3DGRAPE/AL: A Case Study of Parallelization

Elliott E. Schulman
Advanced Management Technology, Inc.
3575 9th Street, Boulder, CO 80304

NAS Technical Report NAS-07-005

July 2007

Abstract

3DGRAPE/AL is a program for generating structured, three-dimensional computational grids used in modeling fluid flow near a solid boundary. These grids are based on a set of elliptic partial differential equations (PDEs), and the program requires that the values of first cell sizes and angles of incidence of the grid lines be specified near a boundary. 3DGRAPE/AL uses a Successive Overrelaxation (SOR) technique and the PDEs are solved by combining SOR with a pipelining algorithm.

The program was originally written for a single-processor system, then re-written for a parallel architecture using Message Passing Interface (MPI) for inter-processor communications. This paper describes the parallelization and optimization procedures used.

1 Introduction

3DGRAPE/AL is a program for generating structured, three-dimensional computational grids used in modeling fluid flow near a solid boundary. These grids are based on a set of elliptic partial differential equations (PDEs), and the program requires that the values of first cell sizes and angles of incidence of the grid lines be specified near a boundary. 3DGRAPE/AL uses a Successive Overrelaxation (SOR) technique and the PDEs are solved by combining SOR with a pipelining algorithm.

The program was originally written for a single-processor system, then re-written for a parallel architecture using Message Passing Interface (MPI) for inter-processor communications. This paper describes the parallelization and optimization procedures used.

The total computational domain is comprised of a set of blocks, generally of different sizes and shapes. A parallel solution entails the assignment of multiple processors to the blocks. How the block is subdivided, as well as the do loop order, determine its efficiency. Various possibilities will be considered, and the configuration which produces the best performance selected.

The pipelining algorithm for the SOR produces scalable results, and will be discussed in detail. Some MPI coding examples illustrating typical communications are also included.

2 Mathematical Model

The mathematical formulation starts with three standard Poisson equations:

$$\begin{aligned} \xi_{xx} + \xi_{yy} + \xi_{zz} &= P(x,y,z) \\ \eta_{xx} + \eta_{yy} + \eta_{zz} &= Q(x,y,z) \\ \zeta_{xx} + \zeta_{yy} + \zeta_{zz} &= R(x,y,z) \end{aligned} \tag{1}$$

In most applications (other than grid generation), x, y, and z are the three spatial coordinates, usually equally spaced when discretized. ξ, η, and ζ are often field variables like temperature or potential functions. The source terms, or forcing functions, P, Q, and R, are functions of x, y, and z. Typically, solutions to this type of problem are smooth–a desirable quality for numerical grids.

These equations can be transformed so that their roles are reversed, with x, y, and z as the dependent variables, and ξ, η, and ζ the independent variables. ξ, η, and ζ form a mutually orthogonal system, and take on consecutive integer values. In Fortran they range from 1 to the number of points along each axis. The smoothness should be preserved in the solutions for the inverse functions. The transformed equation contains all first and second partial derivatives, and is of the general form:

$$A_{11}\mathbf{r}_{\xi\xi} + A_{22}\mathbf{r}_{\eta\eta} + A_{33}\mathbf{r}_{\zeta\zeta} + 2(A_{12}\mathbf{r}_{\xi\eta} + A_{13}\mathbf{r}_{\xi\zeta} + A_{23}\mathbf{r}_{\eta\zeta}) = -J^2(P\mathbf{r}_{\xi} + Q\mathbf{r}_{\eta} + R\mathbf{r}_{\zeta}) \tag{2}$$

There are three equations: `r` is the vector `(x,y,z)`, and the A coefficients are the same for the three equations and consist of fourth order products of first derivatives. The Jacobian, J, is of third order (sixth order squared); this is a highly nonlinear system.

The dependent variables `x`, `y`, and `z` are specified on all boundaries, i.e., Dirichlet boundary conditions. This completely determines the problem and will produce a unique solution.

However, for grid generation, it is also desirable to specify the normal derivatives at the boundaries, i.e., Neumann boundary conditions. This makes for an over-determined system which will, in general, have no solution. The source terms P, Q, and R are added to the equations to aid in satisfying the additional Neumann boundary conditions. However, even though these terms can strongly influence the solution, they are incapable of satisfying them exactly. Instead, values of P, Q, and R are sought which come as close as possible to the exact conditions.

3 Computational Design

3.1 SOR

The numerical solution to equation (2) is to be achieved by an SOR-type sweep performed over all interior points. Specifically, the Gauss-Seidel method is applied, which reduces the residue at the current point to zero using recently updated values at neighboring points.

The forcing terms set at the boundary are used for calculating the forcing terms at interior grid points. Transfinite Interpolation (TFI) is used to perform multiple function blending of source terms into the volume domain. At startup, TFI is used to initialize the grid from its Dirichlet boundary conditions. Combined with exponential damping and anisotropic blending, it is also used to update the forcing terms at each computational point.

For single processors, almost the entire computation time is consumed by SOR. This is due to the large operation count required to evaluate the A coefficients in equation (2)–about 300 floating point multiplications and additions per point. In addition, TFI adds about 100 operations.

Between SOR iterations, several auxiliary functions must be performed:
1. Setting the source terms at the boundaries based on the prescribed grid angles and cell size.
2. Maintaining continuity between blocks at block interfaces.
3. Diagnostics, a history of the computation, tracking its progress, and convergence.

The operation count for these functions is much smaller than for updating the interior grid points. However, the latter scales well, and as the number of processors increases, execution time decreases. The auxiliary functions become relatively more significant, and it is important that they also be parallelized.

3.2 Pipelining

Using second-order accurate finite differences to express all the first and second partial derivatives (including cross derivatives), the computational stencil consists of 19 points. These are the 27 points in the 3 x 3 x 3 cube surrounding a central point, less the 8 corner points (see Figure 1).

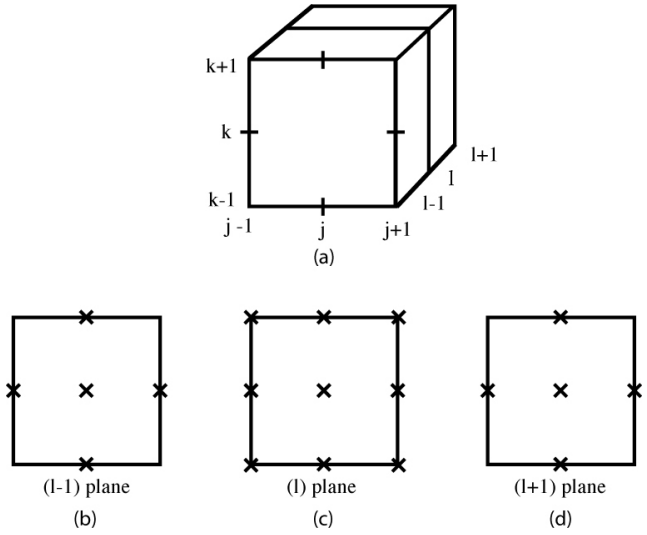


Figure 1: Computational stencil.

The SOR volume updates are executed in three nested do loops–one for each of the three variables, ξ, η, and ζ. Ordering of the loops determines the sequencing of the updates. Following the notation used in the original version, the do loop variables are `j`, `k`, and `l`, corresponding to the ξ, η, and ζ axes. For ease of presentation, we will refer to `j` as the east-west direction, `k` as the north-south, and `l` as the up-down.

There are six possible do loop orders: `l-k-j, l-j-k, k-l-j, k-j-l, j-l-k`, and `j-l-k`. For example, with `l-k-j`, the form is:

```
do l = 11, l2
do k = k1, k2
do j = j1, j2
  update (j,k,l)
enddo
enddo
enddo
```

Each update depends on values at neighboring points, including some that have themselves been updated (the "successive" part of SOR). For `l-k-j` ordering, the point `(j,k,l)` must be preceded by updates at the nine points: `(j,k-1,l-1)`, `(j-1,k,l-1)`, `(j,k,l-1)`, `(j+1,k,l-1)`, `(j,k+1,l-1)`, `(j-1,k-1,l)`, `(j,k-1,l)`, `(j+1,k-1,l)`, and `(j-1,k,l)`. These are a subset of the grid point neighbors that are west, south, and down with respect to the point `(j,k,l)`.

Efficient parallelism requires that all processors be simultaneously active. This dependency implies that points near the upper and northern boundaries must wait for previous updates from a number of grid points. However, it is not necessary to update all points to the west, south, and down. Instead, only its neighbors (and its neighbors' neighbors, etc.) are required. For example, assume that `(j1,k1,l1)` is the first interior point `(j1-1,k1-1`, and `l1-1` are boundaries). Then, point `(j1,k1,l1+1)` depends only on new values at `(j1+1,k1,l1)` and `(j1,k1 +1,l1)`, and not on the entire `l1` plane.

The calculation starts near the origin (the southwest corner on the lowest plane) and then radiates outward, filling successive `j-lines` into `j-k` planes, and eventually, the entire `j-k-l` volume. Assume that the block is divided into sub-volumes, each of which is assigned its own processor. Initially, only one processor is active–the one containing the origin. Eventually, it will reach a point "owned" by another processor. After some necessary communications, this point is ready to be updated, and the second processor becomes active. This can be repeated until eventually all processors become active–a time when maximum parallelism is realized. Later, the first, and then other processors run out of work.

This is a classic pipeline algorithm. The first phase is filling the pipe; starting with one processor, the others are successively activated. This is followed by full utilization where all processors are active. Finally, the pipes are emptied as each processor finishes its part of the computation. A performance analysis of pipelining is discussed in Section 5.

3.3 Processor Assignment

A parallel solution requires the simultaneous computation of points in separate regions. A fundamental decision for 3DGRAPE/AL is how many processors to assign to each block. The parallel code has the flexibility of allowing multiple processors per block, or multiple blocks per processor, or both.

The optimal assignment is one which achieves load balancing–i.e., all processors have approximately the same amount of computational work. It would be inefficient for some processors with small workloads to be idle while waiting for other processors with greater loads to finish. The total volume should be divided such that each processor contains approximately the same number of computational points.

Some possible configurations:
(a) Assign all processors to each block: In this case, each block runs in parallel, but the blocks are executed sequentially.
(b) If the number of processors equals the number of blocks, assign one processor to each block: Each block runs in scalar mode, but the blocks are executed in parallel.
(c) If the number of processors exceeds the number of blocks, one or more processors may be assigned to each block: All blocks run simultaneously–some or all running in parallel over a subset of the processors. Load balancing is improved by assigning additional processors to the larger blocks. Also, multiple blocks may be assigned to a single processor in cases where there are a number of very small blocks.

For situations where there is a large disparity in block sizes, configurations (a) and (b) are inefficient. In (a), there is load balancing, but the smallest blocks will not scale well. For (b), there is no load balancing; the smaller blocks will be idle, waiting for the largest block to finish. This will also be true if the number of processors is a multiple of the number of blocks and distributed equally among the blocks.

3DGRAPE/AL must handle multiple blocks of greatly varying size. The flexibility and generality of (c) are needed for load balancing and scalability.

3.4 Domain Decomposition

There are several basic ways to apportion the sub-volumes in a block among multiple processors. The simplest distribution is to cut the volume into slabs by parallel slices across one of the three axes (see Figure 2a). If cuts are then made across a second axis, the volume consists of a set of rods (Figure 2b). When all axes are divided, the volume is a collection of smaller 3D boxes (Figure 2c).

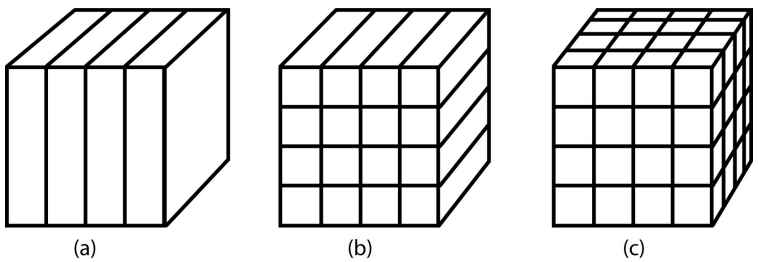


Figure 2: Domain Decompositions.

The optimal choice is determined by the size and shape of the block and the number of processors available. Performance is influenced by cache coherency, communications, and do loop order.

In the early stages of working on 3DGRAPE/AL, a number of tests were performed to see which decomposition was best. It incorporated the pipelining method and, while not rigorous, appeared to show a strong preference for dividing on one axis, i.e., parallel slices. It had the best performance and was the easiest to program.

3.5 Slice Orientation

Each block is divided into a set of parallel slabs, but the orientation of the slices is still to be determined. The location of a variable in the do loop order determines which of the three axes should be divided. For example, consider `l-k-j` loop ordering where `l` is the outer (slow) index and `j` is the inner (fast) index. Let `NPROCS` be the number of processors assigned to a block of size NJ x NK x NL. There are three possibilities:
1. If the domain were to be sliced along the l-axis, then a processor would have to wait for all lower processors to complete most of their work before starting: Between communication calls, `NJ*NK*(NL/NPROCS)` points are processed. In each processor, there is only one send/receive, with the message containing data from `NJ * NK` points. Since a processor does not transfer data until just before becoming idle, only one processor is active at a given time. In fact, this would be an absurd, anti-parallel "solution"–the work total computation is divided among the processors, but they perform sequentially, not simultaneously.
2. Slicing along the j-axis is better: A processor needs to update only `NJ/NPROCS` points between communication calls, implying very small delay before all processors become active. Each processor sends/receives `NK*NL` messages, and each message contains data for only one grid point. However, transferring many small messages is inefficient–communications overhead becomes significant and a detriment to performance and scalability.
3. The best choice is division along `k`, the middle do loop index: In this case, `(NK/NPROCS)` lines of data, each of length `NJ`, are processed between communication calls. Each processor sends/receives `NL` messages with each message containing `NJ` points. This represents an excellent balance between the number of messages and the size of the messages. The delay in activating the next processor is the time it takes to update `(NK/NPROCS) * NJ` points. This is smaller than the total number of points in the processor by a factor of `NL`. For moderately large `NL`, it implies that the pipeline can be filled, with all processors active, in a relatively short time.

On this basis, it is assumed that the slice axis will be the same as that of the second do loop variable.

3.6 Loop Ordering

For 3DGRAPE/AL, as in most other SOR problems integrated out to a steady state, the loop ordering may be important but not critical. Cache coherency can influence the performance, and the convergence rates may depend on the order. The intermediate results may be different, but they all converge to about the same solution.

Thus, the triple nested loop can be performed in any order. The optimal selection for performance can be based on the relative dimensions of the block. It has been established that the splitting

should be along the axis of the second do loop index. It is natural for scalability and load balancing to make this correspond to the largest dimension. The second largest dimension should be the outer loop index since it maximizes the number of fully parallel steps.

Thus, we can let the block dimensions determine both the orientation of the slice as well as the loop ordering. For example, if `j` is the largest dimension and `k` the smallest, ordering of the do loops is `l-j-k`, with splitting along the j-axis.

Often, cache coherency is a major consideration when determining how to subdivide a domain. This implies a preference for unit stride, or equivalently, making `j` the innermost loop. However, in 3DGRAPE/AL, many mathematical operations are needed to update each point, which may make cache coherency less important. In any case, the code contains all six do loop orders, and tests can be run to find the best loop order for a particular block. The convergence rate can be used to determine the best order for the entire calculation.

3.7 Non-Orthogonality

3DGRAPE/AL enables the generation of grids which intersect the body surface at any angle. For most faces on the computational boundary, orthogonality is specified. This constraint can be expressed in terms of derivatives at the boundary. This additional condition is satisfied by adding forcing functions near the boundary.

However, this technique does not do well for non-orthogonal angles. It has previously been found that the solution tends to be unstable, attributable to the stiffness of the equations. Perhaps, it is asking too much of the forcing functions.

For non-orthogonal conditions of the grid at a boundary, the Hilgenstock and White source terms are used. Here, there is no attempt to satisfy the PDE at the boundary. Instead, the angle of the current source terms is evaluated and compared to the desired angle. The source terms are then modified to decrease the difference between the angles. The magnitude of the changes is designed to be gradual, and is based on an empirical formula (using tanh and power functions).

Certainly, this is a non-traditional way of solving a PDE, and the solution it produces requires further explanation. It is important to realize that the angle specified, if non-orthogonal, is never achieved. Thus, the specified angles are "target" angles, and are different from the computed angles. However, the finer the grid, the closer the angles get. This implies that a Hilgenstock solution cannot reach a true steady-state.

As the iterations increase, it is expected that the oscillations will decrease in magnitude until relatively small, but never to zero. There is no unique solution to the problem, but if the oscillations are small, presumably any solution found after many iterations should be acceptable. Grids used for computations must have quality acceptable for accurate flow simulation. However, the quality is rather subjective, and a solution with zero residuals is not required.

3.8 Arrays

Some parallel computer systems, including the Altix, have shared-memory architectures, i.e., any processor can directly access data anywhere, even if it is resident in another processor node. However, MPI does not allow shared-memory access. It is based on a distributed-memory model in which all data is local. If data currently on another processor is required, it must be transferred via explicit message passing.

The total computational domain is a collection of the blocks, and within each block are a number of 3D and 2D arrays. The 3D arrays contain volume data, e.g., the grid coordinates, `x`, `y`, and `z`. The 2D arrays contain data on the six faces, e.g., the boundary forcing terms, `P`, `Q`, and `R`. In practice, the 2D and 3D arrays from all blocks are concatenated into large 1D arrays, one for each variable. The size of each block allocated is based on its dimensions, and an individual block is referenced by its relative offset within the 1D array. On small computer systems, there may be insufficient memory to handle large problem sizes. The memory required can be reduced by assigning storage space only of sub-arrays in the individual processors.

However, if there is sufficient memory, the full arrays can be stored in each processor. Although much of this allocation is redundant, it eases the conversion of scalar to parallel versions and is somewhat easier to program. This storage option was employed in 3DGRAPE/AL.

The coordinate limits of the "active" region in each processor are set by domain decomposition. These become the do loop limits in the parallel code, and only those points in its range are updated.

Surrounding each region are points either on the boundary or "belong" to another processor. In the latter case, these "frontier" points have been transferred from a neighbor processor and are stored in the same array locations.

For example, assume `l-k-j` loop ordering, with splitting along the k-axis. Let `k1` be the initial value of the k loop, i.e., it is the southern-most plane of active points in a processor. If this processor is not on the southern boundary, data is required for points at `k=k1-1`. Currently, these points are resident on another processor, its southern neighbor, on its northern-most plane, `k=k2`. Note that `k1` and `k2` are defined locally, and `k2=k1-1`.

The data is transferred via MPI message passing. From Section 3.5, each message consists of an entire `j-line`. The `j-line` at `(k2,l)` in the neighbor is sent to the current processor where it is stored at the frontier points, `(k1-1,l)`.

3.9 Program Outline

The complete 3DGRAPE/AL program contains several sections that were not parallelized:
1. Data input: With MPI, data read from a file is automatically loaded into all processors.
2. Analysis of results: Various calculations measuring the quality of the grid are run at the conclusion of the SOR iterations. All local arrays are converted into global form, and the original scalar code is used.
3. Output: Under MPI, any processor can write to a file or standard output. The best and simplest way is to have all output be performed from a single processor.

Each of these sections is executed only once, and collectively account for a small fraction of the total execution. The rest, which consists of the bulk of the computation, is amenable to parallelization. A complete iteration step consists of the following parallel procedures and their requisite communications:
1. Update source terms at boundaries: The data from all faces are broadcast to all processors assigned to the block.
2. SOR, including TFI: Lines of data at the processor interface are exchanged between neighboring processors.
3. Inter-block boundary matching: It is preceded by exchanging planes of data near adjoining faces on matching blocks.
4. Diagnostics: Various characteristics of the solution history are evaluated using MPI reduction operations, restricted to processors within each block.

4 SOR Algorithm

The parallelized code has a simple loop structure:

```
do l = 2, ldim-1
do k = k1, k2
  call jline_update (x, y, z, k, l))                                    (3)
enddo
enddo
```

This form is for `l-k-j` ordering, with slicing along the k-axis. Permutations of the characters `j,k,l` in (3) produce the forms for the other five loop orderings. The block is dimensioned `(jdim,kdim,ldim)`, and is split into `NPROCS` slabs, where `NPROCS` is the total number of processors assigned to the block. The processor IDs vary from zero at the southern boundary, to `NPROCS -1` at the northern boundary.

The loop limits, `k1` and `k2`, are local to each processor. Note that `k2` on processor `N` is identical to `k1-1` on processor `N+1`, and `k1` on processor `N+1` is identical to `k2+1` on processor `N`.

The subroutine `jline_update` calculates new values for `x`, `y`, and `z` for all points on a `j-line`, defined as all points for a given `k` and `l` on the line from `j=2` to `jdim-1`. The `j-lines` are the basic computational and communications unit. The code uses only `k` and `l` sweeps, and is essentially a two-dimensional algorithm.

Two types of communications are needed:
1. Let `R(k,l)` represent a `j-line` at `(k,l)`: Then `R(k,l)` depends on `R(k-1,l)` having been updated. If `k=k1`, the latter is on another processor where in local terms, it is the `j-line`, `R(k2,l)`. Immediately after it is updated, `R(k2,l)` must be sent to its northern neighbor. And prior to updating `R(k1,l)`, a processor must have received and stored the `j-line`, `R(k1-1,l)` from its southern neighbor.
2. `R(k,l)` depends on `R(k+1,l-1)` having been updated: If `k=k2`, the latter is on another processor, and is the `j-line R(k1,l-1)`. Immediately after being updated, `R(k1,l)` must be sent to its southern neighbor. And prior to updating `R(k2,l)`, a processor must have received and stored the `j-line R(k2+1,l-1)` from its northern neighbor.

With communications added, the code becomes:

```
do l = 2, ldim  1
do k = k1, k2
  if (k .eq. k1) recv_jline_from_south (R(k1-1,l)
  if (k .eq. k2) recv_jline_from_north (R(k2+1,l-1)

  call jline_update (x, y, z, k, l)

  if (k .eq. k1) send_jline_to_south (R(k1,l)
  if (k .eq. k2) send_jline_to_north (R(k2,l)

enddo
enddo
```

Some additional conditions must be added. Communications are restricted from the first `(0)` and last `(NPROCS-1)` processors, and for extreme values of `k` and `l`. With some slight changes, the final form becomes:

```
do l = 2, ldim  1
  if (not first proc) recv_jline_from_south ( R(k1-1,l) )

  do k = k1, k2
    if (k .eq. k2 .and.  not last proc .and. l .gt. 2)
      recv_jline_from_north ( R(k2+1,l-1)

    call jline_update (x, y, z, k, l)

    if (k1 .eq. k1 .and. not first proc .and. l .lt. ldim - 1)
      send_jline_to_south ( R(k1,l) )
  enddo

  if (not last proc) send_jline_to_north ( R(k2,l) )

enddo
```

On completion, almost all "frontier" points have the current updated values from neighboring processors. The missing `j-lines` are `R(k2+1,ldim-1)`, and it is useful to include them since they will be needed in the next SOR iteration. It costs little to add the following code at the end:

```
if (not first proc) send_jline_to_south ( x*(k1,ldim-1) )
if (not last proc) recv_jline_from_north( R(k2+1,ldim-1)
```

Figure 3 is a complete schematic representation of the algorithm for `kdim=30`, `ldim=6`, `NPTROCS=3`, and arbitrary `jdim`. Note that the planes `k=l`, `k=ldim`, `l=1`, and `l=ldim` are boundaries, and are not updated. The entries are:
`e`: Update of a single `j-line`
`E`: Update of multiple `j_lines`
`S`: Send `j_line` to another processor
`R`: Receive `j_line` from another processor

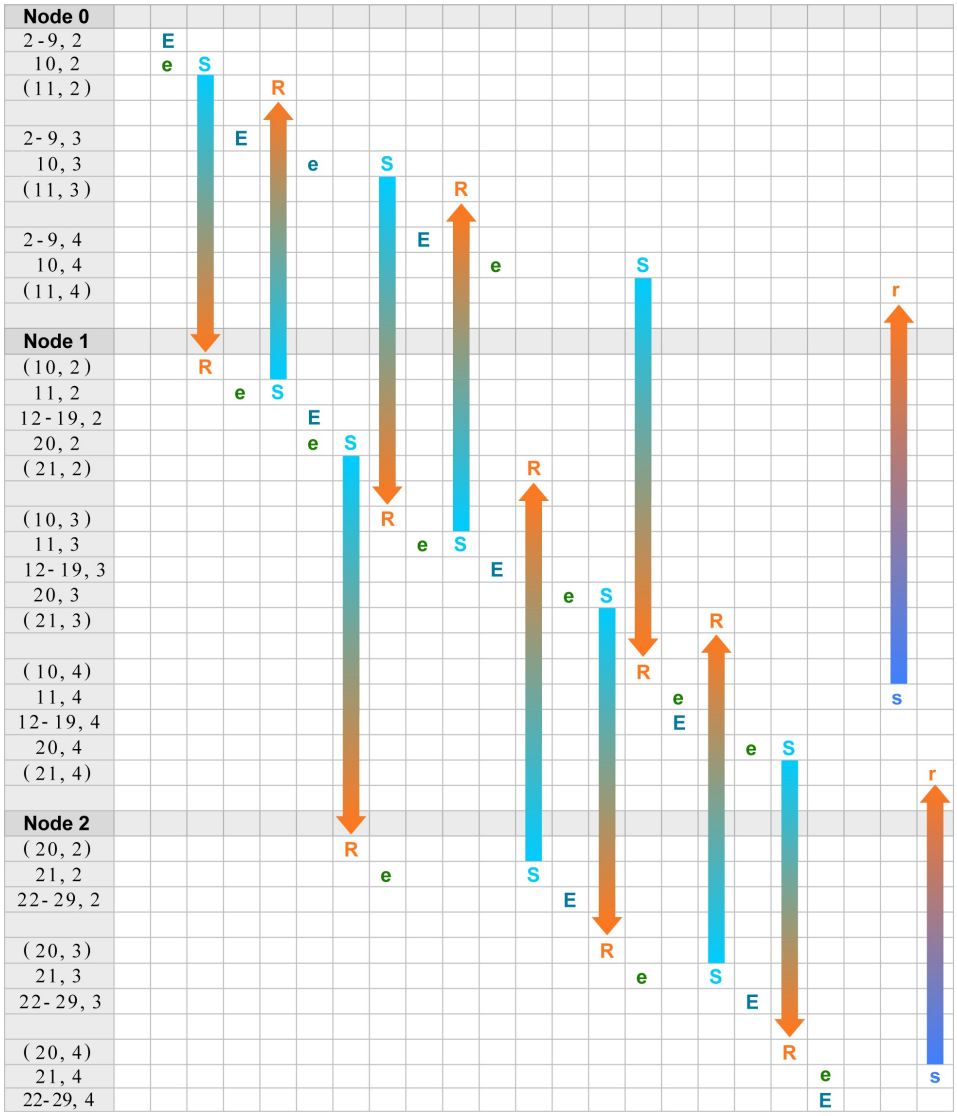


Figure 3: Execution and communications schematic.

At the start, only processor 0 is active. The operations are performed in the following order:
`l=2`:
Processor 0 updates interior points, `k=2` to 9, and the last point, `k=10`. The `j_line (10,2)`, is sent to processor 1.

Processor 1 is idle until it receives the `j_line (10,2)` from processor 0. The first point, `k=11`, is updated and the `j_line(11,2)` is sent to processor 0. The interior points, `k=12` to 19, and the last point, `k=20`, are updated. The `j_line (20,2)` is sent to processor 2.

Processor 2 is idle until it receives `j_line (20,2)` from processor 1. The first point, `k=21`, is updated, and the `j_line (21,2)` is sent to processor 1. The interior points, `k=22` to 29 are updated.

l=3:
Processor 0 updates interior points, k=2 to 9. It then receives j_line (11,2) from processor 1, updates k=10, and sends the j_line(10,3) to processor 1.

Processor 1 receives j_line (10,3) from processor 0, updates k=11, and sends the j_line (11,3) to processor 0. The interior points, k=12 to 19 are updated. It then receives j_line (21,2) from processor 2, updates k=20, and sends the j_line (20,3) to processor 2.

Processor 2 receives j_line (20,3) from processor 1, updates k=21, and sends j_line (21,3) to processor 1. The interior points, k=22 to 29 are updated.

l=4:
Processor 0 updates the interior points, k=2 to 9. It then receives j_line (11,3) from processor 1, updates k=10, and sends j_line (10,4) to processor 1.

Processor 1 receives j_line (10,4) from processor 0 and updates k=11. The interior points, k=12 to 19 are updated. Then it receives j_line (21,3) from processor 2, updates k=20, and sends j_line (20,4) to processor 2.

Processor 2 receives j_line (20,4), and updates k=21 and the interior points , k=22 to 29.

Finally:
Processor 0 receives j_line (11,4) sent from processor 1, and processor 1 receives j_line (21,4) sent from processor 2.

For convenience of presentation, the schematic has aligned the send and receive pairs. But these operations are not necessarily synchronous. For example, processor 1 sends the j_line (11,2) to processor 0 immediately after it is updated. This message is not read and used in processor 0 until a later time–after it has updated multiple interior points. This delay is useful since no time is wasted in waiting for the message to arrive.

The horizontal axis in the schematic (Figure 3) is not a linear timeline, and it does not show which operations are current at a particular time. For example, the interior updating of l=4 on processor 0, l=3 on processor 1, and l=2 on processor 2, are not in the same column. However, the algorithm is designed so that these updates are simultaneous, wholly or partially. This overlap is the essential nature of parallelism.

5 Scalability

The number of points in a block is the product of its dimensions, jdim, kdim, and ldim. Let W be the time to update one point, and T1 the time needed for the whole block, if performed on a single processor. Then, T1 = W * jdim * kdim * ldim. (This is approximate since it assumes that all points, both interior and on the boundary, are computationally equivalent). If the block is divided into NPROCS processors, each processor takes TP = T1/NPROCS to update its portion of the block.

Assume the loop ordering is l-k-j, with the domain sliced on the k-axis. At the beginning, only one processor is active. According to the algorithm, each processor must have updated jdim * (kdim/NPROCS) points before it can pass a j-line to the next processor. It will take

l	iproc = 0	iproc = 1	iproc = 2	iproc = 3
9	8	9	10	11
8	7	8	9	10
7	6	7	8	9
6	5	6	7	8
5	4	5	6	7
4	3	4	5	6
3	2	3	4	5
2	1	2	3	4

Figure 4: Execution simultaneity.

`NPROCS-1` such steps before the last processor can start, and each step take `jdim * (kdim/NPROCS) * W`. Let `TD` be the time elapsed for activating the last processor. Then,

```
TD = (NPROCS-1) * jdim * (kdim/NPROCS) * W
```

The time for the entire parallel calculation, `TN`, is determined by the last processor since it is the last to finish. Then, `TN = TD + TP`, or

```
TN = T1/NPROCS + (NPROCS-1) * jdim * (kdim/NPROCS) * W.
```

Note that `jdim * kdim * W = T1/ldim`. For large `N`, this simplifies to:

```
TN = T1 * (1/NPROCS + 1/ldim)
```, and:

```
speedup = T1/TN = N /(1 + N/ldim).
```

Thus, if `ldim` is much greater than `NPROCS`, we can expect fairly good scalability. It is not until `NPROCS` is comparable to `ldim` that there will be significant drop-off in efficiency.

Figure 4 illustrates the dependency of speedup on `NPROCS` and `ldim`. In the example, there are 8 interior levels, `(ldim=10)`, and `NPROCS=4`. The time unit is the time to update the `jdim * (kdim/NPROCS)` points for each value of `l`. The entries are the time step when the update is calculated–equal values indicate simultaneity. During the first and last `NPROCS-1` steps, not all processors are active. If `ldim>>NPROCS`, this inefficiency will be relatively small.

This analysis is optimistic since it does not take into account the communications time, which could be significant. The algorithm is designed to minimize this effect by reducing both the number and size of message passing.

Between SOR iterations, several operations are performed: setting new source terms at the boundary, imposing block matching, and calculating diagnostics. In scalar mode, they require much less time than SOR. However, as the number of processors is increased, if left unparallelized, they become comparable and even exceed the SOR times. Fortunately, they are amenable to parallelization, and while not as efficient as SOR, do not seriously degrade the overall scalability.

6 MPI Implementation

6.1 Parallel Setup

6.1.1 Initialization

```
call MPI_INIT (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, ntprocs, ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, me, ierr)
```

Each processor has a unique ID given by its global rank. The range is from zero to `ntprocs-1` where `ntprocs` is the total number of processors specified in the `mpirun` command line. The local variable, `me`, is the global rank of the current processor.

6.1.2 Assign Processors to Each Block

If the number of processors exceeds the number of blocks, some or all of the blocks will contain multiple processors. It is also possible for one processor to be assigned multiple blocks. For load balancing, the number of processors assigned to each block, `nnt(iblock)`, is proportional to the size of each block. The mapping of processors to blocks is contained in the array `nodelist(im,iblock)`, where `im` ranges from one to `nnt(iblock)`. The related Boolean array, `isprocinblock(im,iblock)`, indicating whether processor `im` is assigned to `iblock`, is a useful programming aid.

6.1.3 Set Loop Order and Slice Orientation

The execution order in the triple nested SOR loop is determined by the relative dimensions of each block. The domain is sliced along the axis having the largest dimension, and it becomes second in order–the smallest dimension being third in order.

6.1.4 Loop Limits and Neighbors

The slice axis is divided into approximately equal segments, and the end points of each are the local do loop limits. Let `num(i), i = 0` to `nnt(iblock)-1` be the size of each segment. Then, the starting address of each segment, `offset(i)`, is the running sum of `num(i)`. `(offset(0) = 0, offset(i) = offset(i-1) + num(i-1).)`

The first point of each segment, `offset(i) + 1`, is a "frontier" cell, storing values transferred from the lower processor. For the first segment, this is on a boundary and is constant. The starting value of the do loop is thus `offset(i) + 2`. Similarly, the end value is `offset(i) + last1(i)`, where `last(i) = num(i) + 1`. The next point is a frontier cell transferred from the upper processor. For the last segment, this is a boundary point.

The `k` loop limits, `k1` and `k2` (in Section 4) can be expressed in terms of the block and current processor. These loops, and many other `k` loops throughout the code, take the form:

```
do k = koffset(me,iblock) + 2, koffset(me,iblock) + klast(me,iblock)
```

The lower and upper neighbors are determined by their relative order in the nodelist array. For processor `iproc`, they are respectively `iproc-1` and `iproc+1`; with k-splitting, they are referred to as the south and north neighbors.

6.1.5 Communicators

Several parallel functions require communications restricted to only processors in the same block. This includes reduction operations, e.g., block maximum and sum, and intra-block broadcast of boundary data. Communicators, which allow different groups of processors to perform independent work, are well-suited for these tasks. They are an almost intrinsic feature of MPI, and are present in almost all MPI calls.

An MPI group is an ordered set of processor identifiers, intimately associated with communicators and used in their creation. The coding for setting up communicators for each block is:

```
call MPI_COMM_GROUP (MPI_COMM_WORLD, ioldgroup, ierr)

do iblock = 1, nblocks
  call MPI_GROUP_INCL (ioldgroup, nnt(iblock), nodelist(1, iblock),
    newgroup, ierr
  call MPI_COMM_CREATE (MPI_COMM_WORLD, newgroup,
    icomm(iblock), ierr)
enddo
```

This creates an array of communicators, `icomm(iblock)`, for `iblock = 1` to `nblocks`. Each communicator contains the set of `nnt(iblock)` processors assigned to it.

6.2 Message Passing

6.2.1 Send and Receive

Each processor independently updates its own portion of the block's total computational domain. At various places in the program, some or all of this data is needed by another processor. The data is transferred via a communications buffer, which is an argument in the MPI communications calls.

Typically, the data transferred is a subset of an array, e.g., a line segment in SOR, or a planar section in matching block interfaces. In general, a gather operation is required to fill the communications buffer before it is sent. On the receiving end, the data is scattered from the buffer into the local

array. Often, several arrays are to be transferred together, e.g., `x`, `y`, and `z`. Communications overhead is reduced by combining them into a single buffer.

Let `x` be a 3D array a with dimensions `(jdim,kdim,ldim)`. The complete syntax for sending a sub-array of `x`, with lengths `nj`, `nk`, and `nl`, starting at location `(j1,k1,l1)` to processor `ito` in the specified communicator is:

```
subroutine sample_send
dimension x(jdim,kdim,ldim)
dimension buf(*)
dimension icomm(*)
include 'mpif.h'

call gather3d (x(j1,k1,l1), buf, jdim, kdim, nj, nk, nl)
call MPI_SEND (buf, nj*nk*nl, MPI_DOUBLE_PRECISION, ito,
  itag, communicator,, ierr)

end
```

Its counterpart for receiving the message from processor `ifrom`:

```
subroutine sample_receive
dimension x(jdim,kdim,ldim)
dimension buf(*)
include 'mpif.h'
integer status(MPI_STATUS_SIZE)

call MPI_RECV (buf, nj*nk*nl, MPI_DOUBLE_PRECISION, ifrom,
  itag, communicator, status, ierr)
call scatter (x(j1,k1,l1), buf, jdim, kdim, nj, nk, nl)

end
```

The message content is the communications buffer, `buf`, a 1D array large enough to hold the sub-array. The integer variable, `communicator`, can be either:
a) `MPI_COMM_WORLD`, in which `ifrom` and `ito` are global processor IDs
b) `icomm(iblock)`, in which `ifrom` and `ito` are their relative positions within the communicator

The remaining arguments in the MPI calls are:
`itag`: An arbitrary integer value
`status`: An argument required by `MPI_RECV`, containing information about the received message including its size. It is dimensioned by the parameter, `MPI_COMM_SIZE`, defined in `mpif.h`
`ierr`: Error code

6.2.2 Broadcast

The broadcast operation is an efficient method for sharing data across multiple processors. The coding model is:

```
subroutine sample_broadcast
dimension x(jdim,kdim,ldim)
dimension buf(*)
dimension icomm(*)
include 'mpif.h'

if (me .eq. ifrom) then
  call gather (x(j1,k1,l1), buf, jdim, kdim, nj, nk, nl)
endif

call MPI_BCAST (buf, nj*nk*nl, MPI_DOUBLE_PRECISION,
  ifrom, communicator, ierr)

if (me .ne. ifrom) then
  call scatter (x(j1,k1,l1), buf, jdim, kdim, nj, nk, nl)
endif

end
```

If `communicator=MPI_COMM_WORLD`, the `sub_array` is broadcast from processor `ifrom` to all other processors. If `communicator=icomm(iblock)`, the broadcast is restricted to processors in the communicator.

6.2.3 Gather and Scatter

The gather and scatter functions are implemented with the following code:

```
subroutine gather3d (x, buf, jdim, kdim, nj, nk, nl)
dimension x(jdim,kdim,*)
dimension buf(*)

irun = 0
do l = 1, nl
do k = 1, nk
do j = 1, nj
```

```
  buf(irun+j) = x(j,k,l)
enddo
irun = irun + nj
enddo
enddo

end

subroutine scatter3d (x, buf, jdim, kdim, nj, nk, nl)
dimension x(jdim,kdim,*)

irun = 0
do l = 1, nl
do k = 1, nk
do j = 1, nj
  x(j,k,l) = buf(irun+j)
enddo
irun = irun + nj
enddo
enddo

end
```

The compiler should have no difficulty optimizing the subroutines written in this form. Note that they also can be used for 2D arrays by simply setting `nl = 1`, and ignoring `kdim`. In the special case of `nj = 1`, they can be rewritten to eliminate the superfluous inner loop (not shown).

6.2.4 Reduction Operations

The `MPI_REDUCE` function performs certain reduction operations such as sum and maximum value over all processors in a given communicator. For example, the code for summing a scalar over multiple processors is:

```
subroutine sample_sum (x, kcomm)
include 'mpif.h'

call MPI_ALLREDUCE (x, x1, 1, MPI_DOUBLE_PRECISION,
  MPI_SUM, kcomm,  ierr)
x = x1

end
```

MPI_ALLREDUCE stores the result on all processors in the communicator, kcomm. For other reduction functions, e.g., maximum or minimum, substitute MPI_MAX or MPI_MIN for MPI_SUM.

Another function is MPI_MAXLOC (or MPI_MINLOC) for finding both the maximum (or minimum) value and its location. This coding is slightly more complicated:

```
subroutine sample_maxloc (x, jx, kx, lx, kcomm)
include 'mpif.h'
dimension xin(1,2), xout(1,2)

call MPI_COMM_RANK (kcomm, me, ierr)
xin(1,1) = x
xin(1,2) = me

call MPI_ALLREDUCE (xin, xout, 1, MPI_2DOUBLE_PRECISION,
  MPI_MAXLOC, kcomm, ierr)

x = xout(1,1)
mnode = xout(1,2)

call MPI_BCAST (jx, 1, MPI_INTEGER, mnode, kcomm, ierr)
call MPI_BCAST (kx, 1, MPI_INTEGER, mnode, kcomm, ierr)
call MPI_BCAST (lx, 1, MPI_INTEGER, mnode, kcomm, ierr)

end
```

Presumably, prior to calling this subroutine, a scan was performed over the local sub-volume of an unspecified 3D array. It produced, x, the local maximum, and the coordinates of its location, jx, kx, and lx.

Note the MPI data type, MPI_2DOUBLE_PRECISION, a 2D array with the second dimension equal to two. In each processor, xin, the array input to MPI_ALLREDUCE, is filled with the local maximum and processor ID. After reduction, the output array, xout, is filled on all processors with the global maximum and mnode, the processor where the maximum is located. Its coordinates, jx,kx,lx, are then broadcast from mnode to all other processors in kcomm.

6.3 Coding Examples

Virtually all applications employing domain decomposition can utilize these message passing constructs. They are general enough to handle the transfer of any sub-volume of an array–a ubiquitous requirement for parallelism, and specific enough to be applied directly to the code. The following examples show how they are used in critical sections of 3DGRAPE/AL.

6.3.1 SOR

For illustration, consider the case with `l-k-j` ordering of the do loops. Other orders are explicitly programmed and have a similar pattern. In all cases, the arrays are indexed by `j`,`k`,`l`, and an array `x`, defined with dimension `x(jdim,kdim,ldim)`, is referenced by `x(j,k,l)`.

The SOR algorithm requires the transfer of newly computed data between contiguous processors. Specifically, it entails communicating specific one-dimensional subsets (lines) located near the subvolume interfaces. Based on the presented analysis, the `l-k-j` ordering implies the block is sliced along the k-axis, and j-lines are exchanged between neighboring processors.

Subroutine `send_jline`, below, sends the j-line at `k=k0`, `l=l0` in arrays `x`, `y`, and `z` to an arbitrary processor, `ito`. It is a generalization of `send_jline_to_south` and `send_kline_to_north` in the SOR discussion. In that case, `ito` is either the processor immediately to the south or north. The same subroutine can be used in `k,l,j` ordering–here, the division is across the l-axis, and `ito` is the processor in the down or up direction.

```
subroutine send_jline (x, y, z, jdim, kdim, ldim, k0, l0, ito)
dimension x(jdim,kdim,ldim), y(jdim,kdim,ldim), z(jdim.kdim.ldim)
include 'mpif.'

ninc=kdim-2
ntot=3 * ninc

n0 = 0
do k = 2, kdim  1
  buf(n0 + k  1) = x(j,k0,l0)
enddo

n0 = n0 + ninc
do k = 2, kdim  1
  buf(n0 + k  1) = y(j,k0,l0)
enddo

n0 = n0 + ninc
do k = 2, kdim  1
  buf(n0 + k  1) = z(j,k0,l0)
enddo

call MPI_SEND (buf, ntot, MPI_DOUBLE_PRECISION, ito, itag,
  MPI_COMM_WORLD, ierr)

end
```

The j-lines from `x`, `y`, and `z` have been collected into the communications buffer, `buf`, and are sent as a unit. For simplicity and efficiency, the gather and scatter calls have been replaced by explicit inline code.

For every send, there must exist a corresponding receive operation in the target processor. The subroutine `recv_jline` is complementary in form to `send_jline`. The argument, `ifrom`, is the processor source of the data.

```
subroutine recv_kline (x, y, z, jdim, kdim, ldim, k0, l0, ifrom)
dimension x(jdim,kdim,ldim), y(jdim,kdim,ldim), z(jdim.kdim.ldim)
include 'mpif.h'
integer status(MPI_STATUS_SIZE)

ninc=kdim-2
ntot=3 * ninc

call MPI_RECV (buf, ntot, MPI_DOUBLE_PRECISION, ifrom, itag,
  MPI_COMM_WORLD, status, ierr)

n0 = 0
do j = 2, jdim  1
  x(j,k0,l0) = buf(n0 + j  1)
enddo

n0 = n0 + ninc
do j = 2, jdim  1
  y(j,k0,l0) = buf(n0 + j  1)
enddo

n0 = n0 + ninc
do j = 2, jdim  1
  z(j,k0,l0) = buf(n0 + j  1)
enddo

end
```

6.3.2 TFI

Between complete SOR sweeps, the source terms `p`, `q`, and `r` are updated on the faces. If the face is "controlled," this update can be based on the orthogonality condition or Hilgenstock forcing. The transfinite interpolation formula is used for extending the source terms smoothly into the interior. The calculation is based on values of the source terms at 26 points on the 6 faces: 1 at the interior points of each face, 12 points on the edges, and 8 at the corners.

This implies that the source terms from all points on every face must be available to all processors in the block. Updating on the faces is performed in parallel, with the results stored locally. The subroutine `sharepqr` copies the source data `p1`,`q1`,`r1`, from each processor containing all or part of the designated face to all processors in the block.

It is illustrated for a face at `l=1` or at `l=ldim`. For storage within arrays, they are referenced as `iface=5` and `iface=6`, respectively. As before, `k=splitting` is assumed.

```
 subroutine sharepqr (iblock, iface)

 (iface = 5 or iface = 6)

 do ia = 1, nnt(iblock)
   iproc = nodelist(ia,iblock)
   nj = jdim
   k1 = koffset(iproc, iblock) + 2
   k2 = koffset(iproc,iblock) + klast(iproc,iblock)
   nk = k2  k1 +1

   nt = nj * nk
   ip1 = 1
   ip2 = ip1 + nt
   ip3 = ip2 + nt
   ntot= 3 * nt

   iaddr= idisface(iblock,iface) + (k1  1) * jdim + 1

   if (me. eq. iproc) then
     call gather3d (p1(iaddr), buf(ip1), jdim, kdim, nj, nk, 1)
     call gather3d (q1(iaddr), buf(ip2), jdim, kdim, nj, nk, 1)
     call gather3d (r1,(iaddr), buf(ip3), jdim, kdim, nj, nk, 1)
   endif

   call MPI_BCAST (buf, ntot, MPI_DOUBLE_PRECISION,
     ia  1, icomm(iblock))

   if (me. ne. iproc) then
     call scatter3d (p1(iaddr), buf(ip1), jdim, kdim, nj, nk, 1)
     call scatter3d (q1(iaddr), buf(ip2), jdim, kdim, nj, nk, 1)
     call scatter3d (r1,(iaddr), buf(ip3), jdim, kdim, nj, nk, 1)
   endif
 enddo

 end
```

The source terms, `p1`, `q1`, and `r1` are 1D arrays, and are a concatenation of 2D arrays, collectively containing data from all faces on all blocks. The `idsface` array contains the offset of each face, and is used in setting the starting location, `iaddr`, of the 2D data being transferred. Note that a processor's local ID is given its order within `nodelist`, starting with 0, hence the `ia-1` in `MPI_BCAST`.

7 Conclusion

This paper has presented the various options available for subdividing a computational block among multiple processors. For the SOR method used in 3DGRAPE/AL, the domain is decomposed into a set of parallel slabs. For optimal performance, the relative dimensions of the block determine the slice orientation and do-loop order.

In actual runs of the parallel program, all possibilities for various block shapes were tested, and this analysis confirmed. For a representative large-scale code, the speed-up achieved asymptotes to about 10:1 compared to that of a single processor.

8 Acknowledgements

My thanks to Steve Alter of NASA Langley Research Center for introducing me to the vagaries of grid generation, and for his encouragement and assistance in writing this paper.

Work supported by the NASA Advanced Supercomputing Division under Task Order NNA05AC20T (ITOP II Contract GS-09F-0028Z/TO NNA05AC20T) with Advanced Management Technology Incorporated (AMTI).