shared memory rather than a shared virtual memory. In this case the data are distributed to the P processing elements of a partition rather than being kept on an external disk device such as the Concurrent File System. A mechanism must be in place to keep track of the locations of bricks and to fetch them from another PE when needed. The mechanism for implementing the shared memory could be based on [Will 91]. This method requires smaller I/O bandwidth than the virtual shared memory and it is most suitable for the case when a large number of PEs are used. In this case the size of the VBA stored in each PE is small and each PE can still dedicate to the RBA enough space to hold the largest brick working set.

A second improvement is to minimize the number of brick faults and pre-fetch the bricks in the working set of a master brick rather than fetch them when needed, as in the current implementation. Bricks can be pre-fetched by using the known symmetry in the virus. This would reduce the time a PE is blocked waiting for a brick, but would not decrease the I/O bandwidth requirements.

It is very likely that in the immediate future, even modest research establishments will have networks consisting of tens of workstations. The workstations will use high performance engines like the Alpha microprocessor, recently announced by Digital Corporation and rated at about 150 Mflops peak. A large fraction of the computing cycles available in such a network will be unused and the current MIMD implementation of the electron density averaging program can be moved effectively to a network of workstations sharing a file server. The file server would host the VBA area and each workstation would act as a node of the MIMD system. Indeed, this portability is a major advantage of the programming model presented here and this approach might provide a low cost alternative to buying an expensive MIMD system.

Acknowledgments

The Intel iPSC/860 and the Touchstone Delta System of the Concurrent Supercomputing Consortium (CSC) at Caltech were used. The authors thank the technical staff of CSC and the Purdue University Computing Center for the help they kindly supplied.

References

[Fox 87] Fox, G., et. al., *Solving Problems on Concurrent Processors,* Prentice Hall, Englewood Cliffs, NJ, (1988).

[Intel 91] *** Touchstone Delta System. User's Guide ***

[Ross 72] Rossmann, M. G., *The Molecular Replacement Method*, Gordon and Breach.

[Ross 92] Rossmann, M. G., R. McKenna, L. Tong, D. Xia, J-B. Dai, H. Wu, H-K. Choi, and R. E. Lynch, "Molecular replacement real-space averaging", *J. Appl. Cryst.* 25 (1992), in press.

[Will 91] Williams, Roy, *Associations: A tool for parallel computing with meshes*, CCSF-9-91.