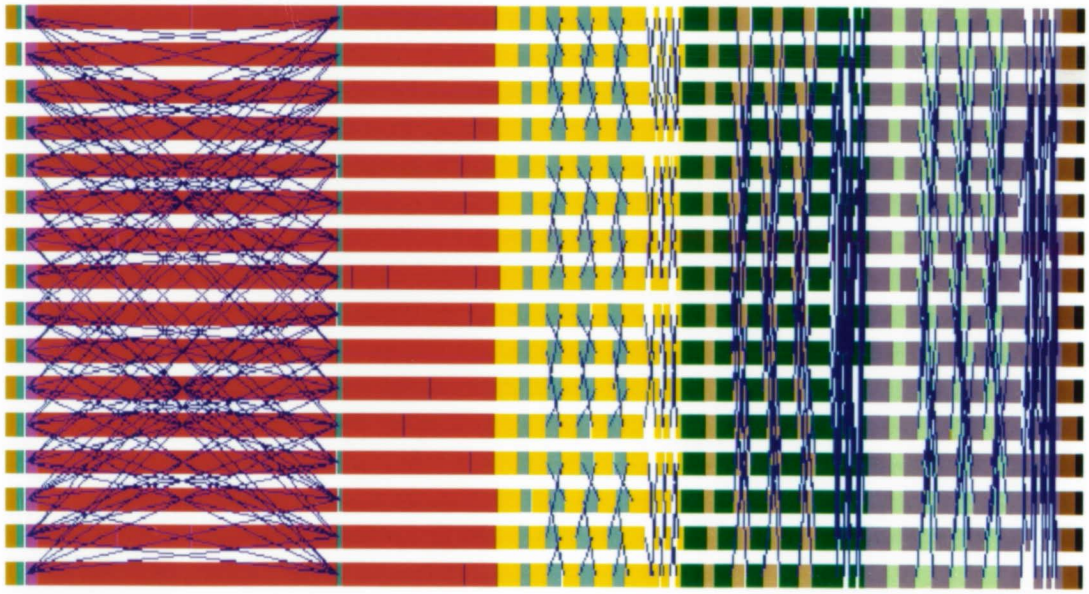
MCAT Institute
Progress Report
93-13

Implementation of ADI-schemes on MIMD parallel computers

Rob F. Van der Wijngaart



July 1993

NCC2-752

MCAT Institute
3933 Blue Gum Drive
San Jose, CA 95127

Implementation of ADI-schemes on MIMD Parallel Computers

Rob F. Van der Wijngaart

1 Introduction

In order to simulate the effects of the impingement of hot exhaust jets of High Performance Aircraft on landing surfaces a multi-disciplinary computation coupling flow dynamics to heat conduction in the runway needs to be carried out. Such simulations, which are essentially unsteady, require very large computational power in order to be completed within a reasonable time frame of the order of an hour. Such power can be furnished by the latest generation of massively parallel computers. These remove the bottleneck of ever more congested data paths to one or a few highly specialized central processing units (CPU's) by having many off-the-shelf CPU's work independently on their own data, and exchange information only when needed.

During the past year the first phase of this project was completed, in which the optimal strategy for mapping an ADI-algorithm for the three-dimensional unsteady heat equation to a MIMD parallel computer was identified. This was done by implementing and comparing three different domain decomposition techniques that define the tasks for the CPU's in the parallel machine. These implementations were done for a Cartesian grid and Dirichlet boundary conditions. The most promising technique was then used to implement the heat equation solver on a general curvilinear grid with a suite of nontrivial boundary conditions.

Finally, this technique was also used to implement the Scalar Pentadiagonal (SP) benchmark, which was taken from the NAS Parallel Benchmarks report [1].

All implementations were done in the programming language C on the Intel iPSC/860 computer.

2 Domain decompositions

The first domain decomposition examined was the static block-Cartesian [2], [3]. Every CPU receives one contiguous block of the global grid and is responsible for computation of the solution on it. This decomposition was found to be inefficient because of the large number of interactions needed between the different grid blocks. These interactions, which take the shape of packets of data (messages) sent between CPU's, suffer from long start-up times (latency).

The second domain decomposition considered was the dynamic block-Cartesian [2]. Here every CPU again receives one contiguous grid block, but the orientation of the grid block changes in order to accommodate the different implicit solution directions used by ADI. This method was found to be significantly more efficient than the static block-Cartesian, but it does not scale well to large numbers of processors. It also requires transmission of very large messages between all the CPU's in the parallel machine, which will lead to congestion (edge contention) on the newest parallel architectures.

The last domain decomposition examined was a multi-partition method that was adapted to the so-called hypercube topology of the iPSC/860 by Bruno and Cappello [5]. Now every CPU receives a small number of contiguous grid blocks (cells, or partitions) that are allocated within the overall grid in such a way that during any of the phases of the ADI algorithm there is work to do for each CPU. This method was found to be optimal in terms of computational speed, because of a very equally distributed work load, a small number of messages sent between CPU's, and the possibility to mask communications by performing computations concurrently.

3 Curvilinear algorithm

The Bruno-Cappello method was subsequently implemented for the heat equation in generalized curvilinear coordinates. Implementation of more complicated boundary conditions, such as C-grid conditions, proved to be efficient and relatively easy because of the high-level data structures used. Consequently, the complaints voiced by other authors [6], [4] that the Bruno-Cappello method is too complex and too inefficient for practical applications could be construed as defects of Fortran, which does not support high-level

data structures. It was found that the Bruno-Cappello method was very well suited for the more involved curvilinear problem with non-trivial boundary conditions, since more communications could be overlapped with computations Moreover, the boundary condition implementation, which is an important source of load imbalance for other methods, could be balanced well in this case.

The results of the above investigations will be presented at the Supercomputing '93 conference to be held in Portland, Oregon, November 15-19, 1993. A draft of the paper to be included in the proceedings is attached in the appendix.

4 Scalar Penta-diagonal Benchmark SP

SP is a model problem that has most of the essential features of the diagonalized Beam-Warming flow solver OVERFLOW, and can be used as a stepping stone for constructing the full-fledged flow solver. It was also implemented using the Bruno-Cappello method, but no timings have been obtained yet.

References

[1] D. Bailey, J. Barton, T. Lasinski, H. Simon, *The NAS parallel benchmarks*, NASA Ames Report RNR-91-002 Revision 2, 1991

[2] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, to be presented at Supercomputing '93, Portland, Oregon, November 15-19, 1993

[3] J.S. Ryan, S.K. Weeratunga, *Parallel computation of 3-D Navier-Stokes flowfields for supersonic vehicles*, AIAA Paper 93-0064, 31st Aerospace Sciences Meeting & Exhibit, Reno, NV, January 11-14, 1993

[4] P.J. Kominsky, *Performance analysis of an implementation of the Beam and Warming implicit factored scheme on the NCube hypercube*, Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 8-10, 1990, IEEE Computer Society Press, Los Alamitos, CA

[5] J. Bruno, P.R. Cappello, *Implementing the Beam and Warming method on the hypercube*, Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Jan. 19-20, 1988

[6] N.H. Naik, V.K. Naik, M. Nicoules, *Parallelization of a class of implicit finite difference schemes in computational fluid dynamics*, International Journal of High Speed Computing, Vol. 5, No. 1, pp. 1-50, 1993

APPENDIX A

Efficient implementation of a 3-dimensional ADI method on the iPSC/860

Rob F. Van der Wijngaart
MCAT Institute, NASA Ames Research Center
Moffett Field, CA 94035

Abstract

A comparison is made between several domain decomposition strategies for the solution of three-dimensional partial differential equations on a MIMD distributed memory parallel computer. The grids used are structured, and the numerical algorithm is ADI. Important implementation issues regarding load balancing, storage requirements, network latency, and overlap of computations and communications are discussed. Results of the solution of the three-dimensional heat equation on the Intel iPSC/860 are presented for the three most viable methods. It is found that the Bruno-Cappello decomposition delivers optimal computational speed through an almost complete elimination of processor idle time, while providing good memory efficiency.

1 Introduction

Implicit numerical algorithms for the solution of multi-dimensional partial differential equations (PDE's) are usually more efficient computationally than explicit methods, when implemented on conventional (vector) computers. However, they are harder to program efficiently on parallel computers due to a more global data dependence than is exhibited by explicit methods. Numerical solution of PDE's typically involves more or less the same operations for all the points in a computational grid used to discretize the problem space. Consequently, domain decomposition is the natural way of creating separate tasks for a parallel computer: a roughly equal number of grid point is assigned to each processor. Depending on the type of implicit algorithm chosen, some domain decompositions perform better than others. Efficiency is also affected by hardware parameters (e.g. network latency and bandwidth, and processor memory) and operating model (e.g. MIMD, SIMD). In this paper we compare three viable domain decompositions for the solution of three-dimensional PDE's using ADI (Alternating Direction Implicit) on the Intel iPSC/860 MIMD parallel computer. The results of this study also apply to other

line-based solution strategies, such as line-relaxation, when multiple sweep directions are used during each iteration.

As an example, we solve the time-dependent three-dimensional heat equation. Since the aim is to assess parallel efficiency, the problem is kept as simple as possible (i.e. Cartesian grid, constant mesh spacing, Dirichlet boundary conditions, constant material properties, no source term). As a result, the computational program is simple and easy to analyze, and the computations per grid point are at a bare minimum. No effort was made to use the simplifying assumptions to reduce communication, so a relatively bad balance results between computation and communication time; a worst-case parallel performance analysis is obtained.

2 Problem formulation

The equation to be solved is:

$$\rho c T_t = \nabla \cdot (\mathbf{k} \nabla T)\,, \tag{1}$$

where T is temperature, t time, ρ density, c specific heat, and $\mathbf{k}$ the conduction tensor. Assuming $\mathbf{k}$ to be a constant scalar, i.e. $\mathbf{k} = k\mathbf{I}$, we get

$$\rho c T_t = k(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2})T\,. \tag{2}$$

Equation (2) is subsequently discretized using central differencing in space and the θ-method in time:

$$(1 - \frac{h\theta k}{\rho c}[(\delta_x^c)^2 + (\delta_y^c)^2 + (\delta_z^c)^2])\Delta T = \frac{hk}{\rho c}[(\delta_x^c)^2 + (\delta_y^c)^2 + (\delta_z^c)^2]T\,. \tag{3}$$

Here $\delta_{x_i}^c$ signifies the central difference operator in the x_i-direction, T is the temperature, ΔT its temporal increment, and h is the size of the time step. The parameter θ controls the 'implicitness' of the problem ($\theta = 0$ yields Euler explicit, $\theta = 1$ gives Euler implicit, and $\theta = 1/2$ defines the second-order-accurate Crank-Nicolson scheme). Equation (3), which is said to be in delta form, defines a matrix equation with a very large bandwidth due to the three-dimensionality of the discrete operator. Approximate factorization reduces this operator to a product of three one-dimensional operators with a bandwidth of only three each (e.g. [1]). So equation (3) is approximated by:

$$(1 - \frac{h\theta k}{\rho c}(\delta_x^c)^2)(1 - \frac{h\theta k}{\rho c}(\delta_y^c)^2)(1 - \frac{h\theta k}{\rho c}(\delta_z^c)^2)\Delta T = \frac{hk}{\rho c}[(\delta_x^c)^2 + (\delta_y^c)^2 + (\delta_z^c)^2]T \tag{4}$$

An outline of the numerical algorithm is:

1. Compute rhs, the right hand side of equation (4).
2. Solve the system $(1 - \frac{h\theta k}{\rho c}(\delta_x^c)^2)A = rhs$ along lines in the x-direction.

3. Solve the system $(1 - \frac{h\theta k}{\rho c}(\delta_y^c)^2)B = A$ along lines in the y-direction.

4. Solve the system $(1 - \frac{h\theta k}{\rho c}(\delta_z^c)^2)\Delta T = B$ along lines in the z-direction.

5. Update T for all interior grid points.

3 Domain decomposition

The Intel iPSC/860 computer on which the problem is solved is of the MIMD (Multiple Instruction Multiple Data) distributed memory type. Data is owned by the individual processors in the processor array, which is structured as a hypercube. The only way that data can be shared among processors is by message passing. Sending or receiving a message takes communication time, which goes at the expense of the computing efficiency. Moreover, synchronization and load balancing are an issue; processors should not be allowed to idle because they are out of work or are waiting for data to be prepared by other processors. The following sections discuss three different domain decomposition strategies and the associated numerical implementations. Although many more such strategies are conceivable, these appear the most viable, for they all have a good load balance and attempt to minimize data communication in some sense.

3.1 Static block-Cartesian decomposition

In the static block-Cartesian case, each processor owns one contiguous Cartesian-product subspace—a block—of the whole grid for the duration of the entire computation. This decomposition assumes a very small latency, relatively low communication bandwidth, and limited storage. The grid blocks are as close to cubical as possible in order to minimize surface area, which in turn minimizes the amount of data to be communicated between blocks. It also minimizes storage of an extra layer of points around the grid block, a common and convenient vehicle in domain decomposition strategies for sharing information with neighboring processors.

A serious drawback is, however, that no single block-Cartesian decomposition is efficient for all line solves in steps 2 through 4. Consider step 2, for example. Here a matrix equation is formed for each line in the x-direction across the whole grid. If this line is contained completely in a single processor (which means that the block is of the width of the grid in the x-direction), then all processors can solve their matrix equations independently, and complete parallelism is obtained. However, since the grid is divided into multiple blocks, there must be at least one coordinate direction, say y, that runs across several blocks. That means that in step 3 no whole y-line can be formed within one processor, and a processor must wait to receive information from neighboring blocks before it can do its part of the forward elimination or the back-substitution; communication is needed during line solves.

When using the Thomas algorithm for solving the tri-diagonal matrix equations, the information to be passed to the next processor during the forward elimination consists of the updated right hand side and the upper and diagonal matrix elements at the end of each line (the matrix elements are not strictly needed in the current constant-coefficient case, but we pass them for generality's sake to reflect the communication requirements of curvilinear (section 6) and fully nonlinear algorithms). If the latency is very large, one can collect all such triplets of values for all the line segments in the current grid block and send them to the next processor as one message. On arrival, they can be unpacked and used by the next processor to advance further along the line in the forward elimination step. But this leads to a severe load imbalance, since only one layer of grid blocks perpendicular to the line solve direction is active at any given time. Instead, we send each triplet individually, giving the next processor something to chew on already before starting on the next line segment of the forward elimination within the current processor. This process, called Pipe-lined Gaussian Elimination (PGE) [2], [3], has a much better load balance, provided each grid block contains many more line segments than there are consecutive grid blocks in any coordinate direction. However, it does require sending many small messages. An alternative to PGE that avoids the latter problem is offered by variants of the cyclic reduction algorithm [3], called substructuring methods. These rely on eliminating as many off-diagonal matrix elements as possible within each grid block in parallel before communicating with processors containing neighboring blocks. Substructuring methods are very similar in appearance to solution methods for periodic problems, and they require a comparable number of arithmetic operations, which is almost twice as many as are needed by PGE. Due to this added computational expense, substructuring methods are not considered in this study.

3.2 Dynamic block-Cartesian decomposition

In the dynamic block-Cartesian case, each processor again owns a contiguous grid block, but this time the decomposition changes between the different line solve stages. This decomposition assumes a large latency, relatively high communication bandwidth, and abundant storage. The dynamic redecomposition (also called Mass Reorganization [4] or Complete Exchange [7]) enables the data lay-out to be tailored to the line solve step it supports. Before solution in the x_i-direction (i= 1, 2, 3), the Cartesian blocks are made to be of the width of the grid in that same direction. The extra expense incurred is the communication needed to redecompose the domain, but no data needs to be transferred *during* any of the three solution stages.

The optimal dynamic subdivision is found as follows. The whole grid contains $n_x \times n_y \times n_z$ points. Let $np_{x_j}^{x_i}$ signify the number of processors (blocks) in the x_j-direction during the x_i-line solves, the total number of blocks being np. Some useful identities are:

$$np_{x_i}^{x_i} = 1\,, \quad \prod_{j=1}^{3} np_{x_j}^{x_i} = np\,, \quad (i = 1,2,3)\,. \tag{5}$$

Between the x- and y-line solves the intermediate solution A on a processor has to be communicated to all other processors that need it. The only information that does not need to be communicated lies in the intersection of the Cartesian blocks of the successive decompositions that reside on the same processor during both line solve stages. The size of the intersection region on a single processor is at most:

$$\frac{n_x}{\max(np_x^x, np_x^y)} \cdot \frac{n_y}{\max(np_y^x, np_y^y)} \cdot \frac{n_z}{\max(np_z^x, np_z^y)} = \frac{n_x n_y n_z}{np_x^y np_y^x \max(np_z^x, np_z^y)}. \tag{6}$$

The total number of grid points in a block is $\frac{n_x n_y n_z}{np}$, which makes the total amount of grid point data communicated equal to:

$$n_x n_y n_z \left(\frac{1}{np} - \frac{1}{np_x^y np_y^x \max(np_z^x, np_z^y)} \right). \tag{7}$$

Similar expressions can be derived for communications between y- and z-line solves, and before the final temperature update (step 5). Assuming all inter-processor data transfer can happen without conflicts, the total communication time t_c is:

$$t_c = n_x n_y n_z \left(\frac{3}{np} - \frac{1}{np_x^y np_y^x \max(np_z^x, np_z^y)} - \frac{1}{np_y^z np_z^y \max(np_x^y, np_x^z)} - \frac{1}{np_x^z np_z^x \max(np_y^x, np_y^z)} \right) c, \tag{8}$$

with c the time to send one floating point number. Using the identities (5), t_c can be simplified to:

$$t_c = \frac{n_x n_y n_z}{np} \left(3 - \frac{1}{\max(np_x^y, np_x^x)} - \frac{1}{\max(np_y^z, np_z^y)} - \frac{1}{\max(np_x^z, np_z^x)} \right) c. \tag{9}$$

It is not easy to see how this expression can be minimized for a certain choice of the $np_{x_j}^{x_i}$. Therefore, two extreme cases are considered.

First, map the planes perpendicular to each line solve direction to a *square* processor array, i.e. $np_{x_i}^{x_{(i+1) \bmod 3}} = np_{x_i}^{x_{(i+2) \bmod 3}} = \sqrt{np}$, $(i = 1, 2, 3)$. This leads to a small aspect ratio of the blocks in the plane perpendicular to the lines solve direction. The corresponding communication time t_c^{sq} is:

$$t_c^{sq} = \frac{n_x n_y n_z}{np} \left(3 - \frac{3}{\sqrt{np}} \right) c. \tag{10}$$

Second, map the planes perpendicular to each line solve direction to a *linear* processor array, i.e. $np_{x_i}^{x_{(i+1) \bmod 3}} = np$ or $np_{x_i}^{x_{(i+2) \bmod 3}} = np$, $(i = 1, 2, 3)$. This leads to a large aspect ratio of the blocks in the plane perpendicular to the line solve direction; the domain is dissected into *slices* stretching across the entire width of the grid in two coordinate directions. Equation (9) shows that the corresponding communication time t_c^{lin} is, in general:

$$t_c^{lin} = \frac{n_x n_y n_z}{np} \left(3 - \frac{3}{np} \right) c. \tag{11}$$

An additional gain is obtained by selecting one particular coordinate direction x_{j_0} and requiring $np^{x_i}_{x_{j_0}} = 1$, $(i = 1, 2, 3)$. Then one of the terms on the right hand side of equation (9) is 1, and the communication time drops to:

$$t_c^{lin} = \frac{n_x n_y n_z}{np}\left(2 - \frac{2}{np}\right)c. \tag{12}$$

Now there is one pair of solution steps between which no communication is necessary at all. For example, suppose that $j_0 = 2$, then the grid dissection chosen for either the x-line solves or the z-line solves will also be adequate for the y-line solves. Coordinate direction x_{j_0} is called the *pile*-direction of the decomposition (see below). A comparison of the communication times for the square and the linear decompositions yields:

$$t_c^{lin}/t_c^{sq} = \frac{2}{3}(1 + \frac{1}{\sqrt{np}}), (np \neq 1). \tag{13}$$

For $np > 4$ the linear decomposition is superior, with gains increasing as np grows.

3.3 Bruno-Cappello multi-cell decomposition

In the Bruno-Cappello case ([4], [5]), each processor owns a collection of grid blocks, called *cells*. This decomposition supports a large latency and a relatively low bandwidth, and requires somewhat more memory than the static but a lot less than the dynamic block-Cartesian decomposition. The arrangements of the equally-sized cells is such, that every coordinate plane that cuts the grid intersects with exactly one cell of each processor. The number of cells is the smallest possible to satisfy the above requirement. If the number of processors is again np, then each processor owns $\sqrt{np}$ cells. Consequently, the total number of cells is $np\sqrt{np}$, which are laid out in a $\sqrt{np} \times \sqrt{np} \times \sqrt{np}$ three-dimensional array. No two cells belonging to the same processor abut, so that no complete lines in any coordinate direction can be formed within one processor; communication is again necessary during line solves, but now we do not have to worry about load balancing the algorithm. Therefore, each cell can finish all its line segments during forward elimination before sending a packet of consolidated data to the adjacent cell for processing.

3.4 Right hand side evaluation

So far the cost of assembling the right hand side of equation (4) has been ignored. Whereas the computing cost of that assembly depends only weakly on the decomposition chosen, the communication cost is proportional to the surface area of each block. The surface area is smallest for the static and largest for the dynamic block-Cartesian decomposition. In the latter case the surface area does not scale with the number of processors; the communication overhead of evaluating the right hand side appears to grow indefinitely. But in all three cases the right hand side can be computed for points interior to the blocks or cells owned by each processor while boundary information is being sent to other

processors. Experiments with the AIMS performance monitoring system [6] show that this communication does not lead to processor idle time for any grids of reasonable size.

4 Implementation issues

All cases are programmed in C. This language provides the flexibility and convenience of mixed-type data structures that keep parameter lists short and clean. It also has the advantage that functions are built in for computing the length of system- or user-defined data types (important for sending messages), that dynamic memory allocation is supported, and that interfaces with Fortran subroutines are possible. Tests on the iPSC/860 show that Fortran77 and C have the same computational efficiency. What seems to be a drawback of C is that it does not allow multi-dimensional arrays of variable size in parameter lists, which forces the programmer to map them into one-dimensional arrays with explicit computation of indices. But this can be done easily and efficiently, with no performance degradation. In fact, having explicit control over array lay-out obviates the need for the auxiliary arrays reported in [4].

4.1 Static block-Cartesian

In the static block-Cartesian approach each grid block has dimensions augmented by one in all directions to account for block interface information. The blocks themselves are arranged in a three-dimensional array such that communication between neighboring blocks is also between neighboring processors (Hamming distance of 1). This can be achieved using binary reflected gray codes (see e.g. [10]). The processor number p of a block with indices (i, j, k) in the three-dimensional array of size $(I_{size}, J_{size}, K_{size})$ is given by $p = \text{gray}(i) + I_{size} * (\text{gray}(j) + J_{size} * \text{gray}(k))$. Experiments using gray codes show a performance improvement of about 5% over the canonical numbering $p = i + I_{size} * (j + J_{size} * k)$ for medium-sized grids ($60 \times 60 \times 30$ to $80 \times 80 \times 40$ grid points) on a 32-processor hypercube.

As was mentioned earlier, the computation of the right hand side vector for points interior to the grid block can concur with the exchange of boundary face information between neighboring grid blocks. This requires the use of asynchronous message passing. Extra speed-up is obtained by using so-called *forced* messages, which bypass system wait buffers and get copied immediately into the application space of the receiving processor. Once the boundary data has been received, the right hand side for points on the edge of the grid block can be evaluated. This strategy offers a significant increase in efficiency, although there is a hidden cost; since the computation of the right hand side is split in two (interior and boundary points), the vector length for each of these steps—most notably for the boundary points—is reduced, which leads to a loss of performance on the iPSC/860 vector processors.

The left hand side matrices for the simple Cartesian-grid case are constant, so they need not be constructed explicitly. Consequently, there is no computational work that can be done when transferring information between neighboring cells during the line-solve phases of the algorithm, and simple synchronous message passing is used.

4.2 Dynamic block-Cartesian

In the linear dynamic block-Cartesian approach each processor owns a *slice* of the whole grid, whose orientation depends on the phase of the solution process. Each processor contains a number of slice variables—one for each physical variable defined on its part of the grid—that hold the data in an array of function values. That array is distributed over a number of *pile* data structures, each of which contains a block of data that can be transferred monolithically to other processors during the change of decomposition direction (Figures 1 and 2). A pile stretches across the grid in the x_{j_0}-direction (see section 3.2). No rearrangement of the values within a pile is necessary after transfer.

During the redecomposition phase, each processor needs to send a (different) pile of data to every other processor in the allocated hypercube. These cannot all be nearest neighbors, so there is a danger of *edge contention* [7]; two messages cannot normally share the same data path—*edge*—between two processors in the hypercube, so if communication requires the paths of several messages to overlap (partially), then they will have to wait for each other until the contended edge is freed.

In Figure 2 the data transfer needed for changing the decomposition direction between the x-line solves and the y-line solves is depicted, assuming the pile is aligned with the z-axis. The hatched piles sitting on processor 1 during the x-line solves have to be distributed among processors 0, 2, 3, 4, 5 for use during the y-line solves. Note that pile 1 (open box) need not be communicated, since it stays on processor 1. This is generally true for pile i on processor i. Conversely, processor 1 also receives piles (shaded) from processors 0, 2, 3, 4, 5 for use during the y-line solves. It obviously does not receive information from itself.

It is found in [7] and [8] that this type of communication, called *complete exchange*, suffers from significant edge contention if programmed in a naive way, i.e.:

for pile = 0, np-1 **do**: **if** pile $\neq$ mynumber **then** send-pile-to-processor(pile)

Communication conflicts are avoided by using Bokhari's *linear* algorithm [7]:

for pile = 1, np-1 **do**: send-pile-to-processor((pile+mynumber) mod np)

This is the strategy employed in this study. It is on a par with the stable method and the pairwise-synchronized method with forced messages also described in [7], while outperforming all other algorithms for the global exchange of medium to large-size messages on medium-size hypercubes. Again, asynchronous communication and forced message types are used, which has the advantage that no delay is caused by placing sizeable messages

associated with each pile on the network.

In order to compute the right hand side vector, each processor needs to have access to temperature values on adjacent slices; these are stored in buffer zones. Buffer zones are not included in the slices themselves—as was the case with the static block-Cartesian decomposition—since this would necessitate a certain repacking of pile data during the complete exchange. Instead, interface data is stored in two buffer arrays on each node, one for either side of a slice. The thickness of each buffer is one, because a seven-point-star stencil is used for computing the right hand side. Buffers are shipped to neighboring processors as single messages. In order to keep these communications as efficient as possible, they are overlapped with the computation of the right hand side vector for points interior to the slices. In addition, the slices are numbered using gray codes such that neighboring slices are on neighboring processors, i.e. $p = \text{gray}(slice)$.

4.3 Bruno-Cappello multi-cell

In the Bruno-Cappello approach each cell has dimensions augmented by one in all directions to account for cell interface information. Many lay-outs are conceivable that satisfy the requirement that each coordinate plane cutting across the whole grid intersect with exactly one cell of each processor. In addition, we demand that for a given communication direction all cells belonging to a certain processor send information to only one other processor. For example, suppose a cell on processor 0 has neighbors on processors as indicated in Figure 3, then all the other cells owned by processor 0 exhibit the same configuration.

Such a lay-out of cells can be constructed as follows; starting with a certain assignment of cells in the 'ground' plane ($k = 0$), every subsequent plane has the same relative assignment of cells to processors—save boundary effects—and is shifted in both the i- and j-directions. In order to preserve the neighbor relation in the z-direction, the (periodic) shift for plane k should be of the form $(a*k, b*k)$. Bruno and Cappello show [5] that it is not possible to construct a hypercube mapping of cells to processors that results in nearest-neighbor communication only, but that it is possible to have a maximum communication distance of 2 in one coordinate direction while preserving nearest-neighbor relations in the other two directions. This mapping is constructed easily using gray-code mappings for the assignment of cells to processors in the ground plane, and by applying either the $(k, -k)$ or the $(-k, k)$ shift to subsequent z-planes. In the latter case, cell (i, j, k) lies on node number $p = \text{gray}\left((i+k) \bmod \sqrt{np}\right) + \sqrt{np} * \text{gray}\left((j-k) \bmod \sqrt{np}\right)$. This mapping requires only one message per processor in each of the six coordinate directions (east, west, north, south, top, bottom) when exchanging boundary information with neighboring cells, regardless how many cells there are per node. Again, we can overlap this copying action with computation of the new right hand side vector for interior points of all cells.

An additional advantage of the Bruno-Cappello decomposition is that all processors have exactly one cell face on each of the six faces of the global grid. That means that

boundary effects are the same for all processors, yielding a perfect load balance automatically. In the static and dynamic block-Cartesian cases, processors owning interior grid blocks have a different work load than those that face a grid boundary.

It should be noted that the Bruno-Cappello method described here is a special case of the more general class of multi-partition methods described in detail in [9]; it is tailored towards a scalable implementation on a binary hypercube topology.

4.4 Resources summary

In Table 1 we summarize the number of messages and the amount of communication (8-byte words) needed during one time step of each of the three implementations, as well as the amount of storage (8-byte words) per grid variable. All numbers are for one processor. The grid contains $n \times n \times n$ points, and the total number of processors is np.

It should be stressed that one third of the communication cost in the dynamic and Bruno-Cappello decompositions is hidden by computations, whereas the static decomposition offers hardly any savings in this regard. That is because most messages in the latter case are sent during the line solve stage, when no overlap of computation and communication is possible. The amount of storage required is just for one variable. What makes the dynamic algorithm memory-inefficient is the fact that certain variables have to be stored multiple times because of the different decompositions. This is especially true when generalized coordinates or nonuniform material properties are used, in which case multiple versions of properties and metrics data need to be stored.

5 Comparison

Three sets of computations were done in order to assess the impact of the different domain decompositions on the parallel performance ϕ, the results of which are presented in Tables 2-4. ϕ, also called the *efficiency*, relates the time to execute a certain problem on np processors ($time(np)$) to the time it takes to solve the same problem on just 1 processor, i.e.

$$\phi(np) = \frac{time(1)}{np\ time(np)} \ . \tag{14}$$

It should be noted that $time(1)$ is the true serial execution time on one processor, stripped of all the parallel overhead.

For each case are also listed the computing speed *Mflops* (millions of double precision floating point operations per second). Mflops were computed by multiplying the total number of grid points by the number of floating point operations per point (41 in this case), and dividing the result by the total elapsed time during one time step. The elapsed time is wall-clock time averaged over 50 time steps. Only the smallest grid will fit on a single processor, so speed-ups for larger grids are computed by comparing with the Mflops for that smallest grid.

The numerical problem solved is the same for each case; the time integration is fully implicit ($\theta = 1$), and the initial values are $T(x, y, z, 0) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ on the unit cube. If the boundary values are kept at zero (Dirichlet boundary conditions), the analytical solution is easily found through separation of variables, i.e. $T(x, y, z, t) = \exp(-\pi^2 t)T(x, y, z, 0)$.

From the three sets of computations (Tables 2–4) it is concluded that the parallel performance of the algorithms generally improves—as expected—when the total grid size increases. It is also clear that the dynamic block-Cartesian decomposition is almost twice as efficient as the static one. The Bruno-Cappello multi-cell decomposition, in turn, is significantly faster than the dynamic block-Cartesian approach, even when only half the number of available nodes is utilized (Bruno-Cappello requires the number of nodes to be a square). Moreover, the grids chosen favor the dynamic block-Cartesian decomposition, which is most efficient for grids that have some small aspect ratio, whereas Bruno-Cappello performs best on a cubic grid. We therefore conclude that the last method is best suited for ADI-type applications.

6 Curvilinear algorithm

Now that the optimal algorithm has been selected for the high-communication ADI-algorithm on a simple rectangular grid, we also apply the Bruno-Cappello decomposition to the solution of the heat equation on a curvilinear grid. It is a straightforward exercise to rewrite equation (2) using the general coordinate transformation $(x, y, z) = (x(\xi, \eta, \zeta), y(\xi, \eta, \zeta), z(\xi, \eta, \zeta))$. The resulting equation is subsequently discretized again, using central differences for all derivatives. In order to enable approximate factorization, mixed second derivatives are all moved to the right hand side, leading to the following (factored) difference scheme:

$$
\begin{aligned}
&(1 - \tfrac{h\theta k}{\rho c}[\delta_\xi J^{-1} g^{\xi\xi} \delta_\xi])(1 - \tfrac{h\theta k}{\rho c}[\delta_\eta J^{-1} g^{\eta\eta} \delta_\eta])(1 - \tfrac{h\theta k}{\rho c}[\delta_\zeta J^{-1} g^{\zeta\zeta} \delta_\zeta])\Delta T = \\
&\tfrac{hk}{\rho c}\Big(\delta_\xi J^{-1} g^{\xi\xi} \delta_\xi T + \delta_\eta J^{-1} g^{\eta\eta} \delta_\eta T + \delta_\zeta J^{-1} g^{\zeta\zeta} \delta_\zeta T + \\
&\delta_\xi (J^{-1}[g^{\xi\eta} \delta_\eta T + g^{\xi\zeta} \delta_\zeta T]) + \delta_\eta (J^{-1}[g^{\eta\zeta} \delta_\xi T + g^{\eta\zeta} \delta_\zeta T]) + \delta_\zeta (J^{-1}[g^{\zeta\xi} \delta_\xi T + g^{\zeta\eta} \delta_\eta T])\Big) \ , \quad (15)
\end{aligned}
$$

where g is the metric tensor and J the determinant of the Jacobian of the transformation.

This scheme involves 150 floating point operations per grid point per iteration, provided g and J are stored for every grid point. The algorithm for doing one time step has to be modified slightly to account for the fact that the difference stencil is no longer a seven-point star on a non-orthogonal grid, but a $3 \times 3 \times 3$-cube with the eight corners excluded. Thus, in order to evaluate a new right hand side, points on the corner of a cell need information from six other processors, instead of just three in the Cartesian-grid case. In order to exchange all necessary boundary data a communication scheme similar to the one outlined in [11] is used, whereby the face data transfer is broken up into three

pairs (east-west, north-south, top-bottom). After the third transfer, all boundary data of the augmented cells has been updated. While each pair is being sent, one third of the right hand side for interior points is computed. This strategy is not as efficient as the one in the Cartesian grid case, where all face data was sent at the same time and the whole right hand side for interior points was computed in one loop, but it still takes advantage of the overlap of communication and computation.

The left hand side matrices for the different approximate factors are recomputed each time step in order to save memory. Here the Bruno-Cappello decomposition offers yet another advantage; during the forward elimination phases the communication of end-of-line data to the next cell can be overlapped with the computation of the next left hand side matrix. This cannot be done in the static block-Cartesian approach, because the messages to be sent are many and small. So in the Bruno-Cappello case the only communication that is not overlapped with computation is that of the solution as it is passed on during the back-substitution phases. This involves only one data item per point of each cell face, as opposed to three during the forward elimination phases.

An additional computational gain is obtained by writing the line solve routines such that the inner loops always run over the first array index of the (intermediate) solution; since all partial line solves within a cell are completely independent, it does not matter if we first finish one line segment and then proceed to the next, or if we do one computation at a time for each line segment within a cell in the direction of increasing first index while keeping the others fixed. This again is not possible using the static block-Cartesian approach, because Gaussian-elimination pipe-lines have to be filled one line segment at a time.

The results of computations done with the thus generalized scheme are presented in Table 5. The program was modified such that some processors were allowed to idle within a hypercube, so that the program could be run on any square number of processors smaller than 128. Speed-up figures refer to the actual number of active processors. If the number of processors is not a power of 2, no useful cell-to-processor mapping can be constructed using gray codes. Consequently, some performance degradation occurs, although this effect is minimized through the overlap of communication and computation. The cases run are selected such that they constitute the biggest grid possible on some number of processors (e.g. a $56\times56\times56$ grid on 4 processors). Interestingly, the increase of the problem size on a fixed number of processors does not always yield a monotonically increasing performance. This may be due to a degeneration of the cache utilization and the increase of memory strides as the problem size grows.

Table 5 shows that the parallel performance of the Bruno-Cappello decomposition degrades relatively slowly for increasing numbers of processors, and that an efficiency of about 75% is feasible on any number of processors, provided the grid is large enough. A maximum performance of 526 Mflops is attained for a 170^3 grid on 121 processors, at an efficiency of 74%.

7 Discussion, summary, and conclusions

Three methods have been investigated for solving ADI-type problems on a MIMD distributed memory parallel computer. The most efficient uses the Bruno-Cappello multi-cell decomposition, which automatically ensures a near-perfect load balance and is easily amenable to overlap of computations and communications —the most important source of reduction of parallel overhead. It also sends the smallest number of messages per iteration, which minimizes communication cost due to high latency, and allows high computational efficiency on individual processors. Solution of the three-dimensional unsteady heat equation in curvilinear coordinates shows good scalability, and performance figures of up to 526 Mflops (double precision) on 121 processors of the Intel iPSC/860 for a large enough grid, even though only 150 floating point operations per grid point are carried out per iteration. The current implementation in C has been extended to include more complex boundary conditions (e.g. adiabatic wall, prescribed time-varying wall temperature or heat flux, wrap-around C-grid, etc.), and it was found that the use of high-level data structures kept the programming complexity as low as that of the static or dynamic block-Cartesian decompositions.

The multi-cell method is expected to offer an even larger relative benefit on the new generation of ring-, mesh- and torus-connected MIMD computers, since their connectivity is less than that of a hypercube, which means that communication distances will increase. Contention-free implementation of the dynamic block-Cartesian decomposition is virtually impossible on these machines, and the static block-Cartesian decomposition will suffer due to increased lengths of message paths of non-overlapped communications.

References

[1] T.H. Pulliam, D.S. Chaussee, *A diagonal form of an implicit approximate factorization algorithm*, Journal of Computational Physics, Vol. 29, p. 1037, 1975

[2] J.S. Ryan, S.K. Weeratunga, *Parallel computation of 3-D Navier-Stokes flowfields for supersonic vehicles*, AIAA Paper 93-0064, 31st Aerospace Sciences Meeting & Exhibit, Reno, NV, January 11-14, 1993

[3] S.L. Johnsson, Y. Saad, M.H. Schultz, *Alternating direction methods on multiprocessors*, SIAM Journal of Scientific and Statistical Computing, vol. 8, No. 5, pp. 686-700, 1987

[4] P.J. Kominsky, *Performance analysis of an implementation of the Beam and Warming implicit factored scheme on the NCube hypercube*, Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 8-10, 1990, IEEE Computer Society Press, Los Alamitos, CA

[5] J. Bruno, P.R. Cappello, *Implementing the Beam and Warming method on the hypercube*, Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Jan. 19-20, 1988

[6] J.C. Yan, P.J. Hontalas, C.E. Fineman, *Instrumentation, performance visualization and debugging tools for multiprocessors*, Proceedings of Technology 2001, vol. 2., pp. 377-385, San Jose, CA, December 4-6, 1991

[7] S.H. Bokhari, *Complete exchange on the iPSC-860*, Technical Report 91-4, ICASE, NASA Langley Research Center, Hampton, VA, 1991

[8] S. Seidel, M-H. Lee, S. Fotedar, *Concurrent bidirectional communication on the Intel iPSC/860 and iPSC/2*, Computer Science Technical Report CS-TR 90-06, Michigan Technological University, Houghton, MI, 1990

[9] N.H. Naik, V.K. Naik, M. Nicoules, *Parallelization of a class of implicit finite difference schemes in computational fluid dynamics*, International Journal of High Speed Computing, Vol. 5, No. 1, pp. 1-50, 1993

[10] T.F. Chan, *On gray code mapping for mesh-FFTs on binary N-cubes*, Technical Report 86.17, RIACS, NASA Ames Research Center, 1986

[11] S.J. Scherr, *Implementation of an explicit Navier-Stokes algorithm on a distributed memory parallel computer*, AIAA Paper 93-0063, 31st Aerospace Sciences Meeting & Exhibit, Reno, NV, January 11-14, 1993

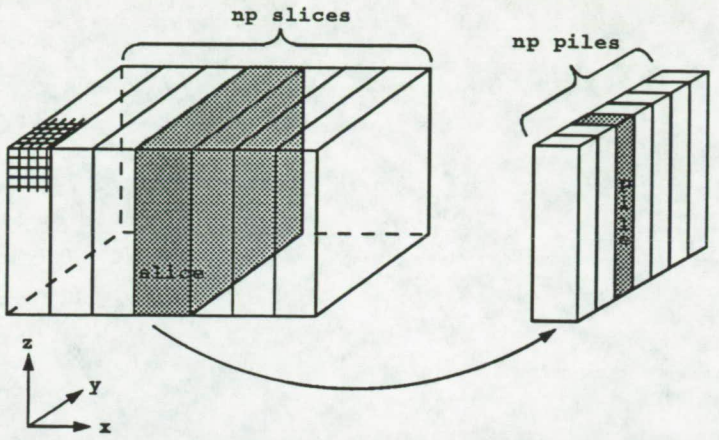


Figure 1: *Storage of slice variable in terms of piles of data*

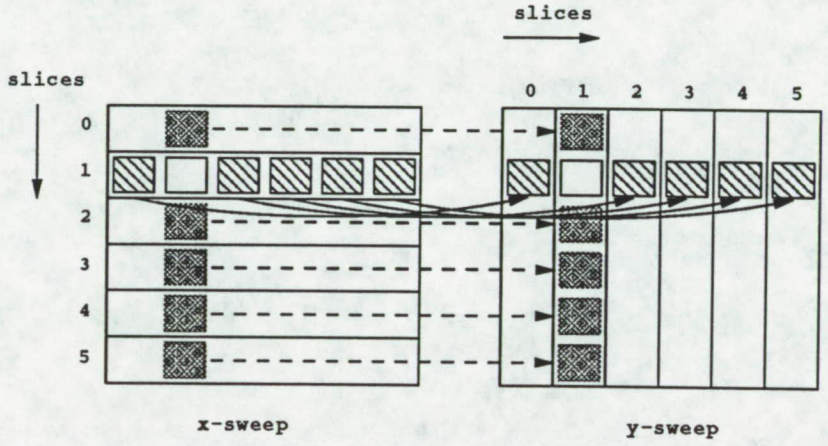


Figure 2: *Exchanging piles of data during change of decomposition*

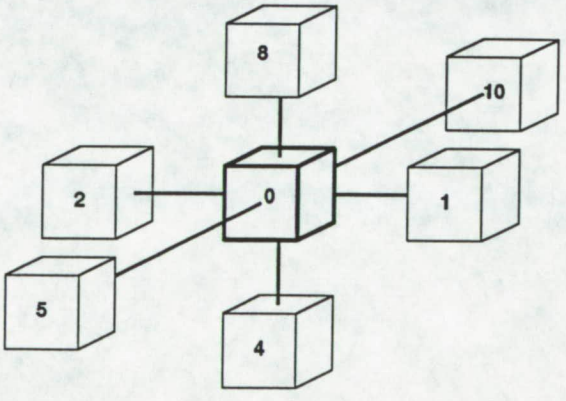


Figure 3: *Neighbors of a cell on processor 0*

	static	dynamic	Bruno-Cappello
communication	$18\frac{n^2}{\sqrt[3]{np^2}}$	$2n^2 + 2\frac{n^3(np-1)}{np^2}$	$18(\sqrt{np}-1)\frac{n^2}{np}$
# messages	$6 + 6\frac{n^2}{\sqrt[3]{np^2}}$	$2np$	$6\sqrt{np}$
storage	$\left(\frac{n}{\sqrt[3]{np}} + 2\right)^3$	$n^2\left(\frac{n}{np} + 2\right)$	$\sqrt{np}\left(\frac{n}{\sqrt{np}} + 2\right)^3$

Table 1: *Storage and communication for all three decompositions*

	Number of processors								
Grid	1	2	4	8	16	32	64	128	
48×48×24	1.00	0.53	0.50	0.28	0.20	0.15	.091	.066	ϕ
	2.94	3.12	5.75	6.55	9.22	14.8	17.2	24.7	Mflops
96×96×48					0.24	0.25	0.17	0.13	ϕ
					11.4	23.4	31.3	49.0	Mflops
192×192×96							0.22	0.19	ϕ
							41.5	71.8	Mflops

Table 2: *Parallel performance of static block-Cartesian decomposition*

	Number of processors								
Grid	1	2	4	8	16	32	64	128	
48×48×24	1.00	0.83	0.72	0.58	0.44	0.27			ϕ
	2.94	4.87	8.43	13.7	20.8	25.2			Mflops
96×96×48				0.69	0.60	0.49	0.29		ϕ
				16.3	28.1	46.0	55.3		Mflops
192×192×96							0.47	0.28	ϕ
							89.0	104.	Mflops

Table 3: *Parallel performance of dynamic block-Cartesian decomposition*

Grid	Number of processors 1	4	16	64	
48×48×24	1.00	0.94	0.60	0.23	ϕ
	2.94	11.0	28.0	42.9	Mflops
96×96×48			0.85	0.54	ϕ
			40.2	101.	Mflops
192×192×96				0.77	ϕ
				144.	Mflops

Table 4: *Parallel performance of Bruno-Cappello decomposition*

Grid	Number of processors 1	4	9	16	25	36	49	64	81	100	121	
36^3	1.00	0.88	0.76	0.67	0.53	0.43	0.34	0.30	0.23	0.18	0.16	ϕ
	5.91	20.7	40.3	63.2	77.6	92.3	99.3	112.	109.	109.	114.	Mflops
56^3		0.89	0.83	0.74	0.70	0.63	0.56	0.52	0.44	0.37	0.32	ϕ
		21.1	44.3	70.4	103.	135.	162.	195.	209.	217.	232.	Mflops
74^3			0.86	0.82	0.76	0.72	0.66	0.63	0.57	0.50	0.45	ϕ
			45.6	77.6	112.	153.	191.	240.	271.	294.	325.	Mflops
89^3				0.83	0.80	0.76	0.72	0.68	0.64	0.59	0.55	ϕ
				78.2	118.	161.	208.	258.	307.	350.	390.	Mflops
102^3					0.80	0.78	0.73	0.72	0.66	0.63	0.55	ϕ
					118.	165.	211.	273.	318.	375.	394.	Mflops
113^3						0.78	0.76	0.72	0.71	0.65	0.64	ϕ
						166.	221.	272.	339.	382.	455.	Mflops
124^3							0.67	0.75	0.71	0.69	0.66	ϕ
							196.	282.	339.	410.	474.	Mflops
138^3								0.75	0.74	0.70	0.69	ϕ
								282.	354.	415.	494.	Mflops
149^3									0.72	0.71	0.70	ϕ
									347.	421.	500.	Mflops
160^3										0.74	0.71	ϕ
										438.	508.	Mflops
170^3											0.74	ϕ
											526.	Mflops

Table 5: *Parallel performance of Bruno-Cappello decomposition for curvilinear case*