Research in Parallel Algorithms and Software for Computational Aerosciences

Neal D. Domel[1,2]
Report NAS-96-004, April, 1996
domelnd@lfwc.lockheed.com

Abstract

Phase I is complete for the development of a Computational Fluid Dynamics parallel code with automatic grid generation and adaptation for the Euler analysis of flow over complex geometries. SPLITFLOW, an unstructured Cartesian grid code developed at Lockheed Martin Tactical Aircraft Systems, has been modified for a distributed memory/massively parallel computing environment. The parallel code is operational on an SGI network, Cray J90 and C90 vector machines, SGI Power Challenge, and Cray T3D and IBM SP2 massively parallel machines. Parallel Virtual Machine (PVM) is the message passing protocol for portability to various architectures. A domain decomposition technique was developed which enforces dynamic load balancing to improve solution speed and memory requirements. A host/node algorithm distributes the tasks. The solver parallelizes very well, and scales with the number of processors. Partially parallelized and non-parallelized tasks consume most of the wall clock time in a very fine grain environment. Timing comparisons on a Cray C90 demonstrate that Parallel SPLITFLOW runs 2.4 times faster on 8 processors than its non-parallel counterpart autotasked over 8 processors.

---

[1] Lockheed Martin Tactical Aircraft Systems, Fort Worth, Texas
[2] Funded by NASA Ames Research Center under contract NAS2-14057

1.0 Introduction

Since their origin, computer architectures have evolved from single-instruction, single data (SISD) processors. A significant improvement in speed accompanied the development of single-instruction, multiple-data (SIMD) processors, often called vector processors. The next step in architecture development offering further improvement in speed is the multiple-instruction, multiple-data (MIMD) processor, or parallel processor (Ref. 1).

Parallel vector machines are heavily used in industry for Computational Fluid Dynamics (CFD). These machines, like the Cray C90, generally have a few (around 8) tightly coupled processors all sharing the same memory. This type of platform is often used for code autotasking, which distributes instructions to the processors. Although this approach improves the execution speed, the improvement often fails to scale with the number of processors. Scalability is more likely to be achieved if the program is coded such that each processor independently executes its instructions on its uniquely designated piece of memory. This "distributed memory" philosophy allows the processors to be much more efficiently utilized. However, this requires more coding effort than the shared memory autotasked approach.

Massively parallel machines generally have many (over 100) processors with distributed memory (i.e., each processor has its own memory which is not directly linked to other processors) (Ref. 2). Because of the large number of processors, these machines have the potential for dramatic improvements in overall execution speed. However, the benefit they offer has gone largely unappreciated for years because of the high level of effort required to enable a pre-existing code to run in a distributed memory environment (Ref. 3). However, the complexity of problems demanding simulation continues to outpace the speed improvements of conventional hardware. Thus, the necessity of faster turn-around has been influential in the development of massively parallel hardware and software.

SPLITFLOW is a CFD code developed at Lockheed Martin Tactical Aircraft Systems (LMTAS). This code automatically builds an unstructured Cartesian grid around the geometry of interest. The Euler equations are then solved on the grid (the Navier-Stokes version is under development). The grid is periodically refined such that various features may be resolved with more grid cells (Ref. 4). Thus, the user is not required to manually generate a structured grid. SPLITFLOW has proven to be a powerful tool for quickly predicting properties around extremely complex geometries. The popularity of SPLITFLOW has warranted an effort to improve its speed beyond that possible with autotasking on multiprocessor computer. Parallel computing is the next logical step in accomplishing this objective. The solver, and some non-solver tasks, have the potential for nearly ideal parallelization.

The objectives of parallelizing SPLITFLOW:

1) Ability to use massively parallel machine
   -Access to machines like the IBM SP2 is expected to increase.
2) Improve performance on conventional coarse grain machines
   -Access to the Cray C90 and J90 is common and heavily used.
3) Network several workstations together for a single application
   -SGI workstations are abundant at LMTAS, and may be dedicated at night.
4) Maintain modularity and consistency with non-parallel SPLITFLOW
   -Developments in SPLITFLOW must be easily implemented in parallel SPLITFLOW.

Many architectures have specialized high performance libraries for passing messages between processors. A single message passing package common to most machines is necessary to satisfy the considerations listed above. Thus, Parallel Virtual Machine (PVM) was selected as the message passing protocol (Ref. 5).

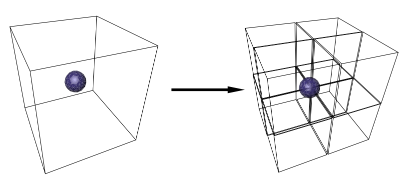
The addition of multiple moving bodies is planned for a future development phase.

2.0 General Description of SPLITFLOW

Code formulation

Cartesian grid techniques have been developed as a means of fast automatic grid generation (Ref. 6,7). The methods generally utilize nested cell subdivision to generate the computational mesh around geometries. The grid generation is generally automatic and can handle extremely complex geometries. SPLITFLOW is a finite-volume Euler/Navier-Stokes code which utilizes cubical cells. Attributes of SPLITFLOW include automatic cell division and domain boundary decomposition from a computer-aided design (CAD) surface definition. The code is upwind in the inviscid regions, and flux limiters are available to reduce oscillations near shocks. Inviscid regions utilize Cartesian grid topology, while a prismatic grid generator (under development) is used for viscous regions. As shown in Figure 2.1, the Cartesian grid method produces rapid subdivision of root cells, and a known cell aspect ratio for ease of reconstruction of face information. Solution grid adaptation is included within the code, using several user-selected functions. The code offers extremely fast user setup times, on the order of 20 to 40 minutes.

Figure 2.1: Cell Dividing into 8 Children



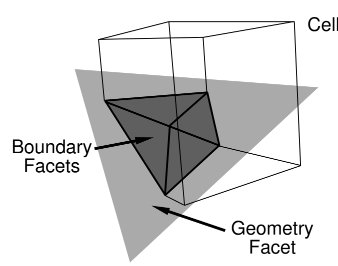
Surface Representation

The surface geometry is input as a triangulated surface mesh. This mesh is provided by the engineering computer aided design (CAD) package used to define the configuration. By interfacing with the CAD package directly, conversion of geometry to CFD surface definitions is eliminated. The surface in the CAD file is defined as a list of X, Y, and Z coordinates and a connectivity in the form of three node numbers corresponding to the indices of the forming points of each triangle making up the surface. The geometry facets are oriented such that the surface normals point into the computational domain. Subsets of the facets can be grouped together in a series of ASCII files, so that in the assembly of the faces of the grid described below, each can be associated with a particular boundary condition type such as no-slip, symmetry, characteristic slip wall, etc.

Grid Generation

The construction of the Cartesian grids within SPLITFLOW begins with a boundary face file consisting of triangular facets describing all 6 faces of the grid, including the body surface. For viscous analysis (not included in this report) the prismatic grid generator would be employed to build an initial grid suitable for viscous analysis. The Cartesian grid would then use the outer layer of the prismatic grid as its boundary surface. As shown in Figure 2.2, SPLITFLOW finds the intersection between the Cartesian cells at the boundary and the surface faces, and constructs smaller facets in the intersection plane which are used to reconstruct each cut boundary cell. Thus, the boundary cells contain portions of the surface boundary and inherently capture the surface resolution provided by the user in the boundary face file. The number of subtriangles (boundary facets) constructed within each surface facet range on the order of 4 to 10, but all the subtriangles are coplanar with the original facet (geometry facet) provided in the face file. Each boundary subtriangle is connected to a unique Cartesian boundary cell. The size of the Cartesian cells, and resulting number of grid levels, is determined by the size of the facets provided in the face file. Some control is provided by setting a scale factor (bndscale) for the facets on each face, and a minimum Cartesian cell length term (dxyzmin), in the input deck.

Figure 2.2: Boundary Cutting Process



An octree data structure is used to store information for each Cartesian cell during the recursive grid generation process. A subdivided cell produces eight new offspring cells, as shown in Figure 2.1. The parent is retained in the grid after the subdivision. The information stored for each cell consists of the global index of the parent cell, the global indices of the eight children that may exist and the grid level of the cell. The grid 'level' refers to the number of times the root cell has been recursively subdivided to create this particular child. Since the position of each offspring cell (in relation to its parent) is predetermined in the subdivision process (due to the Cartesian topology) the neighboring cell indices can quickly be determined. In addition, many of the search procedures make efficient use of the octree data structure.

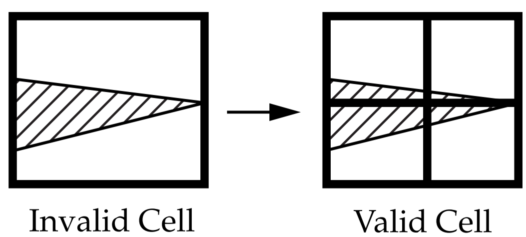
<u>Initial Grid Refinement</u>

The initial Cartesian grid is generated based on the resolution of the triangulation of each of the surfaces in the boundary surface file. All surfaces are triangulated, including the far-field boundaries. Generally, the surface of the vehicle of interest will contain a much denser mesh of triangular facets than far-field boundaries. The root cell defined by the boundary face file is termed grid level 1, and is subdivided in the X, Y, and Z directions resulting in eight offspring cells at grid level 2. Figure 2.1 shows the subdivision. Each offspring cell is recursively subdivided based on a cell length-scale criterion. The length scale of each cell is compared with the length scale of all the geometry facets that are contained within the cell or are touched by the cell. The cell length scale is defined as the length of the sides of the cell. The length scale of the geometry facet can be defined as the average length of the three sides of the facet. If a particular cell is larger than the facet length scale multiplied by a user-specified scale factor, the cell is subdivided. This process continues down each branch of the octree data structure until all cells without offspring satisfy the length scale criterion.

During the subdivision process, grid smoothing constraints are enforced. No cell can have more than four neighbors on any side. This is equivalent to limiting the differences in grid levels between adjacent cells to one. This constraint is enforced so that the octree data structure can be used to rapidly determine the neighbor information of the cells on all grid levels. Any refinement resulting from this constraint quickly propagates through the grid. The resulting grid has fine resolution cells near the bodies, and coarse resolution cells in the far field.

The robustness of the grid is checked. Cartesian grid generation may result in invalid cells which are divided into multiple distinct volumes near thin sharp regions. SPLITFLOW uses an area summing approach to sum the X, Y, and Z area components of the boundary facets in each cell that lies along the boundary. First, if any of the area components sum to zero while the maximum magnitude of the area component is non-zero, then the cell may be an invalid cell. Second, if large negative and positive summations occur then the cell may be invalid. These checks assure that invalid cells are eliminated. Figure 2.3 illustrates refinement to “fix” an invalid cell.

Figure 2.3: Refinement of an Invalid Cell

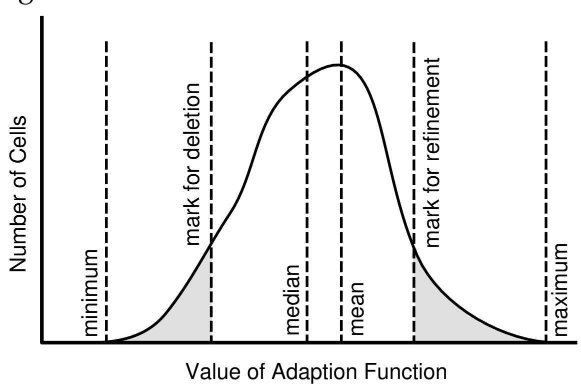


Grid Adaptation

Once the volume grid has been created based on the face geometry, cells within the volume grid are subdivided additionally during the solution to various levels, depending on the local flowfield gradients. SPLITFLOW contains gradient computations of several functions such as static pressure or Mach number. These functions are selected by the user, and are used to refine or coarsen the grid. The gradient of each chosen adaptation function is computed across the cell and multiplied by a length scale. This length scale is calculated from the cell volume and is then adjusted by an exponent based on a user selected term. This gives some control for supersonic flows where shocks cause such high gradients that the cells near the shock tend to dominate the adaptation function statistics.

The statistical approach used for assessing the need for grid adaptation is shown in Figure 2.4. This approach dramatically reduces the requirement for user decisions about grid adaptation. Unlike other Cartesian grid schemes, no min/max cell size or tolerance needs to be defined, and no user-defined “sequence” of adaptation (such as a number of cycles each having several grid levels within each cycle). Rather, the actual gradient information is computed across every cell in the entire domain. Physically-based adaptation functions (selected by the user), such as pressure or velocity, are calculated using these gradients. The user simply defines the thresholds of the values on the adaptation function at which cells will be marked for refinement or deletion. These thresholds (called gradmn and gradmx) are applied to the statistics of the adaptation function(s). Refinement occurs automatically for cells which exceed the threshold. Cells which fall below the lower threshold of the adaptation function are marked for deletion. Deletion (which coarsens the grid) occurs for cells in which all 8 children have been marked. The objective is to create a uniform value of the adaptation function across all the cells and avoid either ‘hot spots’ in which large gradients exist, or regions of minimal gradient where cells could be removed without disturbing the solution. After refinement has been completed (or the target number of cells is reached), the grid is smoothed such that adjacent cells differ by no more than one grid level.

Figure 2.4: Statistics Used in Grid Refinement



The user input file contains the grid generation cell resolution terms (bndscale and dxyzmin) which allow control of the minimum Cartesian cell size. The adaptation of the volume grid to flowfield gradients is controlled by the terms gradmx and gradmn in the input file.

As the solution proceeds, refinement events occur periodically. Cells are added or deleted, and the residual spikes then falls. The general trend for the residual is to progressively drop, and generally 3-4 orders of magnitude of convergence of the L2 norm of the residual are achieved.

Numerical Formulation

The governing equations are the Reynold's averaged, compressible Navier-Stokes equations. The discrete-integral form of the equations for an arbitrarily-shaped cell is given as:

$$\frac{\Omega}{\Delta t}\Delta Q + \sum_{m=1}^{ns} (F_i - F_v)_m \bullet (n_m \sigma_m)$$

where *ns* is the number of sides of the cell (*ns* may be high for a cell with complicated boundary cuts). The flux *F* and vector *Q* of unknowns are from the conventional conservation-law formulation. The cell volume is represented by Ω and Δt is the time step. The outward-pointing unit normal vector for face *m* is n_m and the surface area is given by σ_m. The inviscid flux for face *m* is denoted F_i, and the viscous flux as F_v. The viscous terms are only calculated in the developmental version of SPLITFLOW which uses the prismatic grid near the solid surfaces. The version of SPLITFLOW which has been released uses a Cartesian grid for Euler applications only.

A steady-state solution to the governing equations is obtained by using an implicit time marching scheme. Upwind fluxes are used for the inviscid terms, and central differences are used for the viscous terms. A point-wise implicit time integration scheme with sub-iterations is used to advance the solution. The numerical form of the implicit equation is:

$$\left[\frac{\Omega}{\Delta t}I + \sum_{m}^{ns}\left(\frac{\delta F}{\delta Q_c}\right)_m\right](\Delta Q_c^s) = \left(-\left[\sum_{m}^{ns}\left(\frac{\delta F}{\delta Q_n}\right)_m\right]\right)\Delta Q_n^{s-1} - (Res)^{N-1}$$

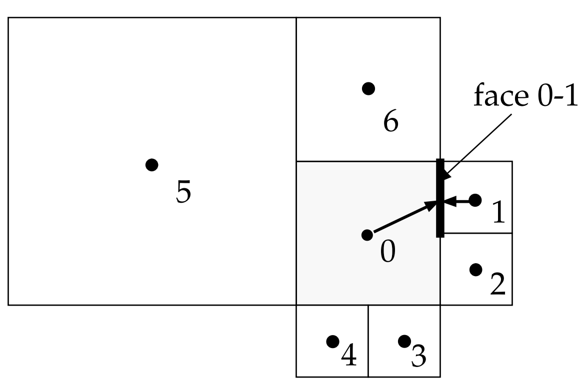
where *c* is the cell of interest, and *n* indicates the neighbor of *c* which resides across face *m*. *Res* is the residual vector computed as the sum of the fluxes over the cell. *I* is the identity matrix. The current time level and sub-iteration level are designated with *N* and *s*, respectively.

The flux Jacobians are the inviscid Jacobians consistent with Roe's scheme, assuming no extrapolation of data. By using the δQ's from the previous sub-iteration for the neighbor cells and adding the influence to the right-hand side, the equations require a block inversion of a 5X5 matrix for each cell. The inverted matrix is computed during the first sub-iteration and stored for use in subsequent sub-iterations. Typically, 10 to 20 sub-iterations are used to converge the implicit equation at each time level. Sub-iteration convergence is monitored by the code.

The Courant-Friedrichs-Lewy (CFL) number is automatically adjusted by the code, depending on the sub-iteration convergence characteristics. CFL numbers on the order of 5 or more are possible for most problems.

The inviscid fluxes are computed using Roe's approximate Riemann solver. Second-order steady-state accuracy is achieved by extrapolating from each direction to define the state on each side of a cell face. The flux is then calculated by Roe-averaging these two states. These extrapolated states are a bit more complicated to calculate in the octree grid than with a typical structured grid. Figure 2.5 shows a cell with different sized neighbors. The state at face 0-1 is extrapolated from the centroid of cell 0 by combining the gradient information at point 0 with the state interpolated between points 0 and 1. They are combined by removing the influence of point 1 from the gradient at point 0, which was calculated from the conditions at the centroid of cell 0 and all of its neighbors (including cell 1). For a grid with one neighbor per side, this is equivalent to conventional linear extrapolation resulting in a second-order differencing scheme. This is <u>not</u> equivalent to a Fromm scheme, which would preserve some influence of point 1 in the state on the 0 side of face 0-1.

Figure 2.5: Extrapolation to Cell Face



A minmod or superbee flux limiter is used to reduce the oscillations near discontinuities, and the entropy fix of Harten (Ref. 8) is used to prevent non-physical expansion shocks.

User Work-load

The time required to set up a problem is generally 20 to 40 minutes. The avoidance of volume grid generation and the simplicity of construction of face grids are seminal features of SPLITFLOW. Also, the addition of new surface geometry is easily accomplished, such as a new tail or modified body shape.The steps of user involvement in creation of a SPLITFLOW grid are listed below:

1) The user determines the level of surface resolution using the computer-aided design (CAD) system. This surface definition is made up of a number of triangular facets.
2) The outer boundaries of the domain are defined, and a symmetry plane is constructed by running LMTAS software tools which read the outer boundary points and the centerline of the CAD surface file to generate a faceted triangulated symmetry plane. The user also makes simple ASCII files of the outer boundary faces (consisting of large triangles containing the corner points of the domain).
3) The boundary faces are then assembled into a total file using an LMTAS software tool, 'spfbnd'. This boundary file is one of the input files to SPLITFLOW.
4) The user generates a namelist file containing flow conditions, grid adaptation parameters, surface integration reference terms and requested print data such as surface pressures.
5) The user runs SPLITFLOW.

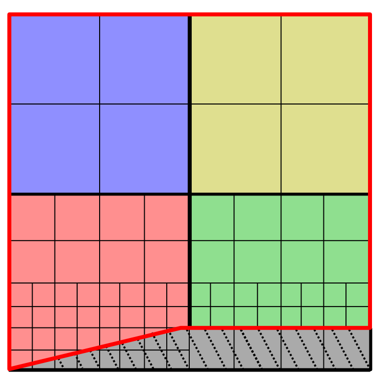
3.0 Approaches Considered for Domain Decomposition

The parallelization of SPLITFLOW causes the overall solution to be divided into several small semi-independent jobs, rather than one large job. The best performance results when these small jobs run simultaneously with computational and memory requirements distributed evenly among them (load balanced). Two approaches for decomposing the problem were considered for best accomplishing this:

1) Decompose domain based on octree data structure:

The problem is divided into pieces according to the first (or second) generation of octree children. This approach allows the initial division of the root cell to occur in one of the jobs. Each first generation child is then sent to a computational node where further grid generation may be performed in parallel with other nodes. Solver tasks may then be performed on the final grid domain residing locally on any node. Figure 3.1 shows a 2-dimensional wedge example divided into 4 subdomains.

Figure 3.1: Octree Decomposition



Advantages:

a) Decomposition of the global domain is very simple.

b) Nearly all tasks (except writing global output files) are parallelized to some degree.

Disadvantages:

a) Memory requirement for any one node may be very high due to the uneven final distribution of grid cells among the nodes. Because any one node may contain more than its “fair share” of grid cells, each node job must be dimensioned accordingly.

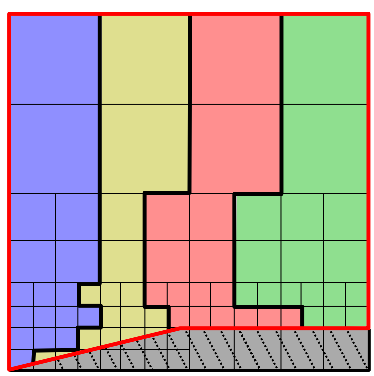
b) Time required to run a problem may not decrease significantly due to the poor load balancing. If one node contains most of the grid cells, then it will take more time to perform its calculations while the other nodes wait.

c) The number of nodes onto which the domain is divided must be a power of two in order to simply distribute the problem onto the nodes.

2) Decompose domain into sub-domains of equal cell number:

The host job generates the grid. The active cells without kids are sorted according to the X (or Y or Z)-coordinate of one corner. The cells are evenly distributed to the nodes. Boundary Cells may be sorted and distributed separately for more uniform memory requirements. Certain grid generation and post-processing tasks may be parallelized separately. Figure 3.2 shows the previous 2-D example decomposed according to the sorted cells.

Figure 3.2: Sorted-Cell Decomposition



Advantages:

a) Memory requirement on each node is more uniform and proportional to the subdomain size.

b) Computational load is well balanced for the parallelized tasks.

c) Any number of nodes may be used.

d) Future development may result in better parallelization of non-solver tasks.

Disadvantages:

a) Grid generation is performed primarily on one node.

b) Domain decomposition is more complicated.

c) Interfaces between subdomains may be large and complex.

Approach 2 was selected as the philosophy to adopt when parallelizing SPLITFLOW because of its superior properties of load balancing memory and CPU requirements.

In a typical problem, the number of boundary facets is approximately twice as high as the number of grid cells. These boundary facets tend to be concentrated in regions of complicated geometry. Experience with a simple airfoil case has shown that a grid divided into subdomains with a uniform cell distribution and low interface areas may have very unevenly distributed boundary facets. Even among the boundary cells, the number of boundary facets may vary tremendously (i.e., one boundary cell may have fifty boundary facets while another has only one). Because SPLITFLOW has several working arrays which are dimensioned to the number of boundary facets, the memory requirement of the code depends more heavily upon the number of boundary facets than the number of grid cells. If the boundary facet distribution is not considered, then the memory demand among the processes may vary by an order of magnitude. Therefore, load balancing for memory is a strong function of boundary facet distribution. However, load balancing for CPU time is a strong function of grid cell distribution (although boundary facets are a small factor in the CPU requirement). Thus, an even distribution of grid cells and boundary facets is mandatory for the benefits of distributed parallel computing to be realized on the computing platforms readily available to LMTAS.

Approach 2 has a higher interface overhead than approach 1. Three penalties are associated with interface cells. First, the additional memory requirement in SPLITFLOW is quite small unless the sum of the interface and non-interface cells exceeds the number of boundary facets, in which case the working arrays must be dimensioned to the larger number. Second, the CPU time required by the solver for an interface cell is approximately one sixth that for a non-interface cell because only one of its six faces usually requires flux calculations. Third, the PVM message passing buffers and libraries can demand significant system time and memory when the number of subdomains (and therefore interface cells) is large.

The decomposition methods discussed in the public literature were not considered because most are intended for standard unstructured grids, and do not address these issues. Also, many of them use recursive bisection and are best suited for a decomposition where the number of subdomains is limited to a power of two. The freedom to decompose into any number of subdomains is a desirable feature for applications at LMTAS. In the future, more sophisticated ways of decomposing the domain may be developed with smaller interfaces, and which address the concerns mentioned above.

4.0 Modifying SPLITFLOW for Parallelization

Actual process of parallelizing SPLITFLOW required the following steps:

1) Identify subroutines required for the solver to work on a subdomain.
2) Group these solver subroutines for use in the "node" code.
3) Keep grid generation tasks and file writing in the "host" code.
4) Identify data transfer requirements from the global domain to the subdomain.
5) Develop an efficient domain-decomposition technique.
6) Develop working versions of "host" and "node" codes.
7) Continue with parallelization of other tasks.

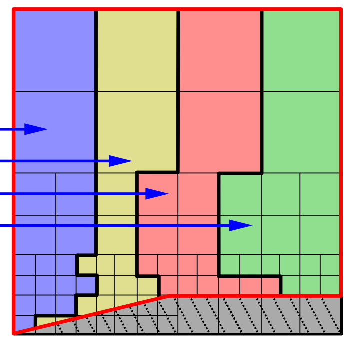
<u>Domain Decomposition</u>

The most important feature which the node code must possess, besides its ability to run in parallel with copies of itself, is that its memory requirement must be proportional to the computational size of the subdomain. That is, if a particular subdomain has one fourth of the grid cells of the global domain, then the memory requirement of the node code should be approximately one fourth that of the non-parallel version of SPLITFLOW. While this may seem like an obvious and minor point, it can cause problems because it forces the grid cells and boundary facets to be evenly grouped and renumbered into subdomains for use in the node code. Consequently, the octree data structure is very difficult to preserve when family lines to first generation cells are disrupted to reside on different processors.

The first step in decomposing the computational domain is to sort the active cells without children according to the minimum X-value of their vertices. The user may opt to sort on the Y or Z value. The sorted cells are then separated into boundary cells and interior cells. (A boundary cell is simply a cell which contains at least one boundary facet.) The boundary cells are distributed such that the boundary facets are fairly evenly divided among the nodes. The interior cells are divided such that the overall number of cells (boundary + interior) on each node is uniform. This procedure produces even distributions of cells and boundary facets for any number of nodes, thereby achieving a load balance. The cells, boundary facets, and neighbor information must be reindexed to correspond to the local grid on each processor. A neighbor which resides on a different node must be treated specially as an interface cell, requiring inter-processor data transfers. Figure 4.1 tabulates the load balance information for the 2-D wedge example. For illustrative purposes, all boundary cells are assumed to contain the same number of boundary facets. Thus, an even distribution of boundary cells is equivalent to an even distribution boundary facets.

Figure 4.1: Decomposition Load Balance

# active cells	# boundary cells	# interface cells
17	8	9
17	8	16
17	8	15
17	8	8

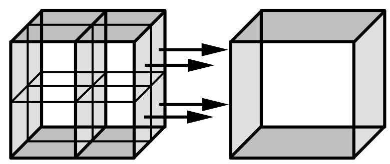


Neighbor Cell Definition

The octree data structure is used extensively for defining the relationship a cell has with other cells. Each cell carries with it several arrays which contain the indices of its parent, children, and neighboring cells. Because adjacent grid cells are allowed to be no more than one grid level apart, each cell may have up to four neighbors on each side. In non-parallel SPLITFLOW, when a cell has four neighbors on a side, then the index of the parent of those four cells is stored. When necessary, that neighbor is checked for children, and information from the appropriate children is retrieved.

In order to avoid the need to know parent and child information on the nodes, the neighbor data structure must be modified. In an old version of parallel SPLITFLOW, it was initially modified to store the indices of four neighbors per side of every cell. This resulted in rather extensive modifications to several subroutines, causing a deviation from non-parallel SPLITFLOW, and consuming a significant amount of additional memory. However, further inspection of the problem revealed that adequate information may be obtained by storing only the pointers from small cells to large cells (or equally sized cells). If a particular side of a cell has multiple neighbors, then, rather than storing the index of the parent, a negative number is stored which serves as a special flag indicating that the neighbors on that side are of a lower grid level. Different negative flags have various meanings indicating whether that cell is a boundary cell, or outside the computational domain, etc. Thus, the neighbor pointer data structure is very similar to that of non-parallel SPLITFLOW, and only active cells without children need to be present on any processor, and full octree information is unnecessary on the nodes. Figure 4.2 shows an example of the one-way pointers which SPLITFLOW requires.

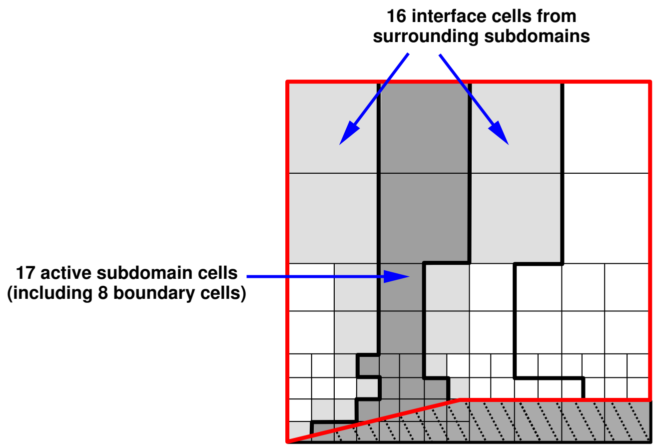
Figure 4.2: One-Way Neighbor Pointers



<u>Interface Cell Definition</u>

Interface cells are cells which must be imported from other nodes because they are adjacent to the subdomain cells assigned to the local node. These cells require inter-processor communication for transferring and updating the interface cell information. Interface cells are found by searching the neighbor pointers for a neighbor residing on another node. Cells point only to larger or equally sized cells, and not vice versa. Thus, when an extra-nodal neighbor index is detected, then that neighbor must be imported, and the cell doing the pointing must be exported as an interface cell to the other node. The pointwise implicit solver requires that the interface cells have updated convergence information every sub-iteration, and an updated solution every iteration. Although the solution is not computed locally for interface cells, the interface updates are required for gradient and flux calculations to be accurate for the subdomain cells. The node code must possess adequate memory resources to store the assigned subdomain cells plus the interface cells. The percentage of cells which are interface cells generally increases with the number of nodes. Figure 4.3 illustrates the computational cells and interface cells for one of the subdomains in the 2-D wedge example.

Figure 4.3: Interface Cell Determination



Parallelizing the Solver

Because SPLITFLOW generally spends a majority of its time in the solver (over 75%), the parallelization efforts have been concentrated on this task. The application of the load balancing techniques, described previously, render a solver with a nearly ideal speed-up over its non-parallel counterpart.

Parallelization of non-Solver Tasks

The non-solver tasks which are currently partially parallelized are: grid refinements, domain decomposition, post-processing, and grid metric calculations. The grid refinement tasks which are performed on the nodes are the calculations of the grid adaptation functions and statistics. The sorting of adaptation functions and octree grid manipulation is performed on the host.

The domain decomposition is coded such that the host code sorts the cells into subdomains, and sends them to the nodes. The nodes determine the interface cells and construct the arrays for transferring interface information among the nodes.

Post-processing consists of the integration of forces over surfaces, and the determination of conditions along a surface contour defining its intersection with a user specified plane. Both of these tasks require averaging boundary facet information onto the appropriate geometry facets. The nodes partially average and integrate within their local subdomains. The host code collects and sums these subdomain values for the global values.

The grid metrics consist of boundary facet areas, unit normal vectors associated with boundary facets, cell volumes, and uncut cell wall areas. The host code determines the boundary facet vertices, areas and vectors. The nodes then use the boundary facet information for calculating the volumes and uncut cell wall areas.

Non-Parallelized Tasks

As mentioned above, the boundary facets are determined on the host. Calculation of the boundary facets is part of the grid generation and refinement process, and relies heavily upon the octree data structure in the search for the intersection between grid cells and geometry facets. The shape of this intersection is required for finding the boundary facet vertices. The determination of this intersection is computationally intensive and involves searching and checking various combinations of conditions. The facet area calculation is rather trivial once the vertices are determined. Because boundary facet determination is part of the grid generation process, it cannot be postponed until the after grid is generated (as is the case with the other grid metrics). The host currently performs all of the boundary facet determination because the octree data structure is not available on the nodes.

Ideas for rewriting the logic to separate and parallelize the non-octree tasks have been discussed, but none have been implemented to date because of planned upgrades to non-parallel SPLITFLOW which involve this task. The production version of SPLITFLOW is the non-parallel version, where most new features and upgrades tend to originate. These upgrades are then incorporated into parallel SPLITFLOW. The current logic in non-parallel SPLITFLOW for boundary facet calculation is likely to undergo a major rewrite for application to multiple moving bodies (an upgrade planned for Phase II of SPLITFLOW parallelization). Boundary facet vertex determination is a very time-consuming part of the grid generation and refinement process, and its parallelization will undoubtedly improve the performance of SPLITFLOW in a massively parallel environment where grid generation tasks consume a large portion of the wall clock time.

Although a non-parallelized task is performed on a single processor, the other processors need not be idle. Such is the case with writing the global output files. The host may be occupied with writing these large files while the nodes are continuing to iterate in the solver. The nodes send convergence information to the host during every iteration. However, the asynchronous nature of PVM transmissions permits the message sender (nodes) to continue executing even though the message may be waiting in a buffer until the message receiver (host) is ready to access it. Several messages from the nodes may collect in this way while the host is occupied with writing out the restart and plot files. The overall wall clock time improves when solving and writing are performed simultaneously. Therefore, normal operation will be in this mode. However, for an accurate breakdown of the wall clock time spent performing various tasks, they must not overlap. Thus, for research purposes when time accounting is of interest, the code may be run in a mode where the nodes are held idle until the host completes the file writing, a very time-consuming task on some architectures.

The run-time behavior is determined in SPLITFLOW by activating (or deactivating) a feature which causes messages to be sent from the host to the nodes at the beginning of every iteration. Enabling this feature also allows the user to stop the code prematurely by editing a “stop-check” file, which is opened and read during every iteration. The nodes are forced to wait for a signal from the host indicating whether or not the current iteration is the last one.

5.0 Performance of Parallel SPLITFLOW

Parallel SPLITFLOW was run on several different platforms. The first two platforms were used for debugging only. Next, two coarse grain shared memory machines are discussed. Last are two massively parallel machines:

1) SGI workstation network at LMTAS (for debugging)
2) Cray J90 at LMTAS (for debugging)
3) SGI Power Challenge at the National Aerospace Simulator (NAS)
4) Cray C90 at Cray Research
5) Cray T3D massively parallel machine at Cray Research
6) IBM SP2 massively parallel machine at NAS

SGI at LMTAS

The vast majority of the developmental work was performed on a network of five SGI workstations including two Personal Irises, two Indigo 2's and an Indy. These machines were used only to debug the parallel code, not for timings. The debugging case was a very small supersonic double wedge which generally had less than 2000 grid cells.

Cray J90 at LMTAS

The Cray J90 was used to debug the code for Cray architecture. Each of the 8 processors on the J90 has a theoretical speed of 200 MFLOPS (Ref. 9).

Test Case for Timing Comparisons

Timing comparisons were performed with parallel SPLITFLOW on the Modular Transonic Vortex Interaction (MTVI) geometry (described in more detail in the validation section) for 99 iterations, with one grid refinement after 50 iterations. Global restart and plot files were written during refinements and at completion. The initial solution, which included the initial grid with all cells initialized to freestream, was read in from a restart file. The generation of this initial grid is unparallelized, and required 255 seconds on one processor on the C90. The number of computational grid cells grew from its initial size of around 110,000 cells and 560,000 boundary facets to its post-refinement size of 151,000 cells and 570,000 boundary facets. The number of grid cells in the full octree data structure grew from approximately 170,000 to 208,000. These numbers varied slightly between architectures due to tolerances in the boundary facet calculations.

Table 5.1 shows the memory requirement of SPLITFLOW on the Cray J90 and the IBM SP2. The host and node codes may be compared with the non-PVM code on the Cray, whose shared memory allows either the PVM or non-PVM approach to be employed. However, the SP2 only allows the PVM approach because of its distributed memory architecture. Generally, on the shared memory machines, the host and node codes were dimensioned such that the sum of the memory requirements of the host and all copies of the node code would fit on the machine. Thus, the memory per node code varied with the number of copies being used, and often exceeded that required to run the actual problem. On the distributed memory/massively parallel machines, however, the host and node codes are dimensioned to fill the local memory of the dedicated processors.

Table 5.1: SPLITFLOW Memory Requirement

SPLITFLOW Code	Number of Cells	Number of Boundary Facets	Required memory (Mbytes)
Cray J90 non-PVM	250,000	600,000	615
Cray J90 host	250,000	600,000	223
Cray J90 node	150,000	300,000	265
IBM SP2 host	700,000	1,400,000	420
IBM SP2 node	70,000	120,000	120

Cray C90 at Cray Research

Each of the 8 processors on the C90 has a theoretical speed of 1000 MFLOPS (Ref. 10). Because this platform represents the hardware available to the typical SPLITFLOW user at LMTAS, it is the most important test bed for parallel SPLITFLOW. It allowed a comparison of non-parallel SPLITFLOW, autotasked over several processors, with parallel SPLITFLOW using Cray's production version of PVM to link several processors.

Tables 5.2a and 5.2b show the timing breakdown of several tasks for parallel SPLITFLOW and autotasked SPLITFLOW. Each run includes three domain decompositions: one for initialization, and two during the grid refinement (one for cell deletion, and one for cell addition). The decomposition time decreases as the number of nodes increases from 1 to 7, but increases slightly for 8 nodes. The solver time, as expected, decreases sharply with the number of nodes. Grid refinement, the most time-consuming non-solver task, requires an amount of time independent of the number of nodes. However, the parallel code is about 35% faster at this task than the autotasked code. Thus, a non-scalable benefit is realized in the partial parallelization of the refinement tasks.

Table 5.2a: Cray C90 Timings for Parallel SPLITFLOW (seconds)

Number of Nodes	Domain Decomposition	Solver	Grid Refinement	File Writing	Overall
1	43.4	1186.0	81.8	17.3	1409.2
2	28.4	607.8	80.5	17.3	794.2
3	23.7	418.5	81.0	17.4	594.5
4	20.9	323.4	80.7	18.5	494.5
7	17.7	222.1	80.8	17.6	387.0
8	18.6	208.8	81.2	21.5	379.4

Table 5.2b: Cray C90 Timings for Autotasked SPLITFLOW (seconds)

Number of Nodes	Solver	Grid Refinement	File Writing	Overall
1	1241.5	130.5	11.6	1651.7
2	830.6	122.6	12.3	1231.0
3	671.1	121.4	11.7	1069.2
4	610.7	121.6	12.4	1009.7
8	511.9	121.4	14.5	920.0

Although the overall wall clock times for the single node PVM case and the single CPU non-PVM case were expected to be approximately equal, the parallel case was about 15% faster. Several factors contribute to the speed difference. First, the logic coded for certain tasks was modified for more efficient parallelization. Second, the parallel version has more efficient memory utilization. The host code has much smaller working arrays on which to perform gathering and scattering operations. Third, the subdomain cells are sorted and reindexed such that neighbors reside near each other in core memory, as well as physical space. Thus, when the neighbors of cell i must be accessed, each neighbor's index will be close to i. This results in reduced bank conflicts and faster memory utilization. Finally, some functional parallelism exists on the host and node. For example, the node could start its next iteration while the host is still writing output (because the stop-check feature is deactivated). However, this particular factor is considered negligible on the C90 because it spends a small percentage of its time writing files. The speed of the C90 on parallel SPLITFLOW was estimated at 220 MFLOPS per processor.

The C90 is the only platform where the stop-check feature was deactivated. This feature was turned off so the host job could share a processor with one of the node jobs for the 8-node case. The host job only needed to be active during start-up, grid refinements, and shut-down. The host could "sleep" through the iterations and allow the messages from the nodes to collect until it "woke up" and received them. Because only one job is allowed to be awake on a processor, the node job which resides on the host's processor must sleep while the host is awake. Thus, some penalty is expected, but that penalty would have been greater if the node had to receive a stop-check message from the host every iteration. The penalty was small enough that the overall performance improved when the number of nodes increased from 7 to 8.

Figure 5.1 shows results of the timing comparison. The speed-up of the solver, and the overall run are plotted. The speed-ups are relative to the single-node non-PVM results. The solver time is calculated on the host as the time it spends waiting for convergence history information from the nodes. Included in the solver time is the message passing among the nodes for interface cell information during the sub-iterations. The solver, as expected, scales fairly well with the number of processors. However, the overall performance is degraded somewhat because the grid refinements are not highly parallelized at this time. The curve of Amdahl's law was calculated from the equation (Ref. 11):

$$Speed - Up = \frac{1}{R + \frac{(1-R)}{N}}$$

where R is the fraction of the time which cannot be parallelized, and N is the number of processors. For this case, R was calculated as the fraction of time spent writing files only, since this task is currently unparallelized.

Figure 5.1 also shows the percentage of time spent in the solver. As this value decreases, the effect of further parallelization decreases because the solver has received most of the parallelization effort to date. The last plot shows the overall percentage of interface cells. It is calculated from:

$$percentage = 100 \times \frac{(interface)}{(interface + computational)}$$

<u>SGI Power Challenge at NAS</u>

The SGI Power Challenge has 8 R8000 processors, each with a clock speed of 90 MHz and a theoretical speed of 360 MFLOPS (Ref. 12). The machine used for most of the cases had 2 Gbytes of memory. This platform provides a second coarse grain shared memory test bed.

Timings were obtained for the test case on the SGI with 1, 2, 3, 4, 7, and 8 processors. The stop-check feature was activated for the SGI cases. Timings were also obtained for the non-parallel code autotasked over 2, 3, 4, and, 8 processors. A single processor non-PVM case was not obtained because of the difficulty in accessing a machine with a 90 MHz R8000 processor for an adequate period of time.

Tables 5.3a and 5.3b show the timing breakdown of several tasks for parallel SPLITFLOW and the autotasked version of SPLITFLOW. The time required for domain decomposition shows a general trend of decreasing with the number of nodes. The solver time, of course, decreases with the number of nodes. The grid refinement time is almost perfectly constant and consumes 1.3% of the overall time for the single processor case. The time spent writing files is fairly constant at 1.5% of the single node time.

Table 5.3a: SGI Timings for Parallel SPLITFLOW (seconds)

Number of Nodes	Domain Decomposition	Solver	Grid Refinement	File Writing	Overall
1	115	10567	144	162	11097
2	88	5357	141	171	5853
3	78	3613	139	176	4107
4	59	2855	139	174	3337
7	65	1860	140	176	2381
8	52	1731	141	188	2242

Table 5.3b: SGI Timings for Autotasked SPLITFLOW (seconds)

Number of Nodes	Solver	Grid Refinement	File Writing	Overall
2	6173	438	85	7098
3	4784	438	87	5703
4	4118	436	86	5045
8	3055	491	94	4076

Figure 5.2 shows the performance for the MTVI test case. All speed-ups are relative to the single-node PVM case. The scalability is fairly linear for the solver and the overall solution because the solver consumes the overwhelming majority of the clock time, even for the 8-node job. The autotasked version on the SGI shows better scalability than the C90 autotasked results, although the PVM version is clearly faster. The percentages of interface cells differ slightly from the C90 because of the sorting of cells which, except for round-off, have the same X-value. The SGI Power Challenge, like the Cray C90, demonstrated that parallel SPLITFLOW performs well in a coarse grain environment. The Power Challenge performed at approximately 30 MFLOPS per processor on parallel SPLITFLOW.

Cray T3D massively parallel machine at Cray Research

Each of the 256 nodes on the T3D contains 64 Mbytes of local memory, and a DEC Alpha chip with a theoretical speed of 150 MFLOPS (Ref. 13). Its architecture requires the number of dedicated processors to be a power of 2 (i.e., 2, 4, 8, 16, etc.). A Cray YMP with 52 Mwords of memory serves as the front end.

Three cases were successfully run for timing comparisons with the production version of PVM. They were run on 16, 32, and 64 nodes. At least 16 nodes are required for this problem on the T3D because of the memory capacity on each node. The stop-check feature was activated for this case for a more accurate accounting of the solver time.

Table 5.4 shows the timings of several tasks in the parallel code.The T3D was the only platform tested where the decomposition time exceeded that of the grid refinement. The number of seconds required for decomposing and refining was fairly constant as the number of nodes increased. The solver time, again, decreases as the number of nodes increases. The file writing time, which is usually independent of the number of nodes, increased with the number of nodes on the T3D. Swapping may have been occurring during this task. Like all parallel platforms, the T3D must spend time allocating memory for message passing. However, this time became increasingly large when the nodes sent their subdomain solutions (large transmissions) to the host for writing a global restart file. Newer optimized versions of PVM may reduce this factor.

Table 5.4: Cray T3D Timings for Parallel SPLITFLOW (seconds)

Number of Nodes	Domain Decomposition	Solver	Grid Refinement	File Writing	Overall
16	198.2	3096.2	144.9	78.6	3628.5
32	204.6	1499.3	144.4	251.8	2214.8
64	209.2	876.3	152.0	344.3	1736.5

Figure 5.3 shows the performance on the T3D. All speed-ups are relative to the case with the smallest number of nodes (16 nodes). The T3D showed a superlinear speed-up when the number of nodes was doubled from 16 to

32. This is possibly due to the entire problem fitting in cache memory. Doubling again to 64 nodes shows a drop off in scalability. Figure 5.3 shows that the 64-node case had a rather high percentage of interface cells (nearly 50%). The interface/PVM overhead is becoming significant within the solver. Parallel SPLITFLOW performed on the T3D at an estimated 5 MFLOPS per processor.

The T3D's low memory per node results in the use of many processors, even for medium sized cases. Thus, the number of interface cells is likely to be high for realistic problems. This increases the number of inter-nodal messages, each of which may require time for memory allocation. Thus, the T3D is suited for running large numbers of parallel processors with small messages transferred among them.

IBM SP2 Massively Parallel Machine at NAS

This platform proved to be the critical machine for testing massively parallel performance, and for calculating final solutions in the validation cases. The SP2 has 160 nodes, each containing 128 Mbytes of local memory (some have 512 Mbytes), and a RS6000 processor with a theoretical speed of 265 MFLOPS (Ref. 14). The RS6000 achieves this limit with a clock speed of 66 MHz and the ability to perform 4 floating point operations per clock cycle.

Although the configuration and memory capacity per processor would have allowed this case to run with 6 or 7 nodes. The smallest number chosen for this comparison was 8 because it is a power of 2, and provides a more familiar starting point for studying the effect of doubling the number of nodes. The stop-check feature was activated for this case.

Table 5.5 shows the timings of several tasks in the parallel code. This is the only platform where the solver time was reduced to such a small fraction of the overall execution time. The time spent decomposing does not show a dependence on the number of nodes. However, the grid refinement time shows a slight decrease as the number of nodes increases. The occasional writing out of restart files to the disk, proved to be more time-consuming on the SP2 than on the other platforms. Although the file writing time shows a slight decrease with larger numbers of nodes, this trend is generally not repeatable. The SP2 had the largest variation in timings for repeated runs. However, this machine was always loaded with multiple users. Only particular nodes were dedicated to running these cases. Thus, i/o conflicts with other users may have introduced some variation in the timings. Also, system bugs and PVM bugs often caused job delays or failures. These problems are being addressed by the staff at NAS and IBM.

Table 5.5: IBM SP2 Timings for Parallel SPLITFLOW (seconds)

Number of Nodes	Domain Decomposition	Solver	Grid Refinement	File Writing	Overall
8	22.3	1272.3	370.3	536.7	2433.9
16	22.4	692.8	363.0	533.2	1848.4
32	33.7	388.0	303.8	491.4	1449.4

Figure 5.4 shows the performance for 99 steps on the MTVI. While the solver shows good parallelization on the SP2, the non-parallelized tasks degraded performance. Due to the large amount of time consumed by file writing, the overall benefit of fine grain parallelization was smaller on the SP2 than the T3D. However, the SP2's higher theoretical MFLOPS rating resulted in faster overall performance. The SP2 performed at approximately 18 MFLOPS per processor on parallel SPLITFLOW.

The SP2 allows any number of processors to be used, and the host code is generally run on a high memory processor (512 Mbytes). However, this memory limit on the host imposes a ceiling of around 700,000 cells on the current version of parallel SPLITFLOW. While this is plenty for most Euler problems of immediate concern, it is not adequate for all of them. Thus, efforts are ongoing to reduce the memory requirement of the host code.

Expected Application at LMTAS

The current state of development of parallel SPLITFLOW makes it well suited for a coarse grain parallel environment (about 10 processors). This environment may be constructed by the average near-term user at LMTAS by networking together available workstations, or by using a multiprocessor shared memory machine.

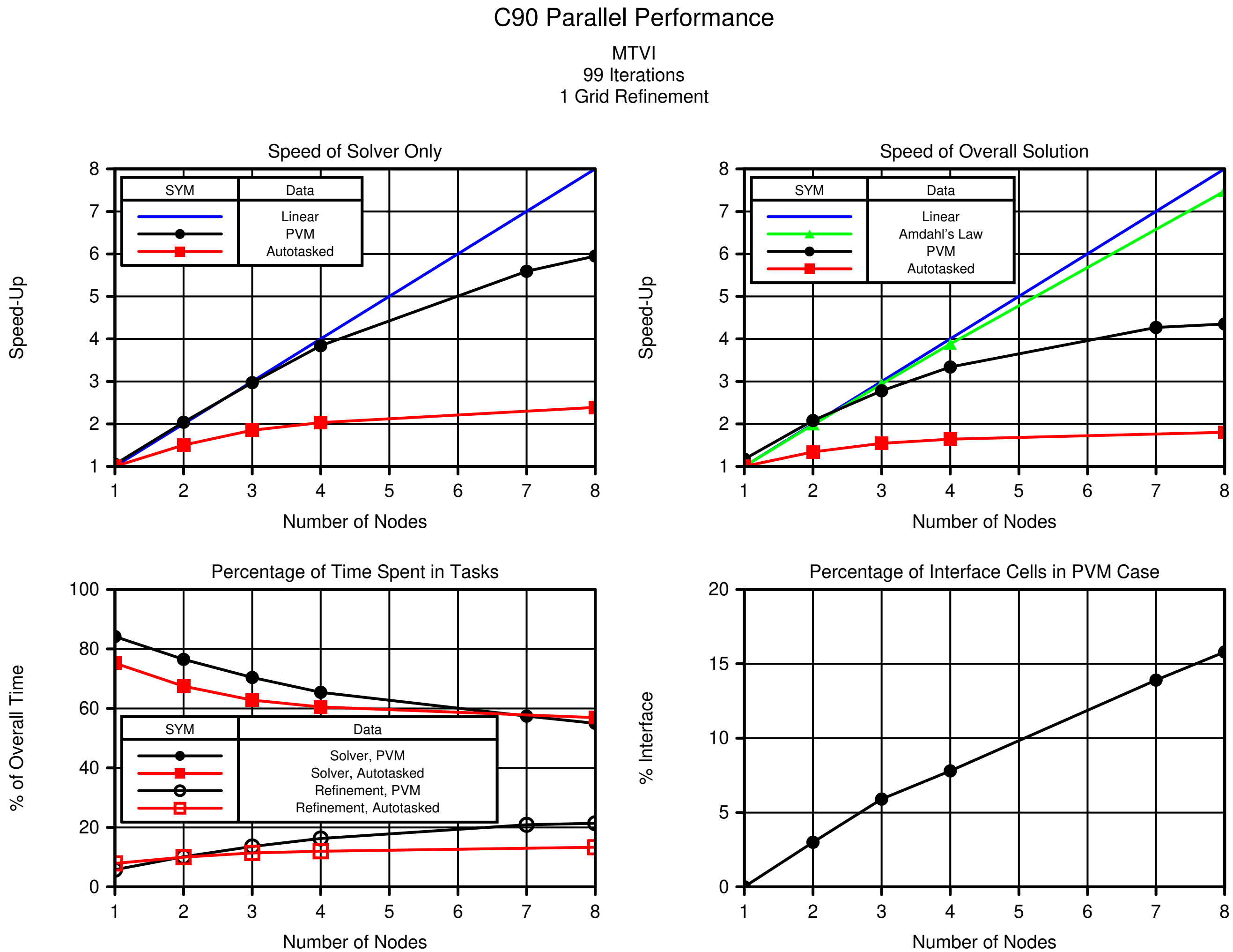


Figure 5.1: Timing Comparisons on the C90

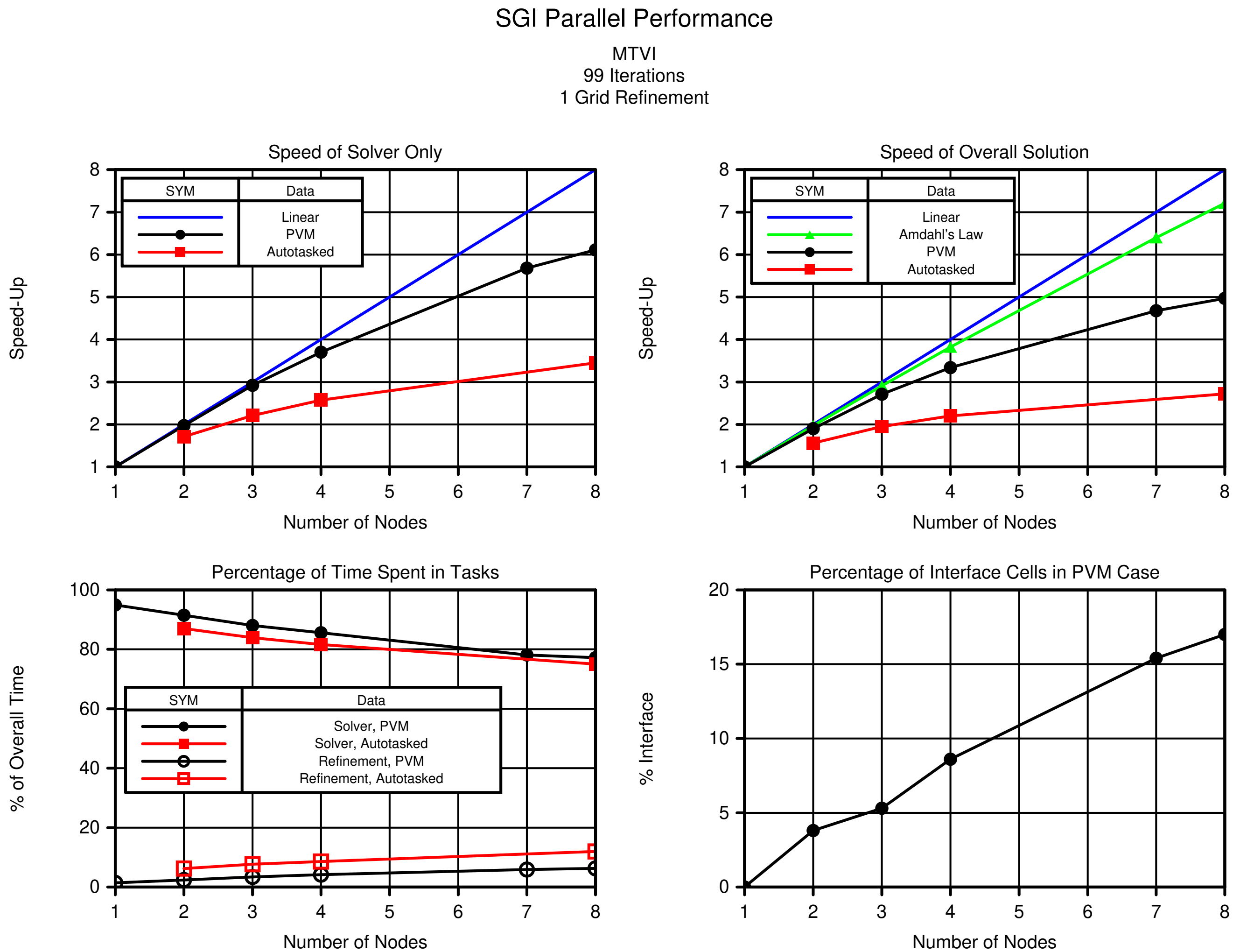


Figure 5.2: Timing Comparisons on the SGI

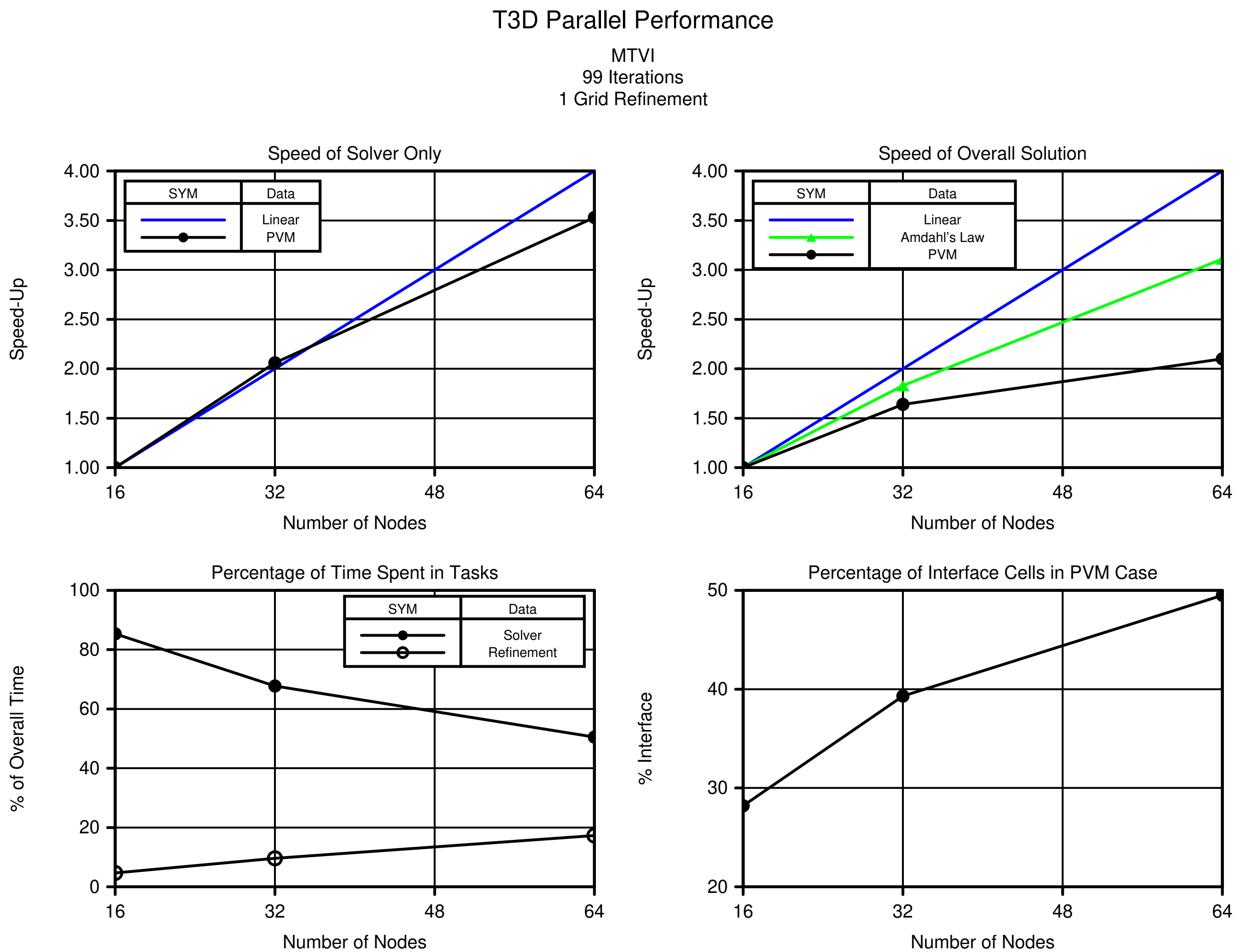


Figure 5.3: Timing Comparisons on the T3D

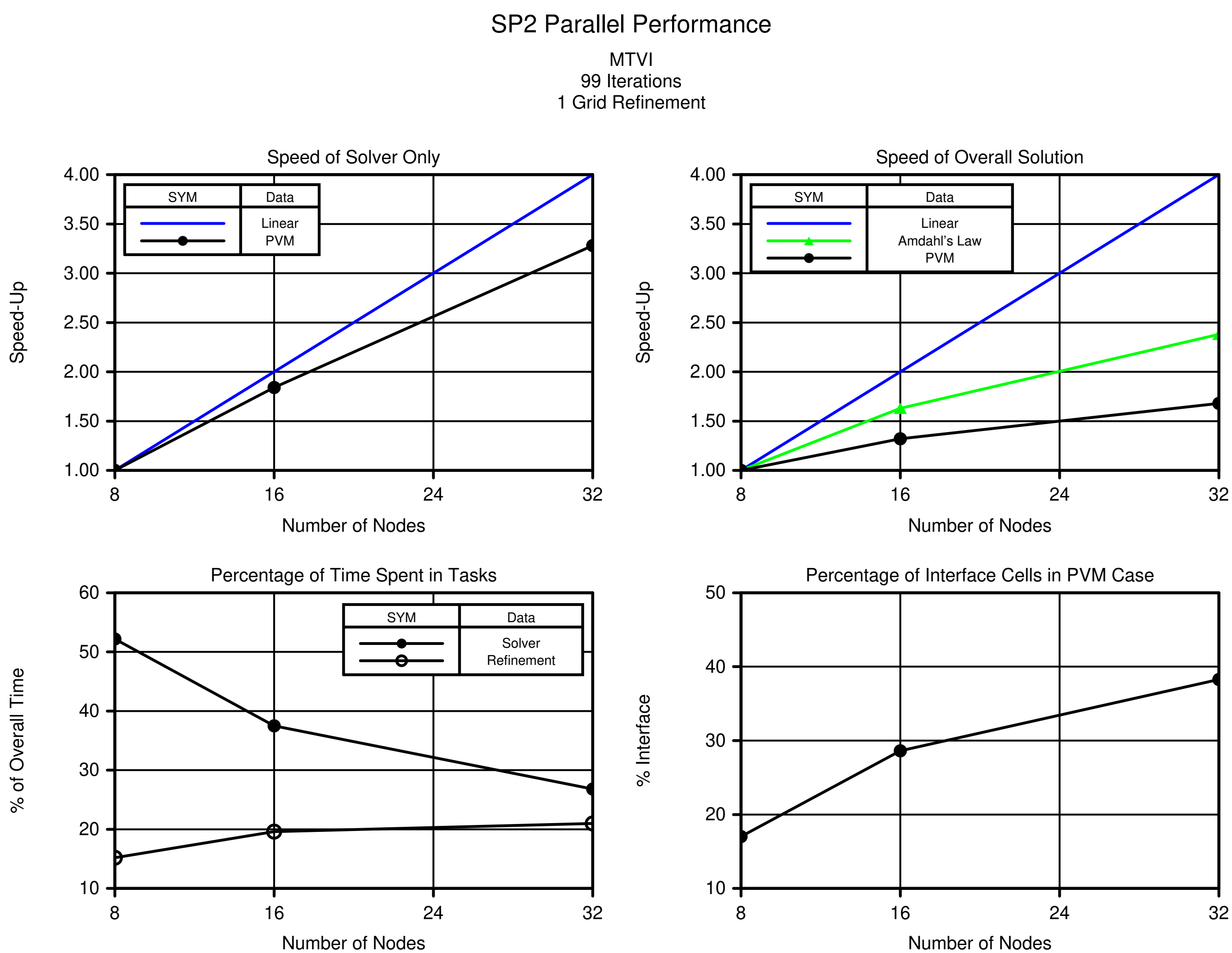


Figure 5.4: Timing Comparisons on the SP2

6.0 Validation of Parallel SPLITFLOW

Two cases are discussed for the validation of parallel SPLITFLOW:

1) Modular Transonic Vortex Interaction (MTVI)
2) Lockheed Wing C

The host code, which was dimensioned to hold 700,000 cells and 1.4 million boundary facets, requires 420 Mbytes of memory. Each node code, which is dimensioned to store 70,000 cells and 120,000 boundary facets, requires 120 Mbytes of memory.

MTVI

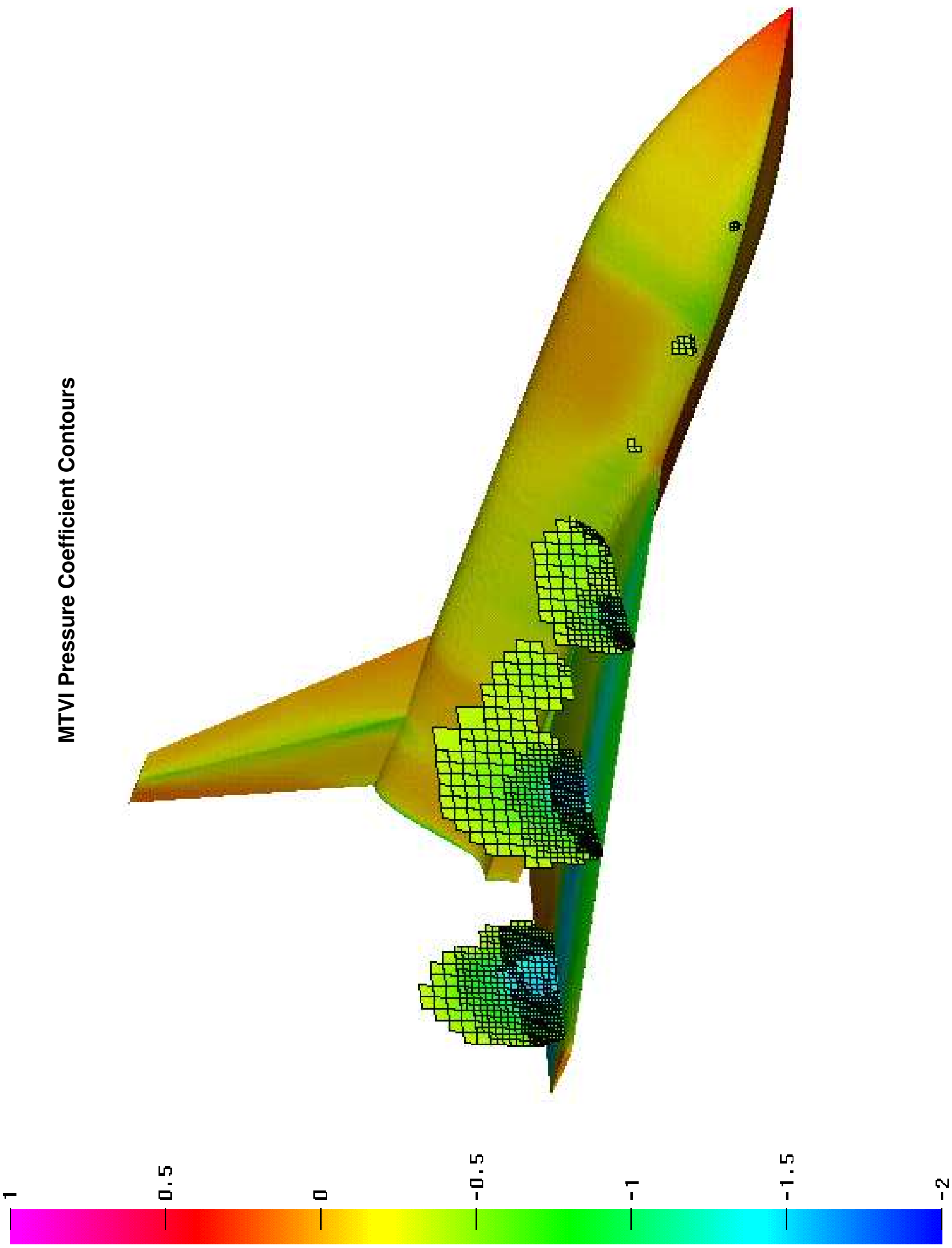
This geometry, which was provided by NASA Langley, has been the subject of Euler investigation for some time. The flight condition used for this validation case is a Mach number of 0.85 and an angle of attack of 10 degrees. Sideslip and leading edge flap deflection are both zero. This case was run on IBM SP2 on 17 processors (1 host and 16 nodes). The stop-check feature was deactivated so file writing on the host could occur simultaneously with solving on the nodes. The superbee flux limiter was used for suppression of numerical oscillations while allowing the solution to capture and resolve the suction in the forebody and wing vortex.

Figure 6.1 shows the general shape of the single tail MTVI. The pressure coefficient is shown on the surface and in the flow field at several fuselage stations of interest. The grid cell clustering toward the low pressure core of the inviscid vortex is visible. Figure 6.2 compares the surface pressure coefficient obtained from wind tunnel data with the SPLITFLOW solution. Figure 6.3 shows the location and cross sectional shape of the vehicle where these comparisons are made. These results show the same trends as previously documented SPLITFLOW results (Ref. 15). On the forebody, the suction peaks are somewhat underpredicted. On the wing, the suction peaks are overpredicted and more outboard than the experimental data indicates. This is due to the Euler code failing to predict a secondary vortex (a viscous phenomenon) which alters the shape of these peaks.

The convergence history is shown in figure 6.4. The solution ran for 700 iterations with the grid refining every 50 iterations. The superbee flux limiter prevented the residual from dropping more than two orders of magnitude. Therefore, the solution was converged until the surface pressures stopped changing at the fuselage stations of interest. The CPU times of the nodes are nearly identical such that the curves are on top of one another. The CPU time of the host is the shallowest curve. The low CPU time of the host does not indicate that the host was idle. The host was occupied with writing the global output files every 50 iterations, a task which requires a large amount of wall time. (Recall that this task may be performed by the host while the nodes are iterating.)

The number of computational cells increased from 112,937 in the initial grid to 382,089 in the final grid. The full octree grid including the parent cells ranged from 172,593 to 473,457.

Figure 6.1: MTVI Pressure Coefficient Contours



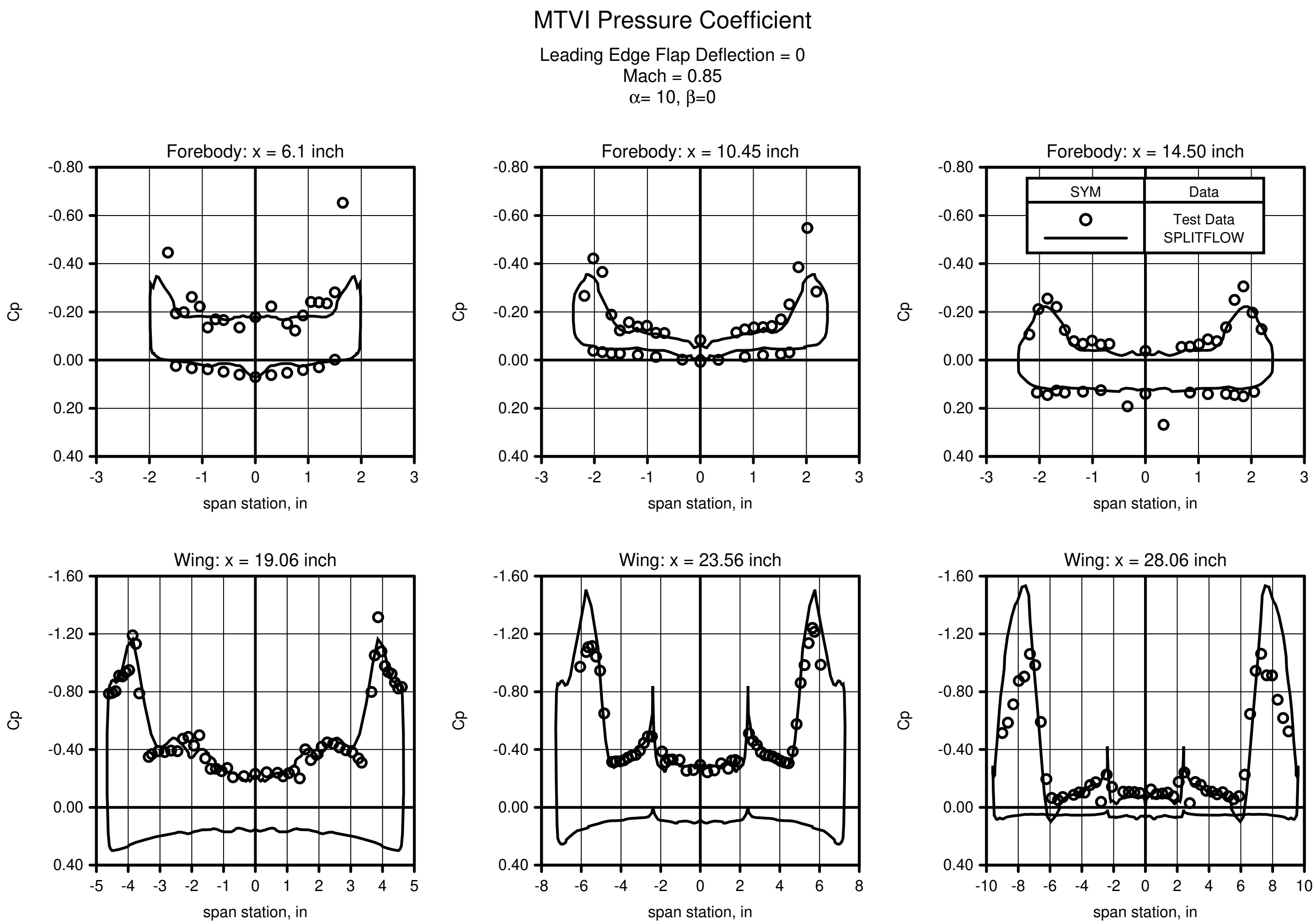


Figure 6.2: MTVI Solution

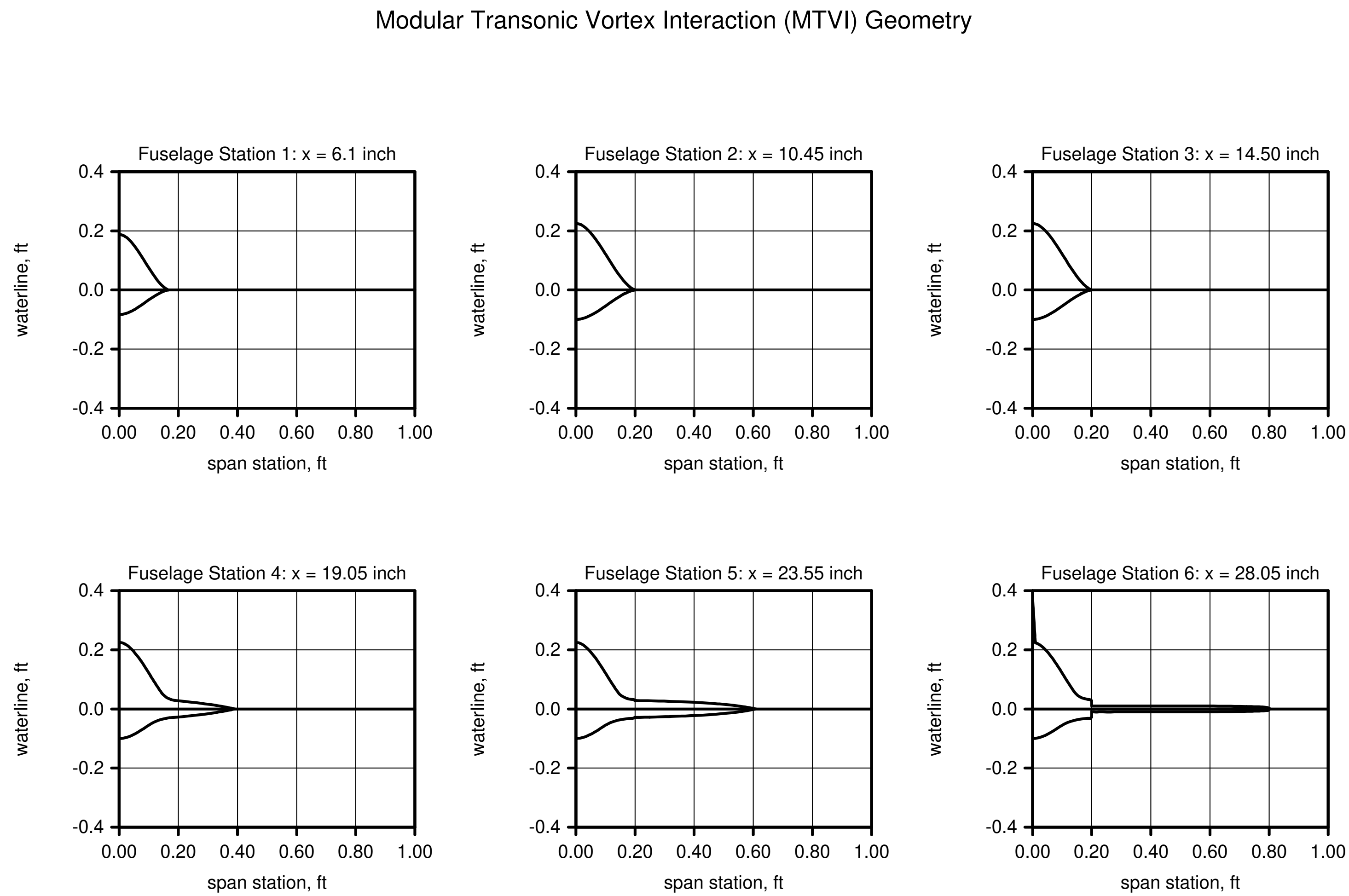


Figure 6.3: MTVI Geometry

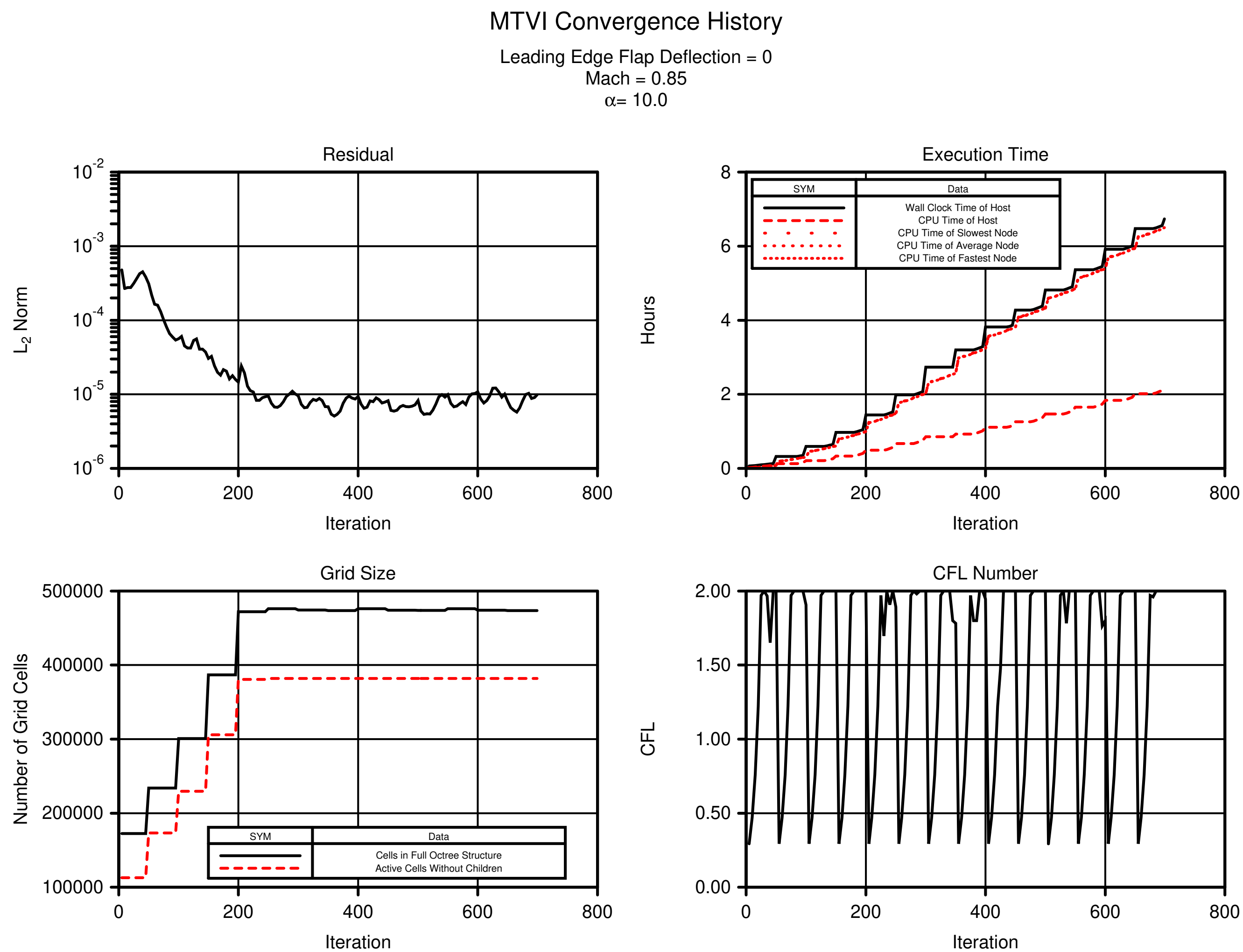


Figure 6.4: MTVI Convergence History

Lockheed Wing C

This configuration was chosen because it is difficult to grid with the octree gridding scheme. Although the geometry of a simple airfoil seems simple, the sharp trailing edge on the wing requires a large number of Cartesian grid cells to resolve. The trailing edge was slightly blunted in order to obtain a grid with a more manageable number of grid cells. Easier gridding may be accomplished with the prismatic version of SPLITFLOW currently under development. However, only the Cartesian version has been parallelized. The flight condition simulated in this validation case is a Mach number of 0.85 and an angle of attack of 5 degrees. This case was run on the IBM SP2 on 17 processors (1 host and 16 nodes). The stop-check feature was deactivated for faster overall timings. The minmod flux limiter was used for robustness, and smoothness in the solution.

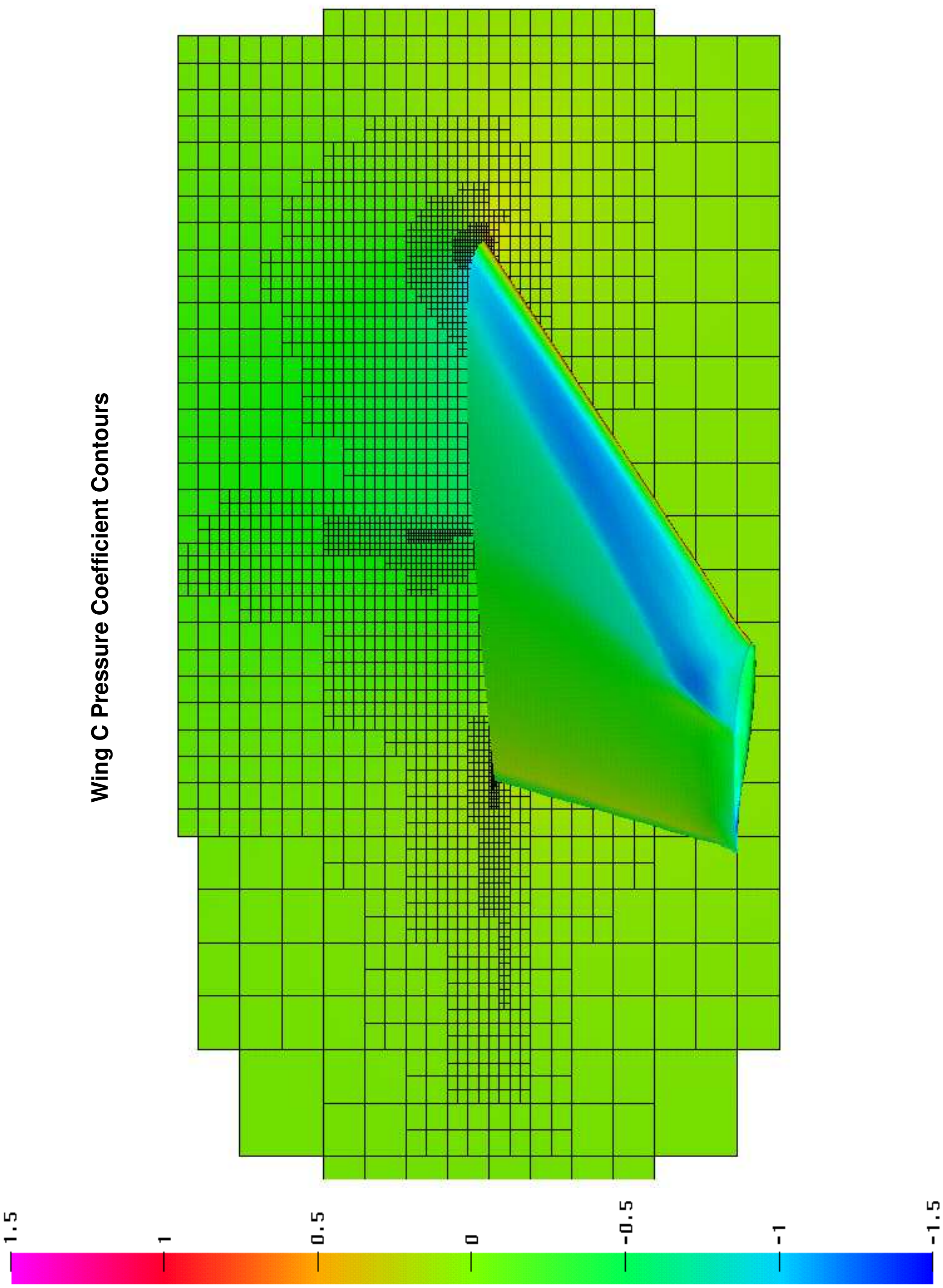
Figure 6.5 shows the general shape of the wing and the grid at the 30% span station. The pressure coefficient contours are illustrated on the surface, and on the grid at the span station mentioned. Grid clustering is visible near the shock, leading edge, and trailing edge of the wing. This is where flow gradients and/or geometry facet spacing require high grid cell density for proper resolution. Figure 6.6 compares the final SPLITFLOW solution with experimental data at several span stations. The largest error occurs where the flow shocks down near the tip. This error is common for Euler solutions over this geometry where a small secondary vortex is not detected by inviscid computations (Ref. 16). The spike in the SPLITFLOW solution at the trailing edge is due to the blunted geometry used in the solution.

Figure 6.7 shows the convergence history. The CPU times of the nodes are nearly identical to one another. The low CPU time of the host indicates that it is not performing computationally intensive tasks while the nodes are iterating. Rather, it is occupied with writing output files because the stop-check feature is deactivated.

The number of computational grid cells grew from 219,359 to 434,062 through adaptive grid refinements every 50 iterations. The number of cells in the full octree data structure grew from 252,505 to 504,337.

.

Figure 6.5: Wing C Pressure Coefficient Contours



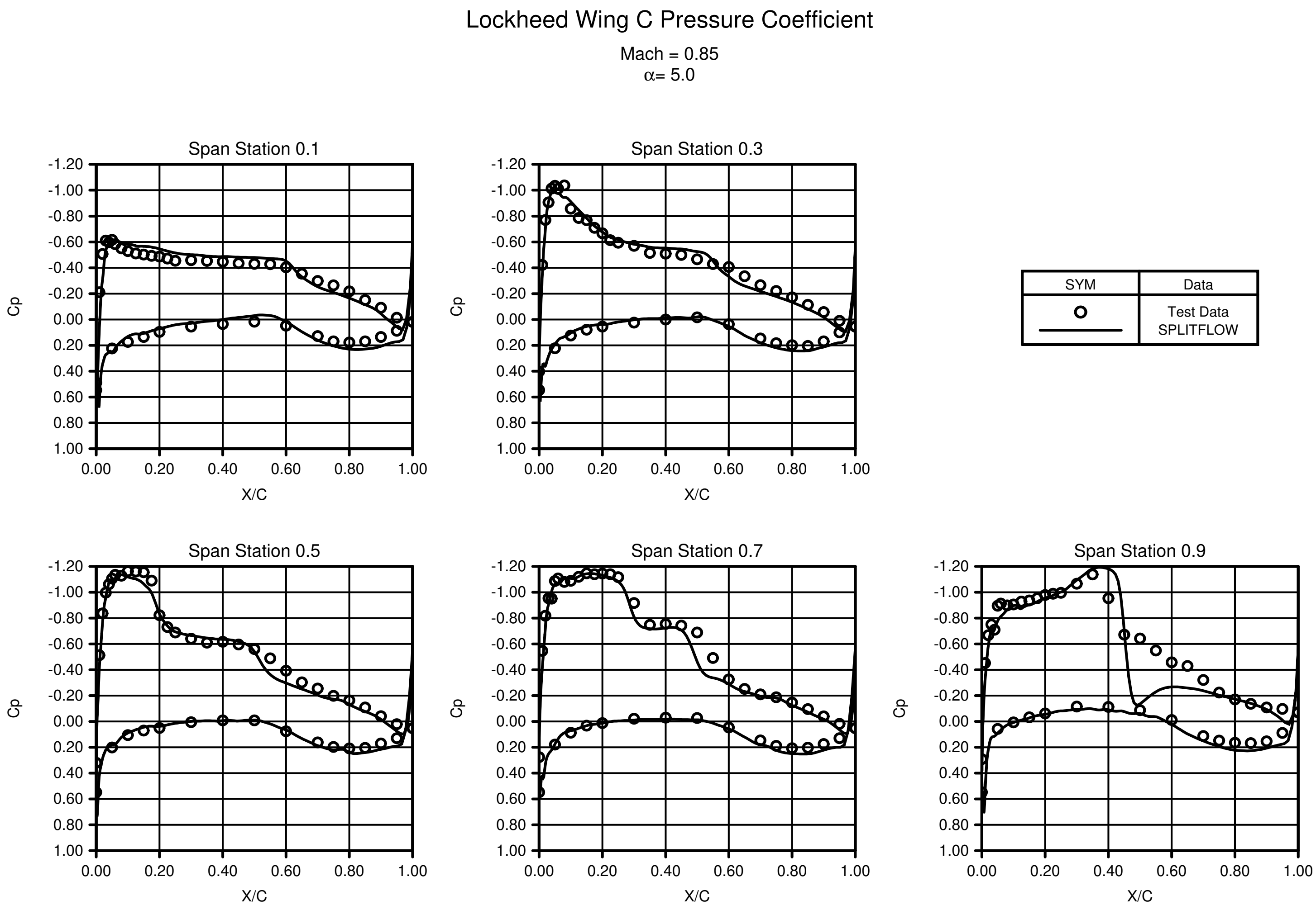


Figure 6.6: Wing C Solution

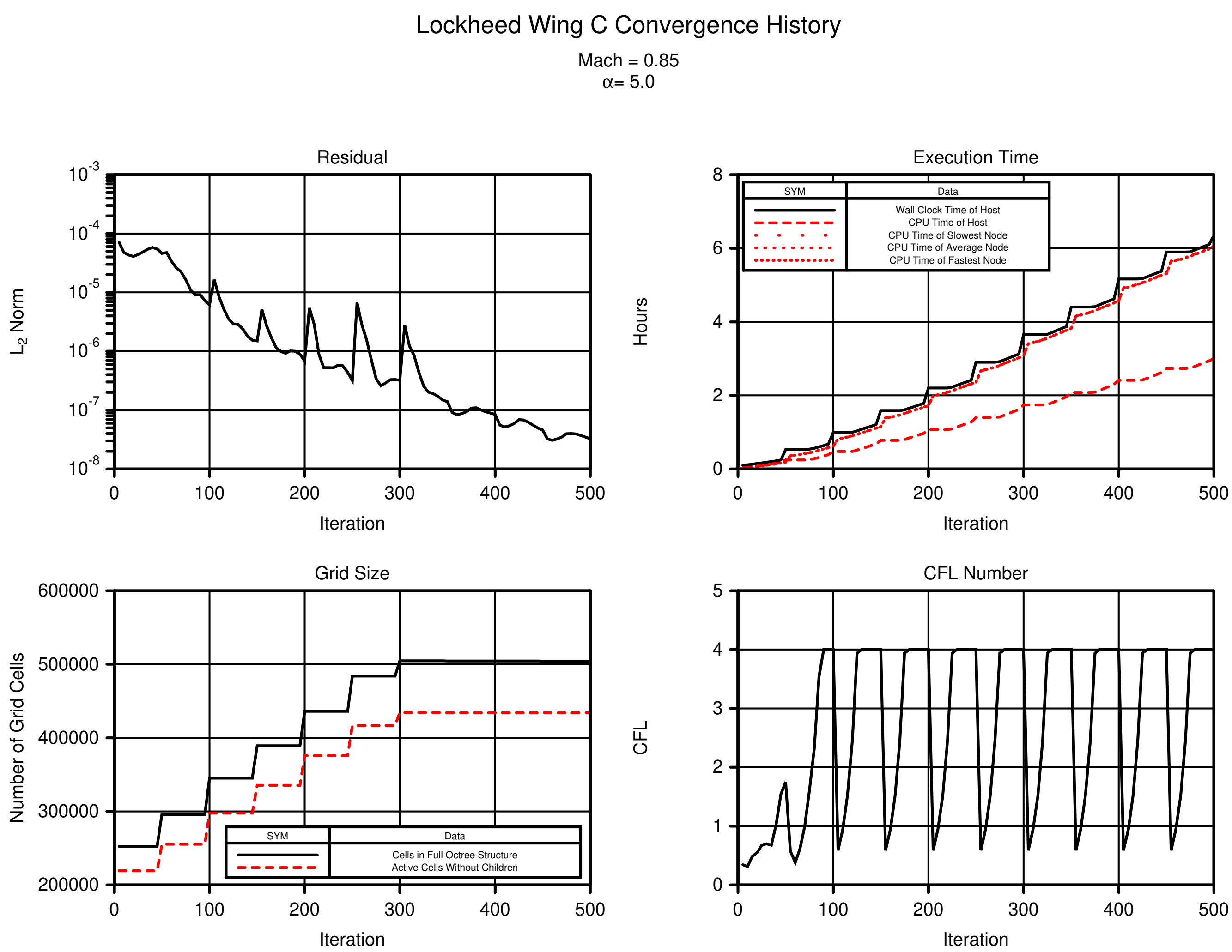


Figure 6.7: Wing C Convergence History

7.0 Conclusions

SPLITFLOW has been modified to run effectively in a distributed memory environment. The domain decomposition procedure quickly divides the global domain into pieces with the grid cells and boundary facets evenly distributed. Thus, load balancing is achieved on the subdomain processors, and is dynamically adjusted as the automatic grid refinement occurs. The solver and post-processor scale well with the number of processors. Timing comparisons between parallel SPLITFLOW and non-parallel SPLITFLOW (autotasked across several processors) on a shared memory machine show parallel SPLITFLOW to clearly be the faster version. Although parallelization on coarse grain machines shows the best scalability, results on massively parallel machines showed them to be viable alternatives to the coarse grain shared memory machines.

A host/node philosophy was adopted to modify SPLITFLOW for parallelization. The "host" code performs most of the input/output and grid generation tasks. The "node" code performs the solving and most of the postprocessing tasks. The host requires 40% of the memory of non-parallel SPLITFLOW. The node has a memory requirement proportional to the fraction of the domain contained by the node. Thus, the node processes are load balanced for computational effort and memory requirements. Timings of the solver show good parallelization over the processors. As the number of nodes increases, the wall clock time is consumed primarily by grid refinement (partially parallelized), and writing output files (unparallelized).

The domain decomposition procedure sorts the active cells without children according to the coordinate of the reference vertex of each cell. The cells which contain boundary facets are distributed among the nodes such that the number of boundary facets is nearly uniform. Interior cells are distributed such that the total number of cells (boundary + interior) on each node is uniform. The initial sorting is performed to reduce the number of interface cells, which are common to more than one subdomain. The decomposition procedure evenly distributes the domain into any number of subdomains.

Performance of parallel SPLITFLOW on the Cray C90 and SGI Power Challenge showed it to be significantly faster than its non-parallel autotasked counterpart. This result indicates that current SPLITFLOW users at LMTAS may expect a speed improvement by switching to parallel SPLITFLOW on the hardware currently in use. Results on massively parallel machines showed the SP2 to be faster, although it spent a large fraction of its time writing output files (i.e., accessing the disk). The T3D produced timing results which were slightly more scalable, but its processors are much slower.

Euler solutions generated with parallel SPLITFLOW match those of non-parallel SPLITFLOW. Results indicate that parallel SPLITFLOW is quite capable of solving flow over complex geometries. The difficulties encountered have been generally due to memory limitations and the use of Cartesian cells, rather than the parallelization of the code. Enhancements are underway to reduce these difficulties, and improve the parallelization of more tasks in the code.

8.0 Acknowledgments

This effort was sponsored by NASA Ames Research Center under Contract NAS2-14057. Alex Woo of NASA Ames Research Center is gratefully acknowledged for his technical guidance and support throughout this contract. Farhad Ghaffari of the Transonic/Supersonic Aerodynamics Branch of NASA-Langley Research Center generously provided technical data on the MTVI. Computer time on the IBM SP2 at the National Aerodynamic Simulation facility was supplied by NASA Ames Research Center. Dedicated time on the Cray C90 and T3D was provided by Cray Research.

9.0 References

[1] Underwood, M., Riggens, D., McMillan, B., Lu, E., Reeves, L., “The Computation of Supersonic Combustor Flows Using Multi-Computers,” AIAA-93-0060.

[2] Gropp, W.D., Smith, E.B., “Computational Fluid Dynamics On Parallel Processors,” AIAA-88-3793-CP

[3] Scherr, S.J., “Implementation of an Explicit Navier-Stokes Algorithm on a Distributed Memory Parallel Computer,” AIAA-93-0063.

[4] Karman, S.L., “SPLITFLOW: A 3D Unstructured Cartesian/Prismatic Grid CFD Code for Complex Geometries,” AIAA-95-0343.

[5] Henriksen, P., Keunings, R., “Parallel Computation of the Flow of Integral Viscoelastic Fluids on a Heterogeneous Network of Workstations,” International Journal for Numerical Methods in Fluids, vol. 18, pp. 1167-1183, 1994.

[6] DeZeeuw, D., Powell, K.G., “An Adaptively-Refined Cartesian Mesh Solver for the Euler Equations,” AIAA-91-1542-CP.

[7] Melton, J.E., Enomoto, F.Y., Berger, M.J., “3D Automatic Cartesian Grid Generation for Euler Flows,” AIAA-93-3386-CP.

[8] Harten, A., “High-Resolution Schemes for Hyperbolic Conservation Laws,”, Journal of Computational Physics,. vol. 49, pp. 357-393, 1983.

[9] Hassman, D., “The Cray J916 Supercomputer System,” http://www.cray.com/PUBLIC/product-info/J90/J916.html

[10] Hassman, D., “The Cray C98 Supercomputer System,” http://www.cray.com/PUBLIC/product-info/C90/C98.html

[11] Long, L.N., Khan, M.M.S., Sharp, H.T., “A Massively Parallel Three-Dimensional Euler/Navier-Stokes Methods,” AIAA-89-1937-CP.

[12] "The POWER CHALLENGE Technical Report," http://www.sgi.com/Products/hardware/Power/chap3.html

[13] Hassman, D., "The Cray T3D -- The Right Tool at the Right Time," http://www.cray.com/PUBLIC/product-info/mpp/T3D_overview.html

[14] Beaumont, C., "Parallel Processors," http://www.nas.nasa.gov/NAS/GenInfo/parallel.html

[15] Finley, D.B., "Euler Technology Assessment Program for Preliminary Aircraft Design Employing SPLITFLOW Code With Cartesian Unstructured Grid Method," NASA Contractor Report 4649, March 1995.

[16] Kinard, T.A., Schabowski, D.M., "An Assessment of Unstructured Grid Technology for Timely CFD Analysis," NASA CR-3291, Surface Modeling, Grid Generation, and Related Issues in Computational Fluid Dynamic (CFD) Solutions, May 1995, pp. 385-400.

<table>
<tr><td rowspan="2"></td><td>NAS TECHNICAL REPORT</td></tr>
<tr><td>Title:
Research in Parallel Algorithms and Software for Computational Aerosciences</td></tr>
<tr><td></td><td>Author(s):
Neal D. Domel</td></tr>
<tr><td>Two reviewers must sign.</td><td>Reviewers:
“I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document.”

Signed: ______________________________

Name: ______________________________

Signed: ______________________________

Name: ______________________________</td></tr>
<tr><td>After approval, assign NAS Report number.</td><td>Branch Chief:

Approved: ____________________________</td></tr>
<tr><td>Date:</td><td>NAS ReportNumber:</td></tr>
</table>