On the Parallel Elliptic Single/Multigrid Solutions about Aligned and Nonaligned Bodies Using the Virtual Machine for Multiprocessors

A. AVERBUCH, E. GABBER, S. ITZIKOWITZ, AND B. SHOHAM

School of Mathematical Sciences, Department of Computer Science, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel

ABSTRACT

Parallel elliptic single/multigrid solutions around an aligned and nonaligned body are presented and implemented on two multi-user and single-user shared memory multiprocessors (Sequent Symmetry and MOS) and on a distributed memory multiprocessor (a Transputer network). Our parallel implementation uses the Virtual Machine for Multi-Processors (VMMP), a software package that provides a coherent set of services for explicitly parallel application programs running on diverse multiple instruction multiple data (MIMD) multiprocessors, both shared memory and message passing. VMMP is intended to simplify parallel program writing and to promote portable and efficient programming. Furthermore, it ensures high portability of application programs by implementing the same services on all target multiprocessors. The performance of our algorithm is investigated in detail. It is seen to fit well the above architectures when the number of processors is less than the maximal number of grid points along the axes. In general, the efficiency in the nonaligned case is higher than in the aligned case. Alignment overhead is observed to be up to 200% in the shared-memory case and up to 65% in the message-passing case. We have demonstrated that when using VMMP, the portability of the algorithms is straightforward and efficient. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

The introduction of parallel computer architectures in the last years has challenged the existing serial methods such as finite differences, finite elements, and multigrid solutions of PDEs. See, for example, [1] through [7] for the implementation of the multigrid methods on message-passing architectures.

Physical problems often require the solution of the corresponding Partial Differential Equations (PDE) around bodies of various shapes. It appears that in such cases there is no simple method to map the body surface into the grid points. There are some methods that generate grid points in such a way that the body surface is aligned with the grid, by using local grids [8, 9]. Another method is based on using Cartesian grids [8, 9] such that the body surface is not aligned with the grid, but we have to take into consideration its existence within the grid. This method eliminates

Received March 1992
Revised July 1993

© 1994 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 3, pp. 13–32 (1994)
CCC 1058-9244/94/010013-20

some problems that are involved in generating grid points for multigrid solution for flow problems over shapes with complex geometry. However, it requires using dummy points (points that are on the body boundary or in its interior) and raises the problem of how to use the corresponding boundary conditions. A potential solution to this problem is to use CDC CYBER 860 [8, 9].

In this article, as a model problem, we investigate the parallel implementation of Poisson single/multigrid solution on two shared-memory and shared bus multiprocessors (multi-user and single-user machines), as well as on a distributed memory multiple instruction multiple data (MIMD) multiprocessor using Virtual Machine for MultiProcessors (VMMP) [10, 11]. For the multigrid solution [8, 9, 12], we apply a set of grids with a decreasing order of refinement, all approximating the same domain to be considered. We start with the finest grid and apply the weighted Jacobi relaxation algorithm. When relaxation begins to stall, we move to a coarser grid, on which relaxation becomes more efficient. After reaching a coarser grid with a satisfactory rate of convergence, we return to the finest grid and correct the solution approximation accordingly.

Unlike previous investigations, we consider two distinct cases: (1) when the body boundary is aligned with the grid and (2) when the body is not aligned with the grid so that dummy points are introduced. We consider both shared-memory and message-passing architectures and compare their performance.

This article is organized in the following way: In Section 2 we present a brief background concerning the weighted Jacobi relaxation method, the multigrid method, and the VMMP software package. In Section 3 we introduce our model problem and its specific parallel implementation. Numerical results are presented in Section 4 and processing time analysis is presented in Section 5. Finally, Section 6 summarizes our results.

2 BACKGROUND

In this section we briefly present some of the methods and software tools used in the solution of our model problem. In particular, we present the weighted Jacobi relaxation method, the multigrid method, and the VMMP package.

2.1 The Weighted Jacobi Relaxation Method

Let $A\vec{u} = \vec{f}$ be a system of linear algebraic equations representing a finite difference solution that approximates an elliptic partial differential equation where A is nonsingular. Let $D = diagA$, $B = D^{-1}(D - A)$ and $\vec{c} = D^{-1}\vec{f}$. Then we can replace the linear system by the equivalent system

$$\vec{u} = B\vec{u} + \vec{c}. \tag{1}$$

In the weighted Jacobi method [13], which will be considered here, a real parameter w is introduced as follows:

$$\vec{u}^{(n+1)} = B_w\vec{u}^{(n)} + w\vec{c} \tag{2}$$

where

$$B_w = wB + (1 - w)I \tag{3}$$

For the case in which the body surface is aligned with the grid, the algorithm consists of using in each iteration the value of real grid points that are participating in the grid relaxation. The boundary conditions exist only on the grid boundary. For the case where the body surface is not aligned with the grid, one may need to know the solution at points that are in the body interior or on its surface. These points are called dummy points. For example, consider the problem of Figure 1 in which value at the point C in a given iteration is computed from the values of C_1, C_2, C_3, C_4. However, the value at A has to be com-

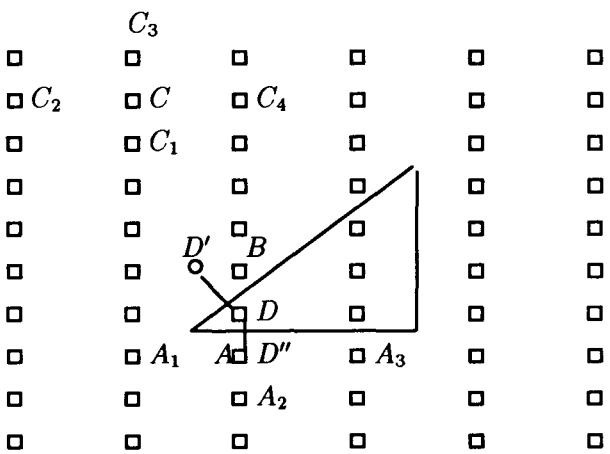


FIGURE 1 Dummy points in the nonaligned case.

puted from the values of A_1, A_2, A_3, and D, when D is located inside the body. In such a case we proceed as follows: We first consider the point that is the mirror image of D along the tangent line to the body boundary, namely the point D''. The value at the point D is then replaced by a weighted average of the values in the neighborhood of D''. Similarly, in order to compute the value at B, the value at D will be replaced by a weighted average of the values of the neighborhood in D'. Note that the value of the dummy point D is different in the computation of A and B because D is reflected to different points in the grid. Body boundary points are replaced by a weighted average of their neighbors values. Note that the value of inner points is not computed.

2.2 The Multigrid Method

Consider a partial differential equation and let $A^h u^h = f^h$ be the corresponding linear system to be solved iteratively on some domain G_h with grid spacing h. In the multigrid method we consider a set of grids with a decreasing order of refinement, G_h being the finest grid. We start with the finest grid and apply some relaxation algorithm. When relaxation begins to stall, indicating the predominance of smooth (low frequency) error modes, we move to a coarser grid in which these smooth error modes appear more oscillatory (corresponding to high frequency modes) and relaxation becomes more efficient. After reaching a coarser grid with a satisfactory rate of convergence, we return to the finest grid and correct the solution approximation accordingly [13].

Let I_{2h}^h be the interpolation operator from the coarser grid G_{2h} to the fine grid G_h and let I_h^{2h} be the one from G_h to G_{2h}. In general, the order of coarse to fine interpolation should be no less than the order of the differential equation to be considered. The most popular multigrid scheme is the μ cycle multigrid scheme. It solves the system $u^h = M\mu^h(u^h, f^h, A^h)$ recursively as follows:

1. Relax γ_1 times on $A^h u^h = f^h$ (according to some relaxation algorithm).
2. If the grid is the coarsest grid then go to 4. Otherwise interpolate

$$f^{2h} = I_h^{2h}(r^h)$$

where

$$r^h = f^h - A^h u^h$$

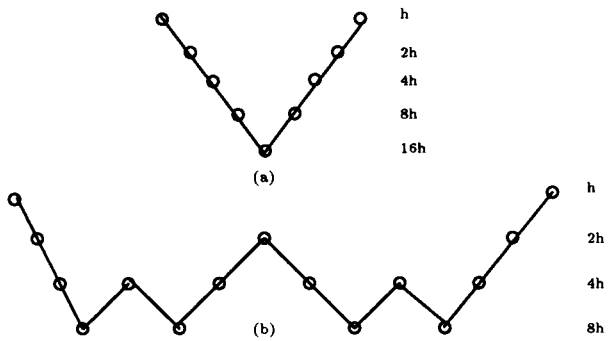


FIGURE 2 Schedule of grids (a) V-cycle and (b) W-cycle all on four levels.

and apply a recursive call on a coarser grid

$$u^{2h} = 0$$

$$u^{2h} = M\mu^{2h}(u^{2h}, f^{2h}, A^{2h}) \ \mu \text{ times.}$$

3. Interpolate and correct the fine grid solution

$$u^h = u^h + I_{2h}^h u^{2h}$$

4. Relax γ_2 times on $A^h u^h = f^h$.

Hence, for instance $\mu = 1$ corresponds to the V-cycle algorithm, and $u = 2$ corresponds to the W-cycle algorithm (Fig. 2).

2.3 VMMP

VMMP [10, 11] is a software package developed at the computer science department of the Tel-Aviv University. VMMP provides a coherent set of services for explicitly parallel application programs running on diverse MIMD multiprocessors, both shared memory and message passing. It is intended to simplify parallel program writing and to promote portable and efficient programming. Furthermore, it ensures high portability of application programs by implementing the same services on all target multiprocessors.

VMMP provides high level services, such as the creation of cooperating set of processes (called *crowd computations* in VMMP), creation of a tree of lightweight processes (called *tree computations*), data distribution and collection, barrier synchronization, topology independent message passing, shared objects memory, etc. All of these services may be implemented efficiently on a variety of target machines. These high level services ensure that their implementation details can be

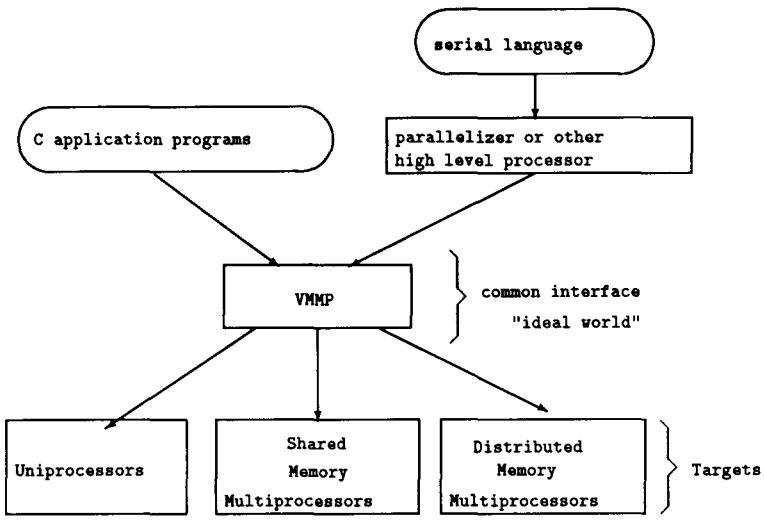


FIGURE 3 VMMP usage.

hidden effectively from the user. VMMP does not provide low level services, such as semaphores, because they are difficult to implement efficiently on all target machines (e.g., on a message-passing multiprocessor).

VMMP may be used directly by the application programmer as an abstract parallel machine to aid portability without sacrificing efficiency. VMMP may also be used as a common intermediate language between high level programming tools, such as parallelizers and specification language processors, and the target machines. In particular, VMMP is used as the intermediate language in P^3C (Portable Parallelizing Pascal Compiler) [14] which translates serial programs in Pascal or Fortran to explicitly parallel code for a variety of target multiprocessors. VMMP usage is illustrated in Figure 3.

This article includes only a short description of VMMP services and capabilities related to our multigrid problem. Full details and other types of problems can be found in Gabber [10, 11].

VMMP Services

VMMP supports the communication and synchronization requirements of parallel algorithms, and especially the requirements of tree and crowd computations. It provides a virtual tree machine to support tree computations and crowd services to support crowd computations. Shared objects memory provides an asynchronous and flexible communication mechanism for all computations. VMMP also supports several data distribution strategies using shared objects or by special crowd operations. A crowd may create a tree computation and vice versa. Figure 4 illustrates the relationship among VMMP services.

Tree Computations. Tree computations are a general computation strategy, which solve a problem by breaking it into several simpler subproblems, which are solved recursively. The solution process looks like a tree of computations, in which the data flows down from the root into the leaves and solutions flow up towards the root. Some ex-

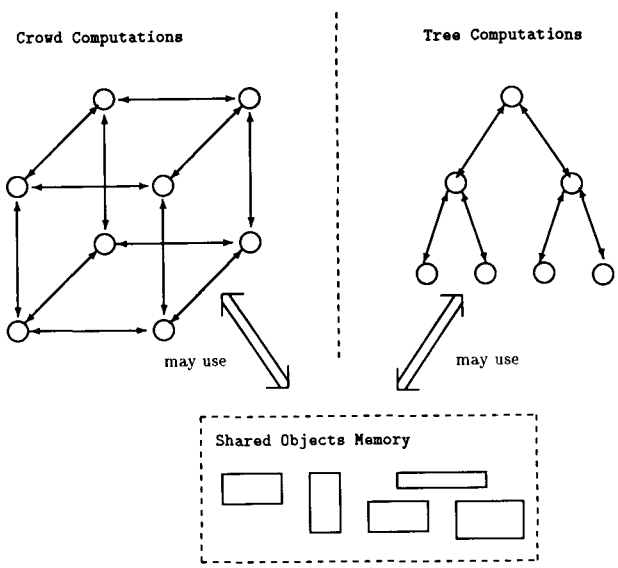


FIGURE 4 VMMP services.

amples of tree computations are divide and conquer algorithms, game tree search, and combinatorial algorithms.

Crowd Computations. Many parallel algorithms are defined in terms of a set of cooperating processes. Each of these processes is an independent computational entity, which communicates and synchronizes with its peers. The processes communicate and synchronize mainly by message passing, which may be either blocking or nonblocking.

The communication structure of crowd computations normally follows some regular graph, such as a ring, mesh, torus, hypercube, or tree. Some computations change their communication structure during the computation and some require an unlimited communication between any pair of processes.

Message passing forces an implicit synchronization between the sending and the receiving processes, because the receiver normally waits for a message from the sender. Blocking send and receive operations imply a much stricter synchronization between the sender and the receiver.

Shared Objects Memory. VMMP implements a shared objects memory with the same semantics on all target multiprocessors. Shared objects may be regarded as virtually shared-memory areas, which contain any type of data, and can be accessed only by secure operations. Shared objects can be created, deleted, passed by reference, read, and written.

The shared objects are intended to distribute data, collect results, and collect global results by some associative and commutative operations. Other uses of the shared objects, such as random updates to parts of a large object, are discouraged, because the implementation of these operations increase overhead on a message-passing multiprocessor.

Concurrent writes to the same shared object are permitted. The value of an object is the last value that has arrived at the master copy of the object. This value is then broadcast to all other copies of the object (if any). A read following a write to the same object will return either the value written or a more recent value. It is possible that two distinct processors reading the same object at the same time will access different versions of the object value. This definition is necessary to guarantee consistent behavior also on message-passing multiprocessors. A safe way to update shared objects is to apply an associative function, which guarantees the correct result regardless of the application order.

The user must manipulate the shared objects only through VMMP services, which guarantee the correctness of the operations. The program must call VMMP explicitly after each change to the local copy of a shared object in order to broadcast the new value to other processors.

VMMP Support for Grid Computations

VMMP provides several high level services that support grid algorithms. These services allow a crowd process to access a slice of the grid and exchange overlapping boundary elements with the neighboring processes.

VMMP supports several grid distribution strategies to ensure low communication overhead and load balancing. They are illustrated in Figure 5. The `Vgrid_part` service computes the boundaries of a slice of the grid that is assigned to the process. The program can use a different distribution strategy by changing a single parameter to the `Vgrid_part` routine. Additional distribution strategies can be programmed explicitly by a combination of data access services.

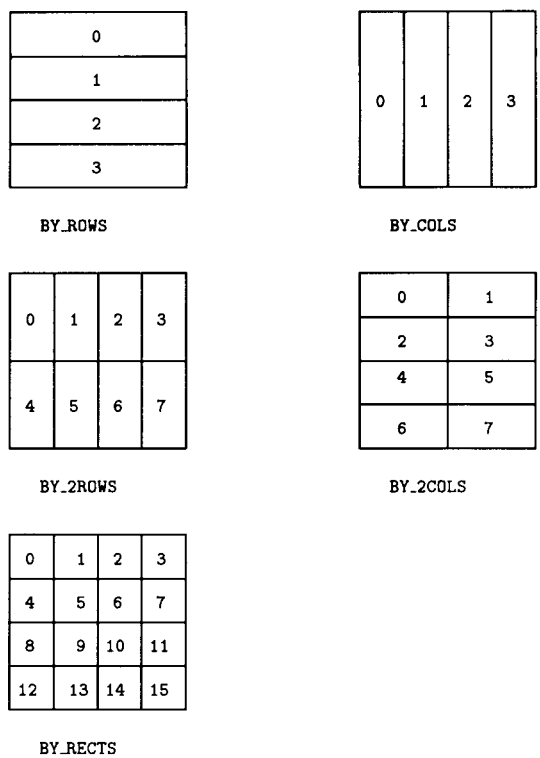


FIGURE 5 Supported grid distributed strategies.

The following C code fragment illustrates grid update operations using VMMP services. It is a part of a program for a solution of a Laplace PDE by relaxation. The multigrid algorithm is implemented by a similar code. The VMMP services used in this code fragment will be explained in the following paragraphs.

```
#define BOUNDARY      1          /* Overlapping boundary width */
/* Body of the crowd process executed on each processor */
void solve(in,out,ids)
ID ids[2];              /* shared objects descriptors */

{
    RECT r;             /* current rectangle */
    double a[N] [N]; /* the grid */
    double err;         /* current error */
    /* calculate our rectangle slice */
    Vgrid_part(BY_ROWS, N, N, BOUNDARY, &r);
    /* Compute the boundary of the processor's slice: */
    low_row = r.first_row+BOUNDARY;
    high_row = r.last_row-BOUNDARY;
    low_col = r.first_col+BOUNDARY;
    high_col = r.last_col-BOUNDARY;
    /* Read processor's slice of the grid, including overlapping elements
       with neighbors */
    Vget_rect(ids[1], a, r, sizeof(double));
    do {
        err = 0;
        /* Relaxation step - compute values of interior grid points */
        for(i = low_row; i <= high_row; i++)
            for(j = low_col; j <= high_col; j++){
                a[i][j] = .....
                /* Compute local error */
                error = max(error, ...);
            }
        /* Exchange overlapping elements with neighbors */
        Vgrid_update(a, r, sizeof(double), 0);
        /* Compute global error */
        Vcombine(&err, fmax_d, 0, sizeof(err));
    } while(err > EPS);
    ...
}
```

The `Vget_rect` and `Vset_rect` are used to read and write a slice of a shared object containing the grid. The slice boundaries were computed previously by the `Vgrid_part` according to the distribution strategy.

The `Vgrid_update` routine is used to exchange the overlapping boundary elements of the grid between neighboring processes. The `Vcombine` routine is used to compute an associative and commutative operation, such as maximum, on values provided by all processes.

VMMP Implementation

VMMP has been implemented on the following multiprocessors:

1. Sequent Symmetry: a 26-processor Sequent Symmetry running the DYNIX operating system.
2. MMX: an experimental shared bus multiprocessor containing four processors running a simple run time executive [15].
3. ACE: an experimental multiprocessor developed at IBM T. J. Watson Research Center running the MACH operating system [16].
4. MOS: a shared-memory multiprocessor containing eight NS32532 processors running MOSIX, a distributed UNIX [17].
5. **Transputers:** a distributed memory multi-

processor consisting of a network of eight T800 transputers running the HELIOS distributed operating system [18].

VMMP simulators are available on SUN3, SUN4, IBM PC/RT, and VME532 serial computers. The same application program will run unchanged on all target machines after a recompilation.

VMMP has been used in the implementation of many other problems, including a parallel generalized in frequency FFT (GFFT) [19], parallel solution of underwater acoustic implicit finite difference (IFD) schemes [20], and a portable and parallel version of the basic linear algebra subroutines (BLAS-3) [21]. VMMP was also used as the intermediate language of the portable parallelizing pascal compiler (P^3C) [14], which translates serial application programs in Pascal into portable and efficient explicitly parallel code.

3 THE MODEL PROBLEM AND ITS PARALLEL IMPLEMENTATION

3.1 The Model Problem and Its Solution Scheme

As our model problem we consider the Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = F(x, y) \tag{4}$$

on a rectangular domain G, which without loss of generality is taken to be the unit square ($0 \le x \le 1$, $0 \le y \le 1$) with Dirichlet boundary condition

$$u(x, y) = g(x, y) \text{ on } \partial G. \tag{5}$$

As indicated earlier, when the body is aligned with the grid, its boundary coincides with the grid boundaries. In the case of an unaligned body we consider a rectangular body as illustrated in Figure 6.

Note that although we consider a specific elliptic problem, our analysis to follow provides qualitative results for the more general cases and may be easily modified to fit other boundary-value problems and more complicated domains.

We use a grid G_h where $h = \dfrac{1}{N+1}$ is the uniform grid spacing along the x and y axes, where N is the number of interior grid points.

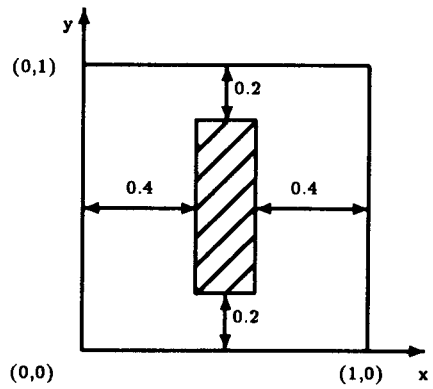


FIGURE 6 The body shape in the aligned and unaligned case.

Approximating the partial derivatives by the corresponding second order central difference equation we obtain

$$\frac{V_{i+1,j} - 2V_{i,j} + V_{i-1,j}}{h^2} + \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{h^2} = f_{i,j} \tag{6}$$

where

$$f_{i,j} = F(x_i, y_j),\ V_{i,j} = u(x_i, y_j),$$
$$x_i = ih (1 \le i \le N - 1),$$
$$y_j = jh (1 \le j \le N - 1),$$

and

$$V_{i,j} = g(x_i, y_j) \text{ on } \partial G (i = 0, N;\ 1 \le j \le N - 1),\ (j = 0, N;\ 0 \le i \le N). \tag{7}$$

Truncation error is of $O(h^2)$.

The weighted Jacobi method applied to Poisson equation becomes

$$V_{ij}^{(n+1)} = wV_{ij}^{(n)} + (1 - w)(\tfrac{1}{4}(V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)} - h^2 f_{ij})). \tag{8}$$

In our analysis to follow, we consider general $F(x, y)$, $g(x, y)$ and w. However, for the sake of simplicity of intermediate convergence tests, in our practical implementation we take $F(x, y) = g(x, y) = 0$. In this case, the exact solution is identically zero and the numerical results are actually the corresponding errors. The convergence parameter is taken as $w = \frac{2}{3}$. In the case where the body is not aligned with the grid, the values $V_{i,j}^{(n)}$ at dummy and boundary points are evaluated as dis-

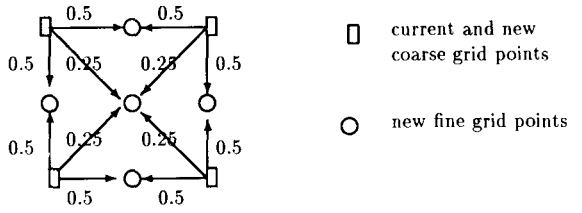


FIGURE 7 Coarse-to-fine interpolation; numbers indicate the corresponding weights.

cussed in Section 2.1. Note that because we consider a rectangular body, only reflections like D'' in Figure 1 (corresponding to the point A) are to be considered. In our implementation, we simplify the average to be taken such that the value of such reflection point is taken as the single value of the real grid point with which it coincides. For instance, when evaluating the value of the point A in Figure 1, the value of D is replaced by the value of its reflection D'', which in turn is replaced by the value of A.

The corresponding interpolation and restriction operators that are implemented here are as described in detail in Hackbusch [13] (pp. 60, 65). If the grid point in G_h coincides with a grid point of G_{2h} in the interpolation procedure, then the value is unchanged. If, however, the new fine grid point of G_h is along a partition line of G_{2h} then its value is taken as the symmetric average of its two neighbors along this line. Otherwise, the value is taken as the symmetric average of its four neighbors at the corners of the appropriate square (Fig. 7).

In the restriction procedure, the weights of neighbor points of G_h along a partition line of G_{2h} is twice as much as that of the neighbors at the corners of the corresponding square. The weight of the point itself is four times as much (Fig. 8).

We use the multigrid V-cycle ($\mu = 1$) as discussed in Section 2.2 with $\gamma_1 = \gamma_2 = 100$. Note that usually the main advantage of the multigrid methods is the fact that the needed number of

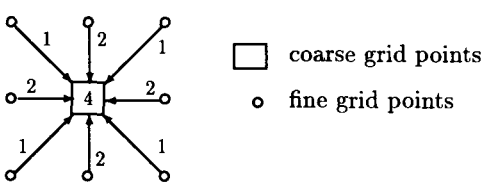


FIGURE 8 Fine-to-coarse grid interpolation; the corresponding weights are the shown numbers divided by 16.

iterations (which is determined by the problem residual) is relatively very small as compared with the single grid case. However, unlike other articles considering parallelization of multigrid solutions, the main purpose of our article here is to investigate parallelization of multigrid solutions of a model problem around aligned and nonaligned body. Therefore, as a first step, we have chosen fixed and relatively large values for γ_1 and γ_2 (for $N > 1$). The case where the number of iterations is determined by the residual problem in the aligned and nonaligned case involves a great deal of load balancing and is under current investigation. The finest grid spacing is $h = \dfrac{1}{N+1}$ and the coarsest grid spacing is $h = \frac{1}{2}$.

3.2 Domain Decomposition

The performance algorithms implemented on shared-memory and shared bus architectures has been of concern, as compared with the case of message-passing architecture, due to the need for global memory access. This access is done through a bus that is much slower than the local memory bus. Multigrid algorithms implement in each level a relaxation algorithm and then a fine-to-coarse grid, or coarse-to-fine grid, data transfer. The locality of multigrid computations enables having it implemented efficiently on both message-passing as on shared bus and shared-memory machines, because the computation of local data enables avoiding frequent use of the global memory. The communication overhead is minimized and there is less traffic on the bus so that less synchronization is required. On message-passing architectures, the algorithm and the problem are scalable. In other words, the performance stays the same if we increase the number of processors, which means that we have linear scalability. However, for shared-memory architecture, the availability of more processors will enhance the level of parallelism on one hand, but will increase synchronization and communication overhead on the other hand. Therefore, on shared bus architectures, there is no linear scalability.

Our parallel implementation of a grid relaxation is based on grid partitioning; the grid is divided into several subgrids each of which is assigned to a single processor. The partitioning in our case is done by splitting G into p horizontal strips of equal areas:

$$G = G^1 \cup G^2 \cup \cdots \cup G^p \tag{9}$$

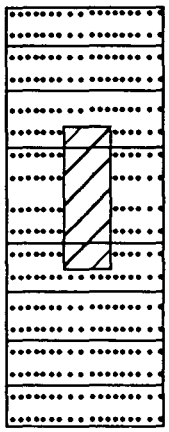


FIGURE 9 Domain partition for $N = 18$ and $p = 9$ (internal rectangle resembles the body shape).

where

$$G^l = \left\{(x_i, y_j) \in \mathrm{G} \mid 1 \le i \le N, \frac{(l-1)N}{p} \le j < l\frac{N}{p}\right\}, \tag{10}$$

p is the number of processors and N is the number of interior grid points along each axis. This is illustrated in Figure 9. If the body shape causes a significant difference in the number of grid points between the strips, the grid will be partitioned into nonequal size strips such that each contains an equal number of nondummy grid points. Note that this partition is different from that of Taylor and Markel [4], who consider only the aligned case and use a partition into square elements.

The relaxation step is performed on each subgrid independently. However, calculations of values at interior subgrid points cannot be done until values from neighboring subgrids are received. For this reason, and in order to avoid frequent access to the global memory, we use overlap areas that will include the needed values from neighbors' subgrids. Data from these overlap areas are transmitted and stored at the beginning of a relaxation step, simultaneously for all subgrids boundaries, and before the calculations of the subgrid interior points are performed. For example, consider a relaxation with two processors. The domain is divided into two subdomains, such that processor 1 is assigned the upper subdomain, whereas processor 0 is assigned the lower subdomain (see Fig. 10a). At the beginning of the relaxation, and before the evaluation of subgrid interior points, all values of the upper subdomain last row are transmitted simultaneously and stored at the

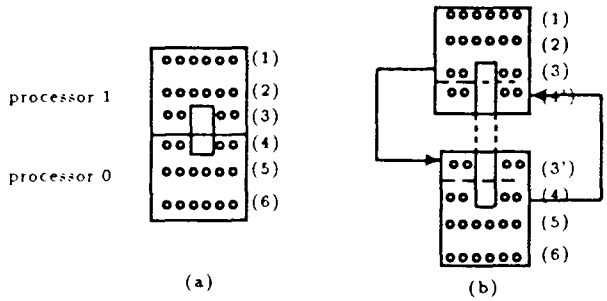


FIGURE 10 Overlap areas for 6 × 6 grid: (a) grid partitioning, (b) overlap data stored in processor local memory; internal rectangle resembles the body shape.

local memory of processor 0, in front of values of the lower subdomain first row (see Fig. 10b). Similarly, all values of the lower subdomain first row are transmitted simultaneously and stored in the local memory of processor 1, following the values of its last row. If dummy points exist, then the corresponding values are transmitted among neighbors in the same manner.

The same grid partitioning method is applied at all multigrid levels, and data transfers from fine grid to coarse grid (see Fig. 11) and vice versa are performed as described earlier in Section 3.1.

Note that the number of grid points per processor in the coarser grid is decreasing and thereby causing a relatively high cost of data transmission. Therefore the number of processors to be utilized in each step of the multigrid algorithm should depend strongly on the size of the grid. Because moving one step down in the V-cycle reduces the number of grid points by a factor of 4, at some step it would obviously be more economic to utilize only some of the available processors. Numer-

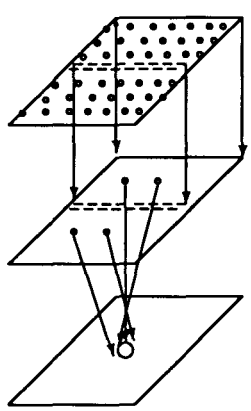


FIGURE 11 Transfer of data from fine grid level to a coarse grid level.

ical tests for our problem have shown that $f(p, m)$, the optimal number of utilized processors, is given by:

$$f(p, m) = \begin{cases} p & m > 2p \\ \left\lfloor \frac{m+1}{2} \right\rfloor & m \leq 2p \end{cases} \quad (11)$$

where p is the number of processors, m is the number of grid points along each axis, and "$\lfloor q \rfloor$" denotes the largest integer which is smaller than q. This way, with the overlapping rows and unless $N = 1$, each processor is assigned at least two rows. For example, in a multiprocessor with 12 processors we will use only 4 of them on a grid of 8×8 points.

Our partitioning has the following advantages:

1. Almost identical load for all processors
2. Reduced communication overhead
3. Grid partitioning and data transmission are kept simple
4. Simple passage from fine-to-coarse grid and vice versa

In the following, a schematic description of our multigrid implementation using the VMMP is presented.

```
Multigrid Program:
/* Init */
Vprocessors() /* get number of processors */
Vdef() /* define multigrid and dummy points structure */
Vcrowd( init ) /* create crowd of processes-each initialize part of the grid */
Vwait() /* wait for crowd computation to finish */
Vget() /* get crowd computation results */

/* Down Relaxation (relaxation and transformation from fine to course grid) */
For grid = finest grid to the courser grid
     Vdef() /* define grid and dummy points structure */
     Vcrowd( relax ) /* create crowd of processes-each relax on part of the grid */
     Vwait()
     Vget()
     Vcrowd( residual ) /* create crowd of processes each compute residual on part of
                           the grid */
     Vwait()
     Vget()
     Vcrowd( full_weight ) /* create crowd of processes-each transfer part of the
                              grid into part of the courser grid using full
                              weighted algorithm */
     Vwait()
     Vget()
Endfor

/* Up Relaxation (relaxation and transformation from course to fine grid) */
For grid = courser grid to the finest grid
     Vdef() /* define grid and dummy points structure */
     Vcrowd( relax ) /* create crowd of processes-each relax on part of the grid */
     Vwait()
     Vget()
     Vcrowd( interpolation ) /* create crowd of processes-each transfer part of the
                                grid into part of the finest grid using interpolation
                                algorithm */
     Vwait()
     Vget()
Endfor

End main program
```

```
   Relaxation function:
/* get local part of grid and dummy points */
Vcrowd_part() /* compute local part */
Vget_slice() /* get local local part of grid and dummy points structure */

/* Relaxation */
Repeat N times
      Compute relaxation result on local grid part
      Vsend( ) /* send local grid boundary values to the adjacent processor */
      Vrecv( ) /* receive local grid boundary values from the adjacent processor */
Endrepeat

Vset_slice() /* prepare result of local grid relaxation for Vget operation */
End relaxation function
```

4 NUMERICAL RESULTS AND PERFORMANCE ANALYSIS

The same parallel single grid relaxation and multigrid algorithms have been implemented on MOS [17], a single user-shared memory machine, on a Sequent Symmetry, a multi-user multiprocessor shared-memory machine, and on a distributed memory (a Transputer network) multiprocessor by using the VMMP software package.

4.1 MOS Performance

This section describes the implementation of the algorithms on the MOS single-user shared-memory multiprocessor. This multiprocessor has eight DB532 processor cards connected to shared memory via a VME bus. Each processor card contains a 25 MHz NS32532 CPU and FPU, 64 KB of write through cache, and 4 MB of local memory. The on-board cache minimizes global memory accesses. The multiprocessor runs the MOS [17] operating system, which is a distributed version of UNIX.

Figure 12 illustrates the MOS multiprocessor structure.

For the reference purpose of parallel speedup measurements, a serial program was executed on a single NS32532 processor, without the overhead of system calls to VMMP and of parallelization. This serial algorithm is the fastest we have been able to derive. The results of our implementation for up to eight processors with VMMP calls are given for aligned and nonaligned cases in Figures 13 and 14. The speedup, defined as serial time/parallel time is given for each parallel processing time, whereas the single processor time and the serial time are given as no. processors = 0. These measurements do not include I/O time. Obviously, the start-up cost at process creation overhead from the use of the VMMP is best demonstrated when we examine the performance of the parallel algorithm on a single processor, relative to the corresponding serial performances in these figures. For example, when the body surface is aligned with the grid and $N = 16$, the processing time for the serial algorithm is 700 seconds whereas the processing time for the parallel algorithm utilizing one processor is 682 seconds. The difference of 18 seconds is due to VMMP calls and in general for one processor the overhead is about 3% of serial time.

We observe from Figures 13 and 14 that the speedup increases with the number of points, and the performance is dramatically better for very fine grids and a relatively small number of processors. This is due to the locality of the involved calculations, so that for large grid sizes, the communication overhead becomes negligible compared with processing time of each processor.

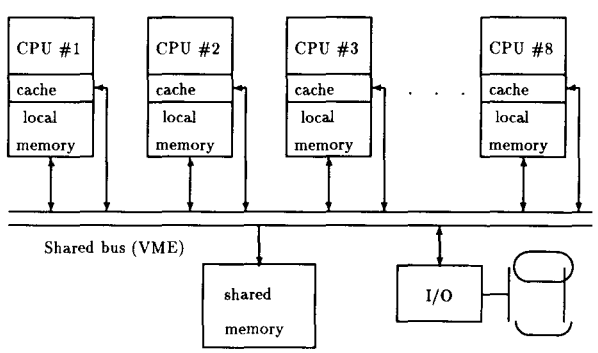


FIGURE 12 MOS-Shared memory multiprocessor architecture.

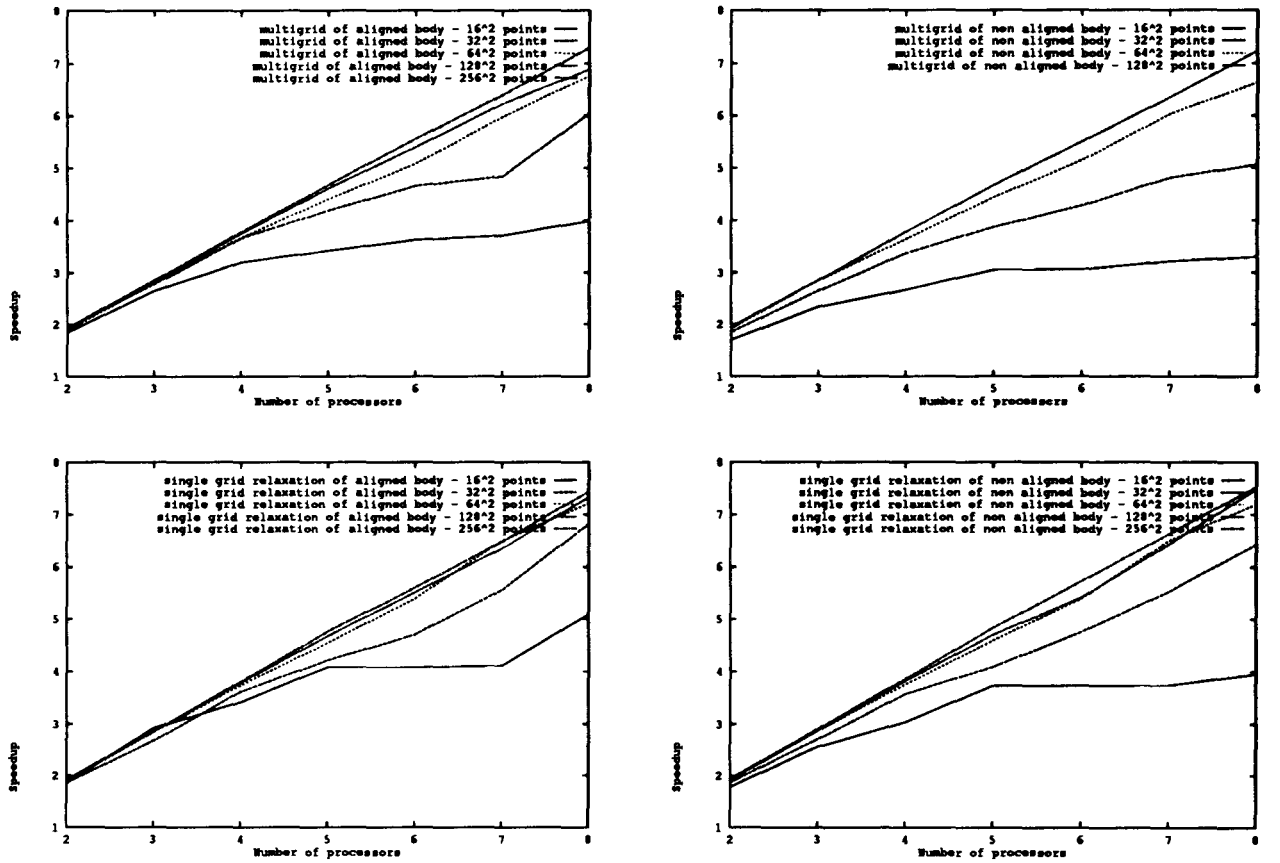


FIGURE 13 MOS speedup vs. number of processor graphs: lower graphs—single grid relaxation; upper graphs—grid relaxation; left—when the body surface is aligned with the grid; right—when the body surface is not aligned with the grid.

Note also that the performance in the nonaligned case for small N and large p is relatively poor due to the enhanced overhead caused by the dummy points. Alignment overhead is shown to be up to about 200% of aligned grid processing time in both the single and multigrid cases.

4.2 Sequent Symmetry Performance

The multi-user Sequent Symmetry machine contains 26 i386 processors with 64 KB of cache for each processor. Global memory size is 32 MB. The Sequent runs the DYNIX operating system, which is a parallel variant of UNIX. Figure 15 illustrates the Sequent Symmetry structure. The results for the single grid and multigrid in the aligned and nonaligned cases are presented in Figures 16 and 17. The performance here is shown to be similar to that of the MOS when the processor's load is large (i.e., for large grid points and small number of processors) and an efficiency of over 90% is achieved. However, some variations are observed for a small number of grid points. Alignment overhead is again up to about 200%. We note, however, that poor efficiency is expected for small N and relatively large p, when the processor's load is very small and communication overhead becomes significant.

4.3 Message-Passing Performance

Our algorithm has been implemented also on a distributed memory multiprocessor comprising eight T800 Transputer nodes. Each node contains a T800 Transputer and 2 or 4 MB of local memory. The transputer clock speed is 20 MHz and memory access time is 80 nsec. The nodes are connected by high-speed bidirectional serial links. There is no shared memory and the Transputers run the HELIOS distributed operating system [18]. The host server provides I/O services for the

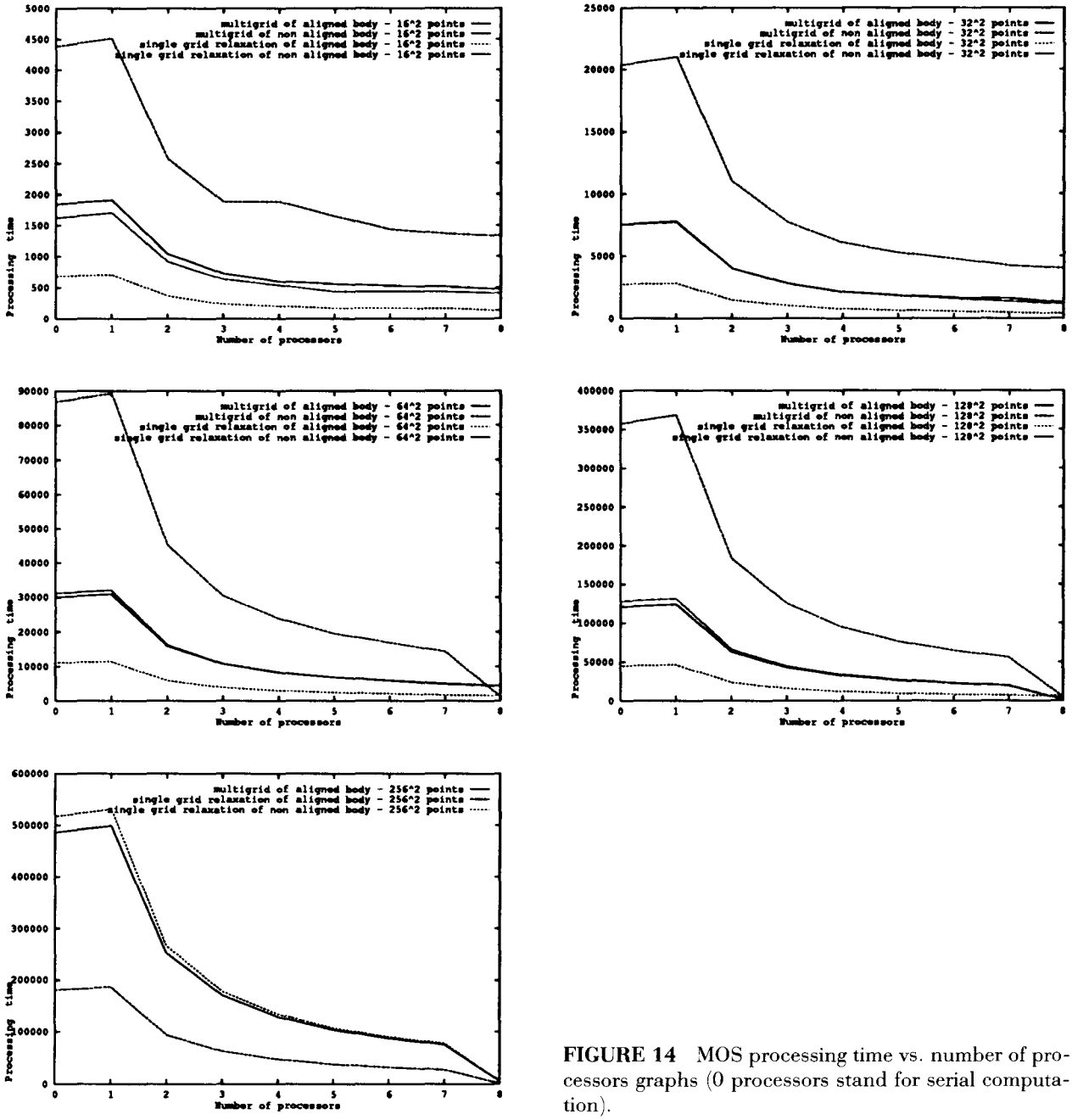


FIGURE 14 MOS processing time vs. number of processors graphs (0 processors stand for serial computation).

Transputers. Figure 18 illustrates the Transputer system structure. Figures 19 and 20 show the speedup and processing time accordingly. When the processor's load is large enough, an efficiency of over 90% is attained. However, for the small processor's load the message-passing performance of our algorithm is degraded significantly as compared with the shared-memory results, mainly due to the increased communication overhead. Note also that the alignment in the message-passing case is reduced to about 20% in the single grid case and to about 65% in the multigrid case.

5 PROCESSING TIME ANALYSIS

5.1 The Single Grid Solutions

For each fixed $N = 16, 32, 64, 128, 256$ we consider the following processing time model:

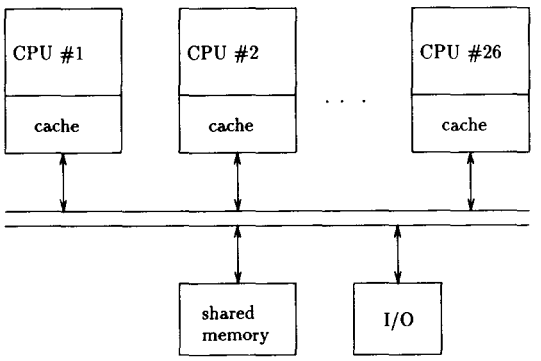


FIGURE 15 Sequent Symmetry shared memory multiprocessor structure.

$$T_{N,p} = n\left(t\frac{N^2}{p} + cN\right) \tag{12}$$

where $T_{N,p}$ is the processing time for single grid relaxation, n is the number of relaxation iterations, N^2 is the number of interior grid points, p is the number of processors, t is the computation time of one grid point in a relaxation step, and c is the proportional factor of overhead time for data transmission among the processors (exchange of overlapped areas).

Using Equation 12 we assume equal time delays for all processors when they try to transmit data on a shared bus while another processor is occupying the bus. Under this assumption, and due to the specific choice of our body shape, neither t nor c should depend on N or p.

In our computations we used $n = 100$.

The values of c and t of Equation 12 that provide the least squares fit to the performance results for different values of N are listed in Table 1.

We can see from Equation 12 that the maximal number of processors to be utilized in our algorithm is the square root of grid points (namely $p = N$). In this case, each processor is assigned one horizontal line (Fig. 9).

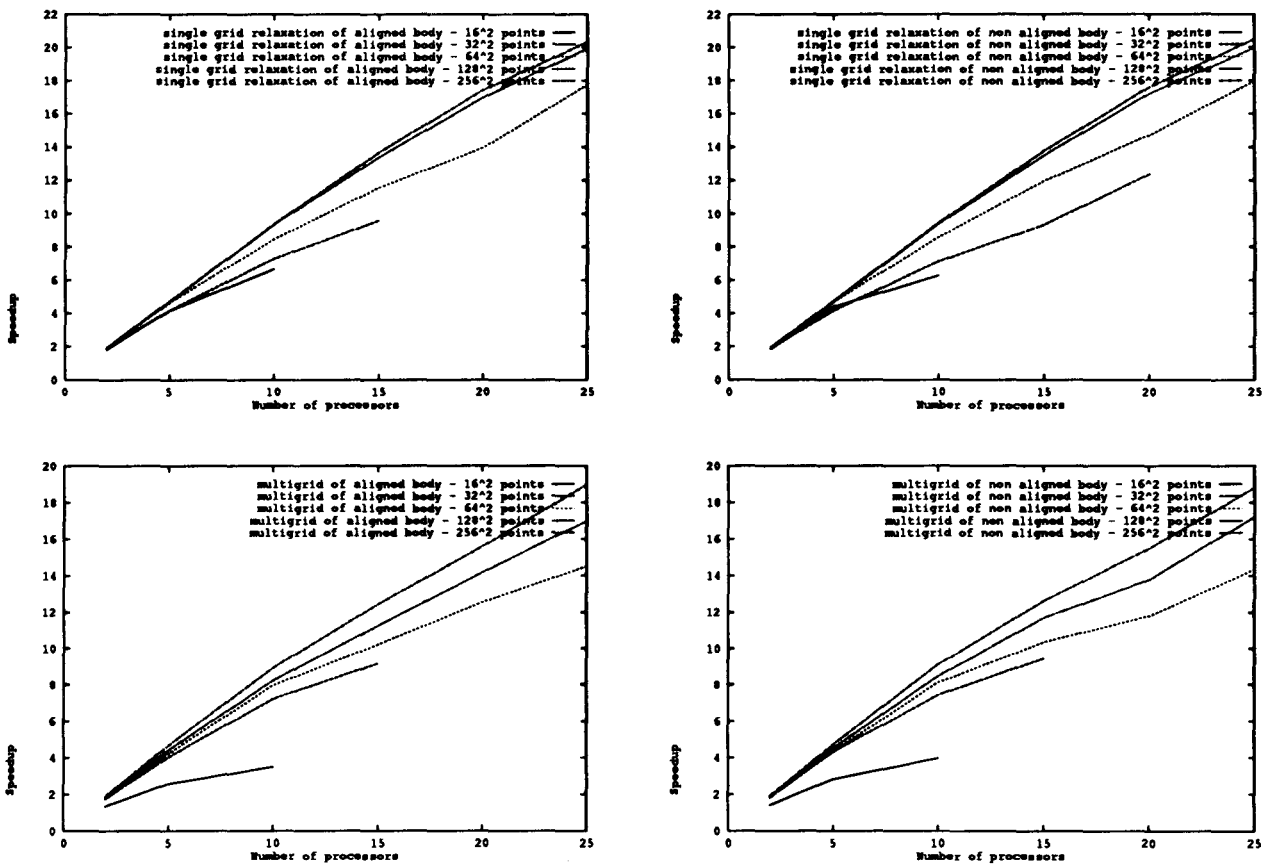


FIGURE 16 Sequent Symmetry speedup vs. number of processor graphs: upper graphs—single grid relaxation; lower graphs—multigrid relaxation; left—when the body surface is aligned with the grid; right—when the body surface is not aligned with the grid.

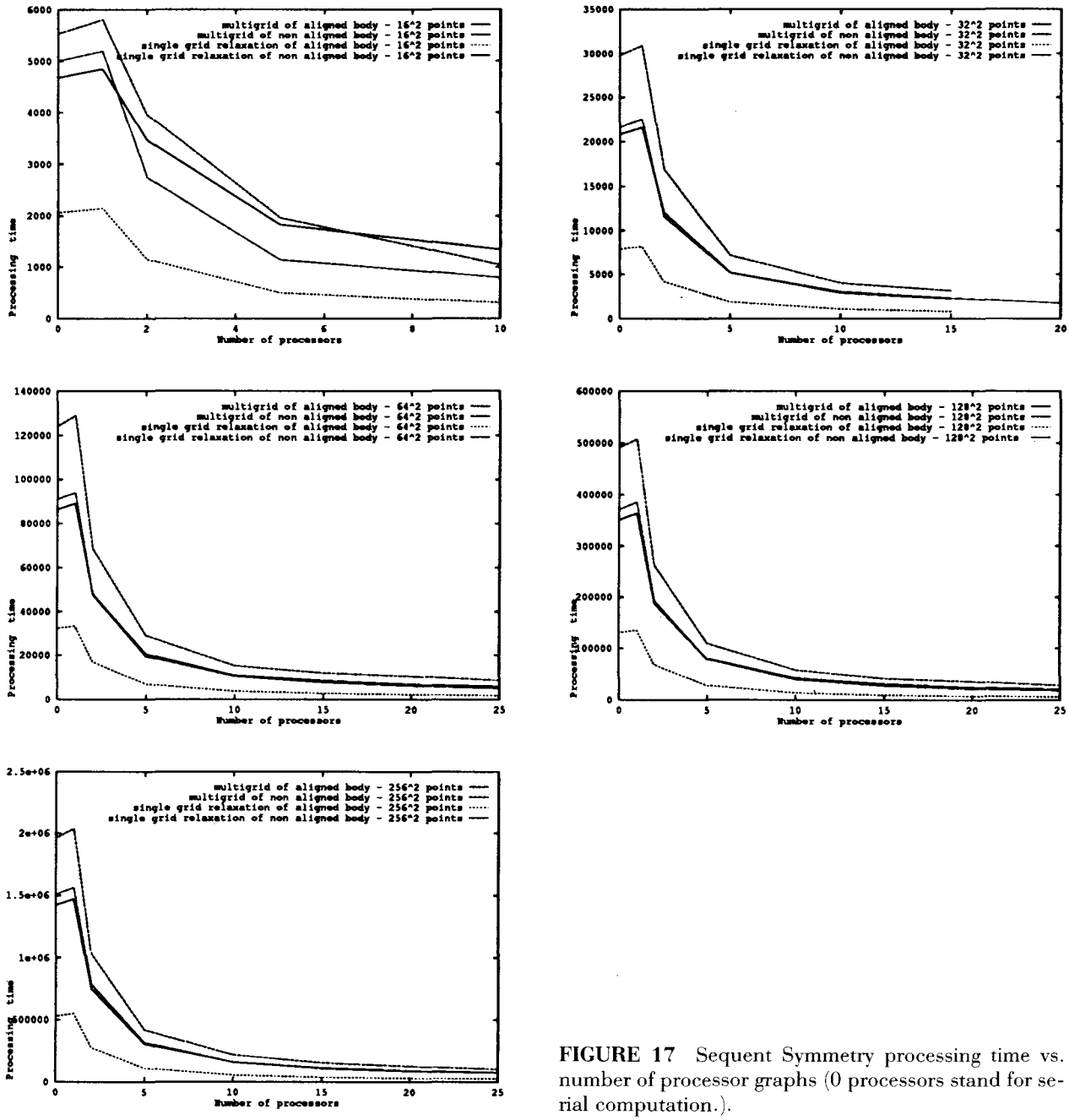


FIGURE 17 Sequent Symmetry processing time vs. number of processor graphs (0 processors stand for serial computation.).

5.2 The Multigrid Solutions

For our V-cycle multigrid solutions in which $N = 16, 32, 64, 128, 256$ we consider the following processing time model:

$$A_{p,N} = 2 \sum_{k=0}^{\log_2 N-1} \left(T_{\frac{N}{2^k}, f\left(p, \frac{N}{2^k}\right)} + F_{\frac{N}{2^k}, f\left(p, \frac{N}{2^k}\right)}\right) \quad (13)$$

where $A_{p,N}$ is the multigrid processing time on N × N grid points using p, processors, $T_{a,b}$ is the relaxation processing time on $a \times a$ grid points using b, processors, $F_{a,b}$ is the time needed to transfer grid with $a \cdot a$ points to $\left(\frac{a}{2} \cdot \frac{a}{2}\right)$ points $\left(\text{or } \left(\frac{a}{2} \cdot \frac{a}{2}\right) \text{ to } a \cdot a\right)$ using b processors, and $f(p, l)$ is defined by Equation 11.

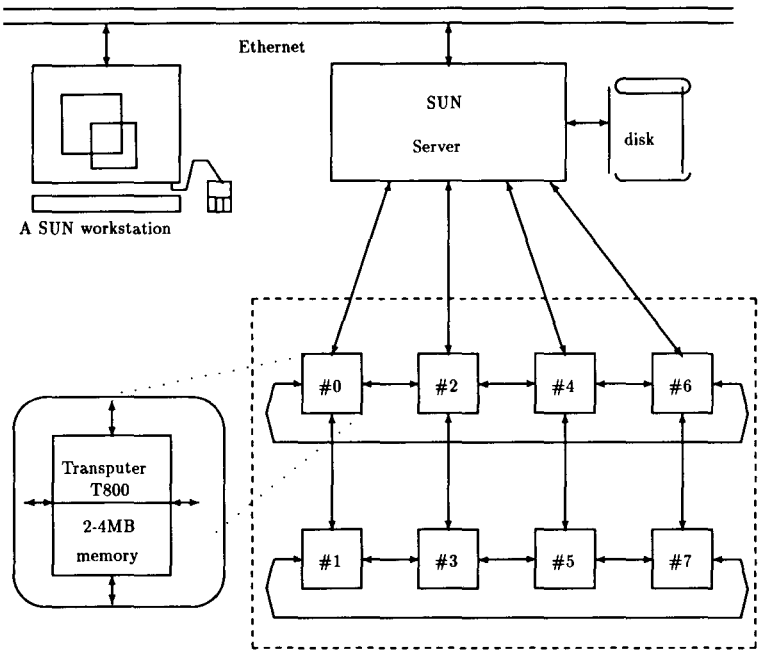


FIGURE 18 Transputer message-passing multiprocessor structure.

Table 1. c and T Values (in Seconds)

Multiprocessor	Points Generation Method	Value	N 16	32	64	128	256
MOS	Aligned	c	0.0311	0.0312	0.0310	0.0372	0.0542
		t	0.0250	0.0265	0.0275	0.0281	0.0283
MOS	Not aligned	c	0.0933	0.0922	0.0911	0.0925	0.0921
		t	0.0587	0.0725	0.0773	0.0797	0.0807
Sequent Symmetry	Aligned	c	0.1097	0.0852	0.0892	0.0885	0.0883
		t	0.0949	0.0962	0.0979	0.1055	0.0911
Sequent Symmetry	Not aligned	c	0.1479	0.2388	0.2273	0.2142	0.2538
		t	0.1849	0.1962	0.2079	0.2155	0.2191
Transputers	Aligned	c	0.2119	0.2258	0.2168	0.2122	
		t	0.0409	0.0451	0.0538	0.0596	
Transputers	Not aligned	c	0.3309	0.3126	0.3061	0.2989	
		t	0.0397	0.0540	0.0662	0.0722	

Table 2. z Values

Multiprocessor	Points Generation Method	N 32	64	128	256
MOS	Aligned	0.127	0.061	0.0193	0.020
MOS	Not aligned	0.447	0.479	0.019	
Sequent Symmetry	Aligned	0.011	0.151	0.027	0.013
Sequent Symmetry	Not aligned	0.368	0.192	0.129	0.356
Transputers	Aligned	0.147	0.119	0.114	0.209
Transputers	Not aligned	0.363	0.288	0.322	0.351

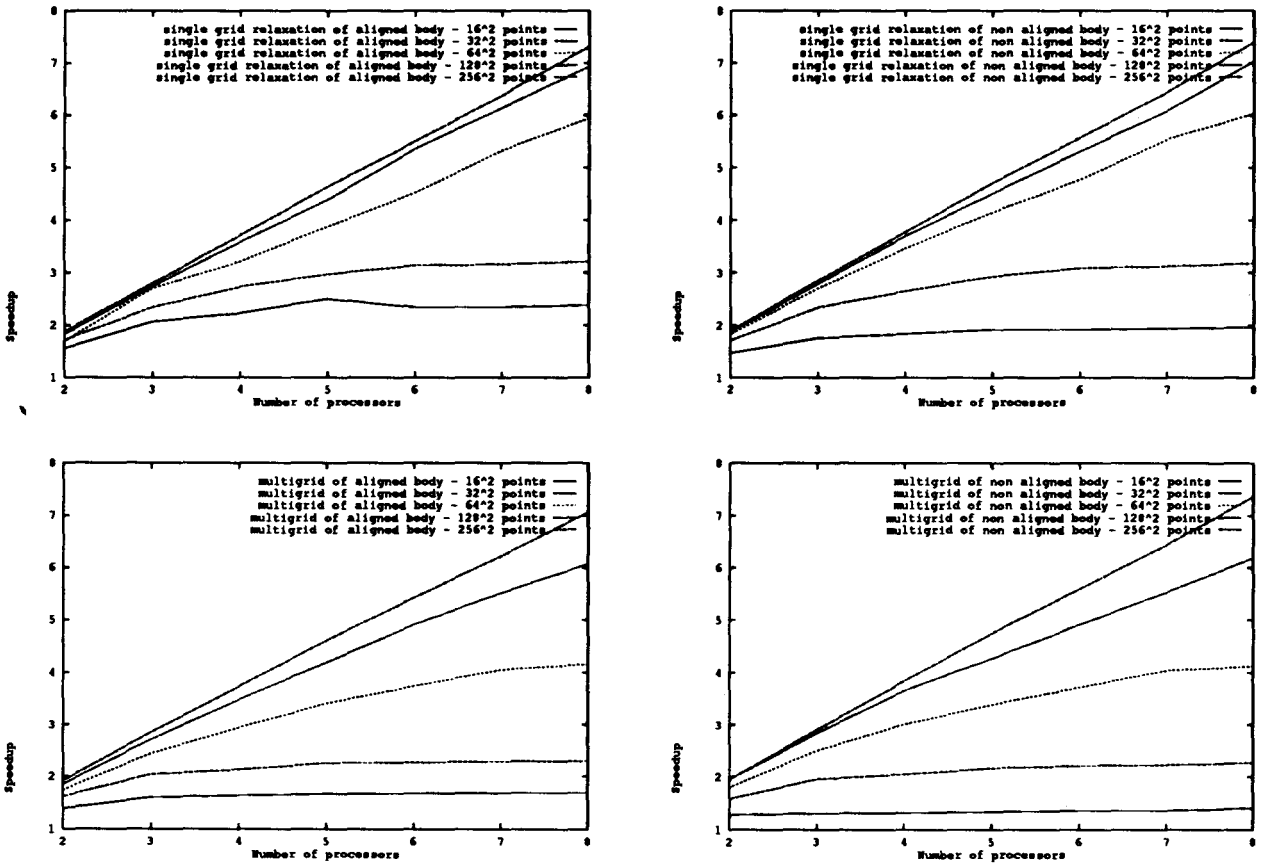


FIGURE 19 Message-passing speedup vs. number of processor graphs: upper graphs—single grid relaxation; lower graphs—multigrid relaxation; left—when the body surface is aligned with the grid, right—when the body surface is not aligned with the grid.

In this processing time model we made the same assumptions of Equation 12 as well as equal time for grid interpolation and for grid projection. Because we are using a full weighted projection we may assume nearly equal times for grid interpolation and for grid projection. Then, neglecting the residual computation (step 2 in the μ − cycle algorithm), $F_{N,p}$ is given by

$$F_{N,p} = z\frac{N^2}{p} \tag{14}$$

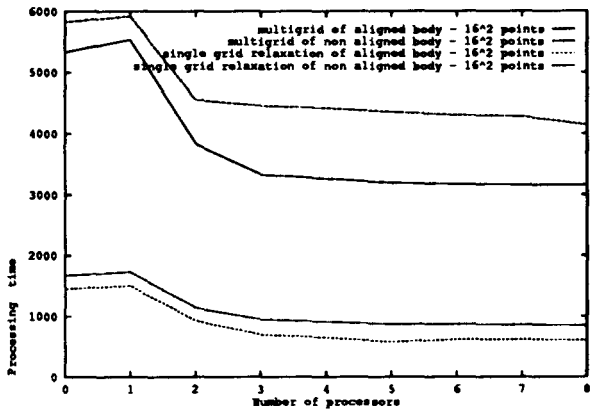
where z is the time needed to receive one grid point value, interpolate (or project) it on finer (coarser) grid, and transmit the point value to the new grid, N^2 is the number of interior grid points, and p is the number of processors.
Using the identity:

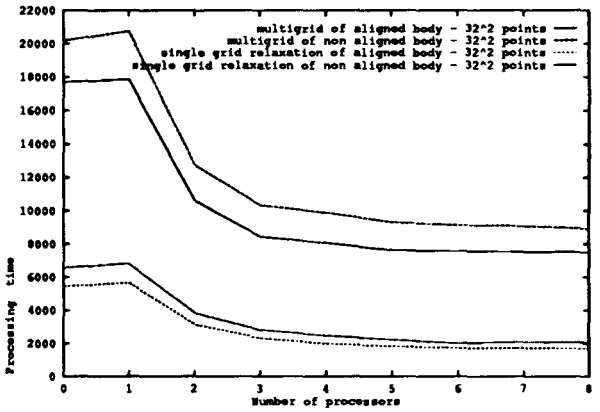
$$F_{N,f(p,N)} = \frac{(A_{p,N} - A_{p,\frac{N}{2}})}{2} - T_{N,f(p,N)} \tag{15}$$

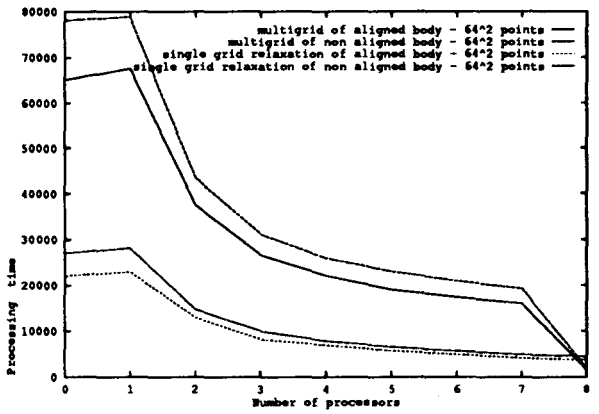
in which $T_{N,f(p,N)}$ is defined by Equation 12, we may evaluate the least squares values of z, corresponding to the performance results. They are presented in Table 2. Note that for the Transputers $z = O(10^{-1})$ whereas for the MOS and for the Sequent machines it varies by a factor of 10 − 20. These large variations are probably due to large variation in bus contention for different runs in the MOS and Sequent machines and due to limited amount of cache memory in those machines.

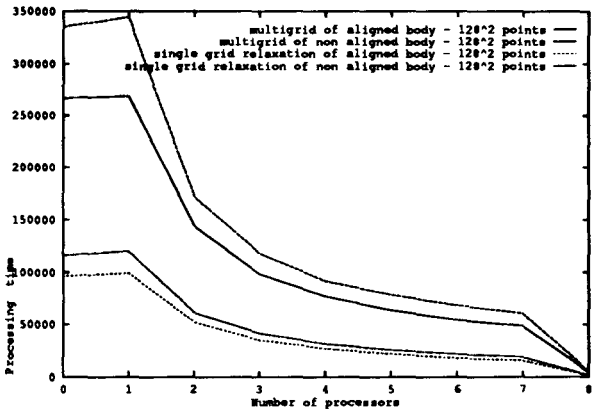
We can now use Equation 13 and predict the processing time if the number of processors is the same as the square root of the number of grid points (namely $p = N$). This is the maximal number of processors to be utilized in our algorithm. We then obtain the information as shown in Table 3.

We can see that in this case, processing time depends on N in a nonlinear manner. The average increase of processing time when N is doubled is about 2.5 in both the aligned and the unaligned









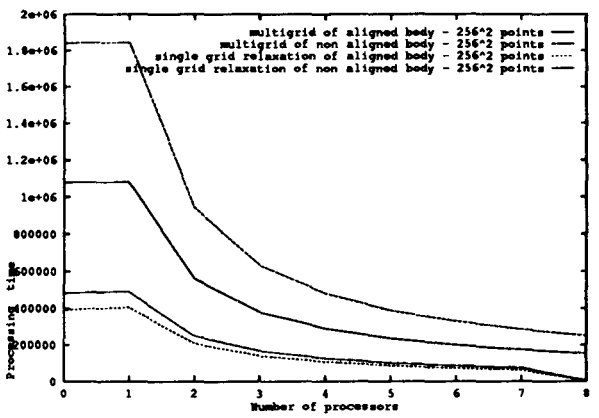


FIGURE 20 Message-passing processing time vs. number of processor graphs (0 processors stand for serial computation).

case. Hence, if $A = g(N)$ then, $g(2N) \approx 2.5g(N)$, and $g(N) \approx N^{\log_2 2.5}$ so that $A = O(N^{1.3})$. The nonlinearity in the speedup measurements can be explained by the change of number of processors being utilized in each V-cycle level in order to increase efficiency, as explained earlier (Table 3).

6 SUMMARY

We have presented and implemented an efficient algorithm for single grid and multigrid relaxation of a boundary value model problem on both shared bus and shared-memory MIMD and message-passing machines by using the VMMP soft-

Table 3. Predict Processing Time Where $p = N$

Multiprocessor	Points Generation Method	No. of Points	Optimal Processing Time
MOS	Aligned	16^2	240
		32^2	618
		64^2	1675
		128^2	3053
		256^2	6863
MOS	Not aligned	16^2	358
		32^2	1643
		64^2	3860
		128^2	8273
Sequent Symmetry	Aligned	16^2	653
		32^2	1707
		64^2	3859
		128^2	8171
		256^2	16832
Sequent Symmetry	Not aligned	16^2	895
		32^2	3708
		64^2	9302
		128^2	20336
		256^2	44730
Message passing	Aligned	16^2	860
		32^2	2603
		64^2	6083
		128^2	13070
		256^2	27860
Message passing	Not aligned	16^2	926
		32^2	3296
		64^2	8099
		128^2	17682
		256^2	37335

ware package. As a model problem we considered the two-dimensional Poisson equation with Dirichlet boundary conditions in a rectangular domain. We do emphasize, however, that our algorithm can be modified accordingly for more complicated boundary value problems. The parallel single/multigrid algorithm is noticed to fit well this architecture when, p, the number of processors is less than, N, the maximal number of grid points along the axes.

For large number of points the efficiency is over 90%. In general, efficiency is higher in the nonaligned case as compared with the aligned case. In the case of a single grid relaxation in the two-dimensional case when $p = N$, processing time increases nearly linearly with N. However, in the multigrid case, this dependence noticed to be nonlinear ($O(N^{1.3})$) mainly because of the change in the number of processors being utilized in each multigrid level (in order to increase efficiency). When we consider a body that is embedded in and unaligned with the grid net, processing time obviously increases, as compared with the aligned case. In our case it is tripled.

ACKNOWLEDGMENTS

The authors wish to thank Dr. B. Epstein, A. L. Luntz, and A. Nachson from the Israel Aircraft Industries, CFD Aerodynamics Department, for introducing the problem and continuing support during all stages of this work. Supported by research grant 0337 of the Israeli National Council for Research and Development.

REFERENCES

[1] G. Chesshire and A. Jameson, *Applications of Supercomputers in Engineering: Fluid Flow and*

Stress Analysis Applications, Proceedings of the First International Conference. New York, NY: Elsevier, 1989, pp. 79–88.

[2] D. Kanowitz, "SOR and MGR[v] experiments on the Crystal multicomputer", *Parallel Comput.*, vol. 4, pp. 117–142, 1987.

[3] C. Koesoemodiprodjo, E. Hensel, and S. Castillo, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 1989.

[4] H. Taylor and S. Markel, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 1989.

[5] J. Burmeister and G. Horton, *Multigrid 3*, Birkhauser: Verlag, 1991.

[6] U. Gartel, A. Krechel, A. Niestegge, and H. J. Plum, *Multigrid 3*, Birkhauser: Verlag, 1991.

[7] W. Sweldens and D. Roose, *Multigrid 3*, Birkhauser: Verlag, 1991.

[8] B. Epstein, A. L. Luntz, and A. Nachson, "Multigrid computation of transonic flow about complex aircraft configurations, using cartesian grid and local refinement, Israel Aircraft Industries, Ben Gurion Airport, Israel.

[9] B. Epstein, A. L. Luntz, and A. Nachson, *GMD-Studian Nr. 110, Proceedings of the 2nd European Conference on Multigrid Methods*, Kolen, 1985.

[10] E. Gabber, "VMMP: A practical tool for the development of portable and efficient programs for multiprocessors," *IEEE Trans. Parallel Distrib. Systems*, vol. 1, pp. 304–317, 1990.

[11] E. Gabber, "VMMP user's guide," Technical Report 239/92, Eskenasy Institute of Computer Sciences, Tel-Aviv University, Tel-Aviv, Israel, January 1992.

[12] D. M. Young, *Iterative Solution of Large Linear Systems*, New York, NY: Academic Press, 1971.

[13] W. Hackbusch, *Multigrid Methods and Applications*, Berlin: Springer-Verlag, 1985.

[14] E. Gabber, A. Averbuch, and A. Yehudai, "Portable, parallelizing pascal compiler," *IEEE Software*, vol. 10, pp. 71–81, 1993.

[15] E. Gabber, *Proceedings of the 3rd Israel Conference on Computer Systems and Software Engineering*, Tel Aviv, Israel: 1988, pp. 122–132.

[16] A. Garcia, D. J. Foster, and R. F. Freitas, "The advanced computing environment multiprocessor workstation," IBM Research Report RC 14491 (no. 64901), IBM T. J. Watson Research Center, March 1989.

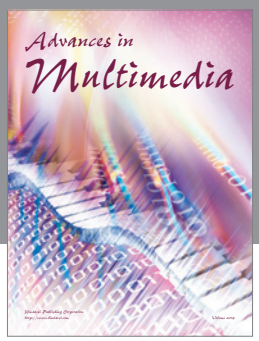
[17] A. Barak and A. Litman, "MOS: A multiprocessor distributed operating system," *Software Practice Exp.*, vol. 15, pp. 725–737, 1985.

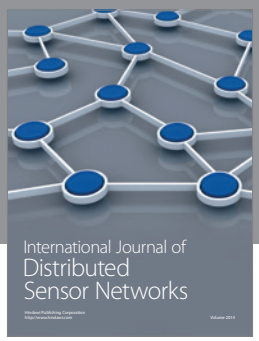
[18] Prehilion Software, *The HELIOS Operating System*, Englewood Cliffs, NJ: Prentice Hall, 1989.

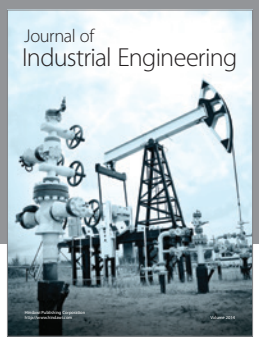
[19] A. Averbuch, E. Gabber, B. Gorodissky, and Y. Medan, "A Parallel FFT on a MIMD machine," *Parallel Comput.*, vol. 15, pp. 61–74, 1990.

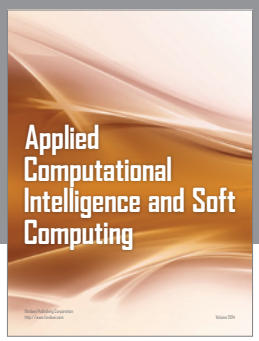
[20] S. Itzikowitz, E. Sammsonov, H. Primack, and A. Averbuch, *Third International Association for Math and Computer Simulation (IMACS)*, Boston: MA, Harvard University, 1991.

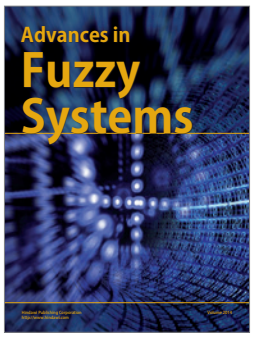
[21] D. Amitai, A. Averbuch, R. Friedman, and E. Gabber, "Portable parallel implementation of BLAS," Technical Report 253/92, Eskenasy Institute of Computer Sciences, Tel-Aviv University, Tel-Aviv, Israel, March 1991.

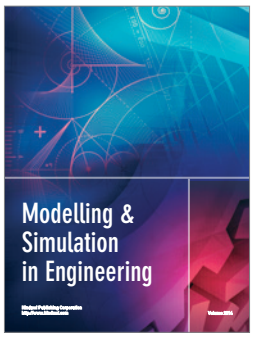


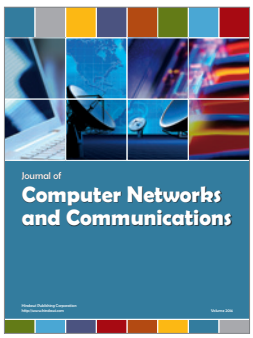












Hindawi
Submit your manuscripts at
http://www.hindawi.com