Parallel and Distributed Computational Fluid Dynamics: Experimental Results and Challenges

M.J. Djomehri, R. Biswas, R. Van der Wijngaart, and M. Yarrow

Computer Sciences Corporation

NASA Ames Research Center, Moffett Field, CA 94035, USA

{djomehri,rbiswas,wijngaar,yarrow}@nas.nasa.gov

Abstract

This paper describes several results of parallel and distributed computing using a large scale production flow solver program. A coarse grained parallelization based on clustering of discretization grids combined with partitioning of large grids for load balancing is presented. An assessment is given of its performance on distributed and distributed-shared memory platforms using large scale scientific problems. An experiment with this solver, adapted to a Wide Area Network execution environment is presented. We also give a comparative performance assessment of computation and communication times on both the tightly and loosely-coupled machines.

1 Introduction

Rapid recent improvements in high-performance computing hardware have made the simulation of complex flow models a viable analysis tool. However, further efficiency increases must be achieved in order to integrate this tool into the design process, which requires large

numbers of separate flow analyses. The only way in which drastic improvements in performance can be obtained in the short term is to utilize extensively parallel and distributed computing techniques. If the flow analyses can be carried out independently, we speak of *parameter studies*, which have significant inherent parallelism and require no interprocess data communication. These problems are well-suited for distributed solution, and are the subject of a separate study. If only one solution can be computed at a time, it must be produced very quickly to make it feasible within the design cycle. In recent years significant strides have been made towards this goal through parallelization of some flow solver programs of importance to NASA Ames. The objective of this paper is to assess the parallel performance of one such code, OVERFLOW, identify potential bottlenecks, and determine the impact of these bottlenecks on performance of the code in the more demanding environment of distributed computing using geographically separated resources.

Two realistic test cases are used for the performance evaluation of the solver. These cases consist of viscous calculations about complex geometric configurations. Case MEDIUM consists of 9 million grid points and case LARGE consists of 33 million grid points. The computations are carried out using the NASA code OVERFLOW, which deals with the geometrical complexity of flow solution domains by allowing sets of separately generated and updated *structured* discretization grids to exchange information through interpolation. We consider both single, tightly coupled architectures (SGI Origin2000 and Cray T3E), and geographically separated machines (multiple Origins) connected using the Globus metacomputing toolkit [8].

The remainder of this paper includes a brief overview of the numerical (Section 2) method and parallelization (Section 3) used. The parallelization approach is compared with that used in our earlier work [1], and a strategy for improvement of the load balance is described in Section 6.1. Results are presented in Section 5, and summary remarks and a discussion of future directions is given in Section 6.

2 Numerical method

The OVERFLOW code utilizes so-called multi-zone overset grid systems based on the Chimera [14] technology to solve the thin layer Navier-Stokes equations, augmented with a turbulence model. It is widely used in the aerodynamics community and is most popular flow solver in use at NASA Ames Research Center. It uses finite differences in space on structured grids and implicit time-stepping to capture near-body viscous effects. A variety of different time-stepping and spatial differencing schemes are available. For steady-state problems such as those studied in this paper, fast relaxation to the final solution is achieved by a combination of a sophisticated multigrid [5] convergence acceleration scheme and a local time-stepping approach, which allows the solution to be updated based on a spatially varying virtual time increment.

Much of the benefit of obtaining fast and highly accurate solutions with OVERFLOW is derived from the special properties of so-called structured discretization grids. Such grids individually are not well suited for geometrically complex domains, so they are used within the context of multiple overlapping grids, each of which covers only part of the domain. The resulting configuration is called an overset grid system. The solution proceeds by updating, at each iteration, the inter-grid boundaries on each grid with interpolated solution data from overlapping grids. See Figures 1a and b. The physical coordinates of the Chimera interpolation points are fixed in time and can be determined prior to the run.

3 Basics of parallel implementation

It is evident from the above description that the solution process for realistic flow problems features at least one level of exploitable parallelism, known as coarse-grained parallelism; the computation of the flow solution on individual grids can be carried out independently by different processors, as described in our previous work [1, 6]. There we presented results of running OVERFLOW in parallel and distributed environments. That version of the code

(See [13, 15]) did not allow individual grids to be distributed across several processors, and hence presented a load balancing problem in case of large disparities between grid sizes. The new version described here, due to Jespersen [6, 7, 3], solves part of this problem by providing the means of parallelizing the solution process *within* a single grid: it provides a second level of exploitable parallelism. While this allows us in principle better to balance the computational load among the processors, three important sources of overall load imbalance remain.

First, all grid points are currently given equal weight in terms of associated work. However, some of the near-body grids require more work per point, because they need to solve the turbulence model in addition to the flow equations. For better load balance we need to give different weights to points inside turbulence regions than to points outside of them.

Second, processors may either solve part of one large grid, or one or more whole grids, but not both. This means that once a processor receives work for part of a larger grid, it cannot further reduce any load imbalance by receiving more grids or grid fragments.

Third, even if the computational work is divided completely evenly among the processors, load imbalances may still result from disparities in communication volumes. The reason for this is twofold. First, if a large grid is distributed across multiple processors, the implicit solution process within this grid must be parallelized, which ordinarily requires a significant amount of communication (indicated by the heavy arrows in Figure 1d). This communication is in addition to any communications required to interpolate data from different neighbor grids. Second, the grid grouping and splitting strategy currently does not explicitly take into account the magnitude of the data volume incurred by the decomposition. In [1] we described a way of minimizing the maximum communication volume between processors. This strategy will have to be further refined to reflect the possibility of individual grids being distributed. We discuss a promising new load balancing strategy in Section 6.1.

Some basic features of the parallel strategy implemented in the OVERFLOW code, used for the experiments reported in this paper, are discussed in subsequent subsections.

3.1 Grouping strategy and grid splitting

The first step of the parallel algorithm involves a simple bin-packing strategy that forms a number of groups, each consisting of a grid and/or a cluster of grids. If no further work division takes place, the total number of grid points per group is limited by the memory space allotted to each processor. As we reported in [1], this strategy may produce a poor load balance, depending on the total number of grids and the size of the large grids in the grid system, and the number of processors available. To fix this, the current grouping strategy in OVERFLOW allows us to divide large grids evenly across multiple processors while maintaining the implicitness of the numerical scheme within the grid; however, because a processor receiving part of a grid needs to exchange information with the other parts *several* times per time step during the implicit solution process, it effectively needs to execute in lockstep with the other processors working on the grid. This has led to the requirement that a processor receiving a partial grid not receive any other work, to ensure that it will not cause the other processors working on the grid to go idle. As a consequence, each part of an equi-partitioned large grid sits in a group by itself. See Figure 1c. The total number of processors equals the total number of groups. All processes run in parallel; this is viewed as inter-group parallelism.

The grouping algorithm is implemented as follows. First, the programmer specifies the maximum number ngp_{max} of grid points that each processor may receive. This number is usually, though not necessarily, related to the amount of local memory available on each processor. Since the total number of grid points ngp_{tot} for a certain overset grid configuration is given, we can now determine the minimum number of processors NP_{min} required to fit the configuration, i.e. $NP_{min} = \lceil ngp_{tot}/ngp_{max} \rceil$. Next, we sort the grids by size in descending order, and break up any grids whose size ngp exceeds the maximum into $\lceil ngp/ngp_{max} \rceil$ pieces. Each such piece is assigned its own group, and hence its own processor, as indicated above. The remaining whole grids whose sizes are below the maximum are assigned to processors in the fashion described in our earlier work [1]. That is, the largest remaining

grid is assigned to the next available processor, until all NP_{min} processors own at least one grid. Each subsequent remaining grid is assigned to the processor that is responsible for the smallest total number of grid points thus far. When no more grids can be placed without exceeding the maximum number of points per processors, the remaining number of points is determined, and the minimum corresponding number of additional processors is computed. We repeat the process of assigning whole grids to processors based on the new processor count, until all grids can be placed.

OVERFLOW uses explicit message passing, based on the standard MPI library, for communicating inter-grid boundary data between processors. This approach is suited for both distributed and distributed-shared memory architectures. With grid splitting, message passing is also required between processors working on the same grid. See Figure 1d. Also, in this case, data communication is achieved by means of manager-worker (master-slave) paradigm between groups of a partitioned zone. One of the processors working within a subpartition of a zone is selected as the managerof all the other processors within the subpartition. Upon completion of one time-step on a partition zone, all the inter-group Chimera exchange between the partitioned zone and other groups is achieved via the master processors.

The steps in the implicit flow solution in which these communications take place are during the evaluation of the so-called right hand side (a data parallel stencil operation), which constitutes a nonlinear forcing term, and during the so-called line solves in the Alternating Direction Implicit algorithm. The line solves, which take place in all three coordinate directions, feature a data dependence in the active coordinate direction, which is resolved using pipelining. Finally, some types of non-local boundary conditions may also require communications (for example, so-called C-grid conditions), but these usually involve a negligible amount of data. A detailed analysis of the right hand side calculation and the line solves shows that an intra-grid interface point involves the communication of 76 eight-byte words per time step, whereas a Chimera interpolation point "consumes" only 5 words per iteration (without turbulence model). This disparity in communication size per interface point

is indicated in Figure 1d by the relative thickness of the arrows that symbolize inter-group communications.

Intra-grid communications take place through updates of "halos" of overlap points [7]. While this is a well-understood process, it adds significantly to the complexity of the implicit solution process, and requires many changes to the serial version of the code.

3.2 Parallel and serial Chimera update

As discussed above, boundary information is interpolated between overlapping grids at each iteration. In serial version of the code, upon completion of of the $n-th$ time-step iteration on one grid, its chimera updates become available to other grids and can immediately be used by the subsequent grids in their $n-th$ time-step iteration. In parallel version, chimera updates take place at the the end of completion of $n-th$ step over all groups; hence, interpolated data on all grids lag by one iteration.

Grids that overlap with other grids in the same and in a different group perform intra- and inter-group interpolations, respectively, between processors. The donor values supplied to the neighboring group are computed locally and then exchanged using MPI calls. This approach is outlined schematically in Fig. 2, where two groups are shown, each containing two grids. Both intra-group and inter-group interpolations take place at the end of each iteration; hence, interpolated data on all grids lag by one iteration.

The major computational loops at the top level of the code include the Time-step loop and the Grid loop. Computations at the latter can proceed either sequentially or in parallel for the serial and parallel codes, respectively, as discussed above. In the serial code, the update is achieved under the Grid loop; whereas, in parallel codes the updates take place under the Time-step loop, outside the Grid loop. This may be thought of as a block-Gauss-Seidel iteration versus a block-Jacobi iteration. The stability region for the former is larger than for the latter, which translates into a faster convergence to a final, steady-state solution. The significant effects of the parallel versus serial updates in practical applications, depend

on the topology of the grid configuration, connectivity and flowfield features, time-step, CFL number, and other input parameters used. For the same application, the Jacobi iteration may require a smaller time step and/or CFL number for convergence of solutions. In such cases the convergence may be influenced by adjusting some algorithmic parameters locally on each grid. A diagram of major OVERFLOW loops for parallel and serial codes depicting the Chimera boundary updates are shown in Figure 3.

4 Parallel distributed computing

One of the early implementation of distributed computing has been the Parallel Virtual Machine (PVM) [10] library. This approach has already been used in isome older version [2, 3] of OVERFLOW, with some limitations. For example all remote computer resources had to be named within the application program, and necessary input data had to be moved to proper location by the user, and number of issues on security and accounting had to be resolved.

The distributed computing methodology used in this work is based on the NASA Information Power Grid (IPG) project [12]. It is one of several infrastructural approaches to so-called grid computing [9]. IPG provides an environment for resource management with the ability to unify multiple physically separated computational resources into a single virtual machine. CFD solutions to geometrically complex, large-scale problems, whose computational memory requirements exceed that of the memory space of available individual resources, can potentially be obtained by employing widely distributed computing; A large data set can be decomposed into an assembly of smaller, more manageable data sets, and then be distributed across a collection of specified resources. This mode of application is usually called "parallel distributed computing". Another area of application that can significantly exploit this capability, would be the parametric study of a large scale problems where real-time CFD solution data of various flow parameters is essential. This mode of

application may be thought of as "embarrassingly-parallel distributed computing". In this paper we describe some experiments of both the plain parallel and the parallel distributed computing modes, applied to the OVERFLOW code.

The "parallel-distributed" methodology is similar to that of the parallel code. It uses the same group strategy and pertinent processor assignment discussed above, but processes are now distributed across distinctly specified resources. The schematic of distributed computing is shown in Fig. 4. In this figure, grid zone 1 and 2 are clustered into group 1 and zone 3 into group 2 by itself. The two processors assigned to both groups 1 and 2 are selected from resource 1. Similarly, zones 4 and 5 are clustered into group 3 and each equi-subpartition of the large zone 6, is assigned to groups 4 through 7. All the processors assigned to each of the groups 3 to 7 are selected from resource 2. Resources 1 and 2 are physically separated. Solid and broken two sided arrows show data communication within each resource and across the resources.

It should be noted that grid splitting is now only allowed within a single resource, in order to avoid too voluminous communications between geographically separated machines. The total number of groups is equal to the total number of CPUs available from all resources. Inter- and intra-grid boundary data has to be transferred between processors as before, again using explicit message passing. This is implemented via the MPICH-G [11] communication library, in conjunction with the Globus Metacomputing Toolkit [8]. Functionally, the entire application is run as a single message-passing program under MPICH, and the application programmer need not be aware of any distinction between the multiple machines.

5 Results

Several experiments are presented to demonstrate the parallel and distributed computing performance of the OVERFLOW CFD code discussed above. Parallel performance results on single, tightly-coupled machines, SGI Origin2000 (O2K) and Cray T3E, are discussed in

Section 5.1, using two large-scale, practical application problems, consisting of static grid systems with a total of 9 (MEDIUM) and 33 (LARGE) million grid points, respectively.

Distributed parallel computing performance analysis on NASA IPG testbed systems, O2K, at Ames, Langley and Glenn Research Centers, using the MEDIUM grid case above, is discussed in Section 5.2.

5.1 Parallel Performance

The current version of OVERFLOW has been run on an O2K, *R12000, 300MHz*, with a total of 128 processors, and on a Cray T3E, *300MHz*, with a total of 512 processors. Test cases are as follows:

- MEDIUM consists of a wing-body configuration mounted on the splitter plate of the NASA Ames 12-Foot Pressure Wind Tunnel (PWT), where internal flow at the test section of the tunnel and about the model has been simulated. The flow domain is discretized with 32 overset grids, and a total of about 9 million grid points.

- LARGE consists of a complex configuration of a high wing transport with nacelles and deployed flaps, discretized with 153 overset grids, and a total of about 33 million grid points.

Figure 5, shows an isometric view of some overset grids for a generic wing-body test object mounted in the 12-Foot PWT for visualization of grids, and Fig. 6 displays the corresponding solution (pressure). The simulation of the tunnel's internal flow about the model takes into account the effects of the tunnel wall and other support equipment interferences. For all test cases discussed here, the main code specifications are the one-equation Spalart-Allmaras turbulence model, the Roe upwind scheme, and the ARC3D 3-factor diagonal scheme, along with the usual second and fourth order smoothing options. Performance statistics of various scaling data are shown in Table 1. The table lists data for the MEDIUM test case and consists of wall-clock time (in seconds/time step), million of floating point operations per

second (Mflops). and the ratio of average to maximum inter-group (Chimera boundary) communication times. Data transfer times for the intra-grid communications are not listed separately. The Mflops reported on the O2K and the T3E are calculated relative to the Cray C90 Mflops with one CPU executed on the same problem.

The T3E is a purely distributed-memory machine, and the size of MPI processes run on a node is limited by the physical memory located on that node. Thus the smallest number of nodes on which the MEDIUM test case can be run is 88, while the LARGE case requires a minimum of 203 nodes. By contrast, the O2K allows MPI processes to use as much memory as is physically available on the whole machine, so no minimum number of nodes is required to solve both test cases. However, we do try to maintain a reasonable balance between number of nodes requested and maximum amount of memory used, so that the interference with jobs run by other users is minimized.

We conclude from Table 1 that the MEDIUM grid configuration allows reasonable scaling up to 124 and 271 processors on the O2K and T3E, respectively. The average speedup on O2K, as compared to the machine linear speedup, based on 4 CPUs, is about 40% for a number of processors in the range of 60 to 124. The code achieves a maximum of 7600 Mflops on O2K with 124 CPUs, as compared with approximately 4000 Mflops on the Cray C90 with 16 processors using the serial code with multitask directives. Similarly, the LARGE grid case results, shown in Table 2, indicate that this configuration scales well up to 96 and 510 processors on the O2K and T3E, respectively. Here the speedup characteristic on the T3E, is calculated based on 203 processors and appears to be linear, but only over a range where the ratio of maximum to minimum number of processors is about 2.5. The O2K experiences the same linear characteristic, based on 16 processors with the ratio of maximum to minimum about 8 for up to 96 processors and then tapers off for a larger number of processors.

It is interesting to note that scalability on the O2K tapers off sooner for the LARGE grid system than for the MEDIUM size configuration. This counterintuitive result is due to

the different distribution of grid sizes and inter-grid communications, and the concomitant poorer load balance. On the Cray T3E the performance for the MEDIUM case deteriorates beyond 271 CPUs due to communication overhead as compared to computation. For the MEDIUM case, average number of grid points per processors is less than 23,000 points for a total of 400 CPUs on the T3E, an order of magnitude less than the capacity per processors. Where as the average grid points per processors for the LARGE grid case on the T3E, is about one fourth of the capacity.

5.2 Parallel distributed computing performance

The NASA IPG testbed currently consists mostly of O2K, *R10000, 250MHz*, systems, and this is the platform we used for the parallel distributed computing experiments. The choice of an all-O2K distributed system was motivated by the desire to eliminate heterogeneity as a possible additional source or load imbalance. The machines used are Evelyn, Whitcomb, and Sharp, located at NASA research centers in California (Ames), Virginia (Langley), and Ohio (Glenn), respectively.

Since intra-grid partitioning is much more communication intensive than inter-grid Chimera interpolation, partitioned grids are never split across multiple geographically separated machines. Moreover, the boundary condition deferment method for latency hiding, described in our previous work [1], is applicable only to Chimera updates, not to the pipelining of the implicit solver in OVERFLOW.

The results of our experiments are summarized in Table 3 for the MEDIUM size configuration only, because the testbed systems at the moment have relatively modest amounts of memory. Several runs were made on various numbers of processors selected from each machine. Listed in the table is a sample run with a total of 2, 4, 8, 16, and 24 processors on multiple resources. Also, listed in the same table for comparison are results for the same number of processors, but on a single resource. There are two numbers listed for communication time, Min./Max., in the last column of Table 3. For clarity of illustration, we assume

a total of 8 processors were selected. The communication time for each of the 8 processors would be deferred depending upon the group to which they were assigned. The minimum and maximum for communication times over the 8 processors are shown in the table. The column entitled "walltime" in Table 3 refers to execution time per time-step in seconds which consists of computation and communication times and is about the same over all processors.

It is evident from this table that the scalability of the code on up to 24 processors is reasonable: although, it is clear that communication overhead becomes increasingly significant if more than 8 processors are used. The comparison of communication times, multiple resources versus that of the single resource, shows that both Min. and Max. communication times are significantly increased. The Min. communication time is about an order of magnitude larger on multiple distributed resources. Although the Max. communication time, for a total of 4, 16 and 24 processors, are very close to each other for the runs on multiple and single resources: it should not be understood that the timing reported, is necessarily associated with the same group. For instance, if the communication time for a group named, A, on a single resource is larger than for group B, it may produce different results when the same if the same job runs on distributed resources. The outcome totally depends on the volume of data exchanged for each group and how the groups are distributed across the resources. The increase in the magnitude of Min. communication time for the distributed case clearly depicts the impact of a lower rate of data transfer across the separated resources. This points to the necessity of load balancing and latency hiding, as argued in the Section 6, below.

The comparison of execution time on distributed versus single resources clearly reflects the integrated effect of computation and communication on execution time. Walltimes for the distributed runs are consistently larger than similar runs on a single resource, but they fall in a range well accepted for many applications. An analysis of walltime characteristics for distributed versus single resources, such as plots of walltime versus the number of processors, demonstrates similar scalable performances. Future plans will explore further algorithmic

enhancement and experiments on various homogeneous and heterogeneous testbeds for large scale applications.

6 Challenges

The performance results discussed above demonstrate the feasibility of parallel and distributed computing on homogeneous IPG testbeds; they further show that performance is significantly affected by an increase in communication over computation time. In a true IPG environment, poorer connectivity and larger latencies due to geographical separation of the computers used, could further impact performance. Modifications must be made to minimize communication overhead and to hide latency by overlapping communication with computation. These two phases of modification, known as "load balancing" and "latency hiding", will be necessary for large-scale computations that involve a large number of processors. Currently, none of the aforementioned techniques have been implemented on OVERFLOW.

6.1 Improved load balancing strategy

It is evident from Tables 1, 2, and 3 that there is a significant imbalance in the amounts of time spent on communications between the processors. The computational load imbalance is not listed explicitly, but it is on the order of 20% or more for most cases. Consequently, significant room for improvement is present. We propose the following strategy for better balancing the load, which requires some new definitions.

Introduce the concept of *effective grid points*. These are weighted such that each effective point takes the same amount of computational work. The weight will be deduced from the presence or absence of a turbulence model for the grid under consideration.

Introduce the concept of *effective communication volume*, to be associated with exterior grid boundaries (related to Chimera interpolation updates), and with internal grid boundaries (related to partitionings of individual grids and the ensuing communications required

by the implicit solution process). This volume equals the number of interface points, times the relative weight of the points (5 for Chimera points, 76 for internal boundary points).

The first step in improving the current algorithm is to allow processors to work on job mixes that contain both partial and whole grids. Since subdividing single grids incurs a significant communication cost, we use the heuristic that it is best to limit that partitioning process to the smallest number of subdivisions possible. Hence, we use the method outlined in Section 3.1 for distributing those grids that exceed the maximum number of points allowed per processor, regardless of the number of *effective* points involved.

Subsequently, we construct a graph whose nodes consist of the numbers of effective grid points per grid or—for distributed grids—per grid subdivision, and whose edges consist of the effective communication volumes between grids and/or subdivision. This graph can then be partitioned using any of a number of efficient graph partitioners (for example, MeTiS [4], as proposed in [1]) that are capable of balancing total node weight per partition while minimizing total weight of the cut edges. The only constraint is that no partition receive more than one grid subdivision. However, the way that subdivisions are created guarantees that no more than one of them will fit on any processor anyway, so the constraint is automatically satisfied. The reason why processors should not receive more than one partial grid is that all processors that cooperate on a particular grid need to synchronize. If processors participate in multiple such synchronized operations, it quickly becomes impossible to balance the load. But if they only participate in one during each time step, this can be scheduled as the first computational task to be performed, and no synchronization penalty is incurred.

When the partitioning is complete, several processors will generally be oversubscribed. That is, they will have received too many grid points. The number of points in the excess grids is then totaled, and a new number of processors is extrapolated from it, after which the assignment process is repeated until no processor exceeds the maximum allowable number of points.

Implementation of this strategy for the current version of OVERFLOW requires only

little coding, and will be carried out for the final submitted version of this paper.

Extension of this method to a widely-distributed computing environment is relatively straightforward: the grouping strategy takes place in two stages. The first only assigns collections of whole grids to individual, geographically separated platforms, whose computational resources are listed as aggregate quantities. Again, we use a graph partitioner to balance computational loads and to minimize communication volumes. In the second stage a careful load balance is obtained within each platform, using the method above.

6.2 Latency tolerant

The second attempt for improving performance of distributed computing is based on techniques that can hide communication while computation is in progress. One such approach was implemented and tested in our previous work [1]. In this approach, named, Deferred Strategy, the time-advancement procedures of the solution scheme were altered. With the original time-stepping procedures, upon completion of computation over all groups, Chimera boundary data is updated and exchanged prior to the start of the next time step. In the deferred scheme, the next time-step computations can start with the previous updates and also in the absence of recent Chimera updates, while this data is being communicated across processors.

In the deferred scheme Chimera updates lag one time-step behind as compared with the original time-stepping scheme, introducing further explicitness into the iteration procedure that might possibly deteriorate the stability of solutions, see Sec. 3.2. Experiments conducted with this approach show no degradation in convergence of the the solutions for a rather large scale steady-state application. However neither do they show any significant improvement in performance on two separated O2K machines for a combined total of up to 8 processors. A successful approach would require a co-processor to be allocated for data exchange alone, freeing other processors for computation. This idea, known as MPI-hide, was initiated as a research project at the time of this work, at Argonne National Laborotary, but as yet has

not been implemented in the OVERFLOW code, but it is a subject of future research in this area.

References

[1] S. Barnard. R. Biswas. S. Saini. R. Van der Wijngaart. M. Yarrow, L. Zechtzer, I. Foster, O. Larsson. Large-scale distributed computational fluid dynamics on the Information Power Grid using Globus. Frontiers of Massively Parallel Computation, February 21-25, 1999

[2] P. Buning, W. Chan, K. Renze, D. Sondak, I.-T. Chiu, J. Slotnick, R. Gomez, and D. Jespersen. Overflow user's manual, version 1.6au. NASA Ames Research Center, 1995.

[3] D.C. Jespersen. Parallelizing overflow: Experiences, lessons, results. In *NASA HPCCP/CAS Workshop*, August 25-27, 1998.

[4] G. Karypis and V. Kumar. A fast and high quality multi-level scheme for partitioning irregular graphs. Department of Computer Science Tech. Rep. 95-035, University of Minnesota, 1995

[5] T. Pulliam, D.C. Jespersen, and P.G.Buning. Recent enhancements to overflow. AIAA-97-0644, 1997.

[6] M.J. Djomehri and Y. Rizk. Performance and application of parallel overflow codes on distributed and shared memory platforms. In *NASA HPCCP/CAS Workshop*, August 25-27, 1998.

[7] G.P. Guruswamy. Y.M. Rizk, C. Byun, K.Gee, F.F. Hatay, and D.C. Jesperse. A multilevel parallelization concept for high-fidelity multi-block solvers. In *SC97: High Performance Networking and Computing, San Jose, CA*, November,1997.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[10] A. Geista, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22:789–828, 1996.

[12] W. Johnston, D. Gannon, and W. Nitzberg. Information Power Grid implementation plan. *Working Draft*, NASA Ames Research Center, August 1999

[13] R. Meakin. On adaptive refinement and overset structured grids. In *13th AIAA Computational Fluid Dynamics Conf.*, AIAA-97-1858, 1997.

[14] J. Steger, F. Dougherty, and J. Benek. A Chimera grid scheme. *ASME FED*, 5, 1983.

[15] A. Wissink and R. Meakin. Computational fluid dynamics with adaptive overset grids on parallel and distributed computer platforms. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1628–1634, 1998.

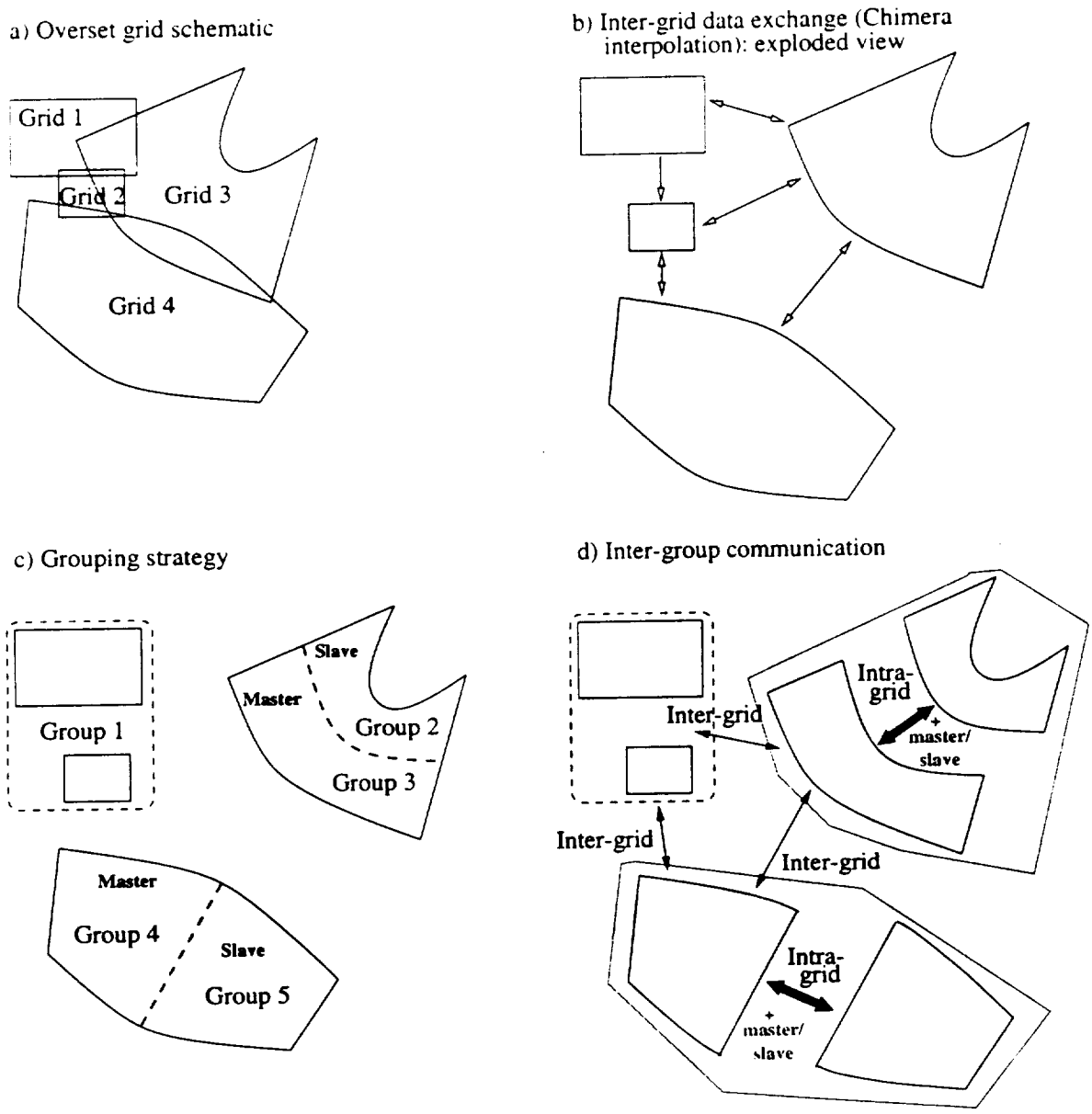

Figure 1: Basic partitioning strategy.

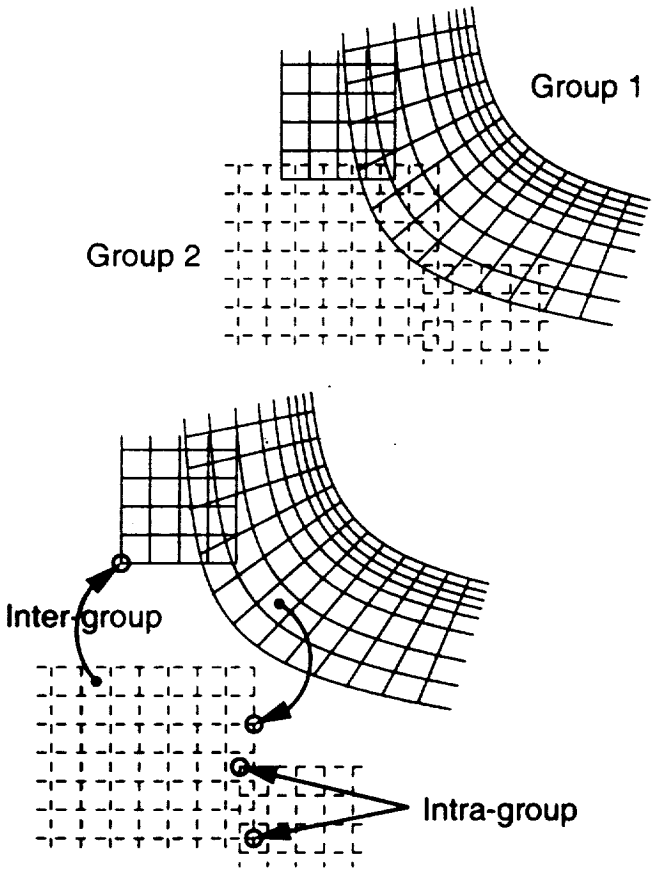

Figure 2: Inter-group interpolations between grids.

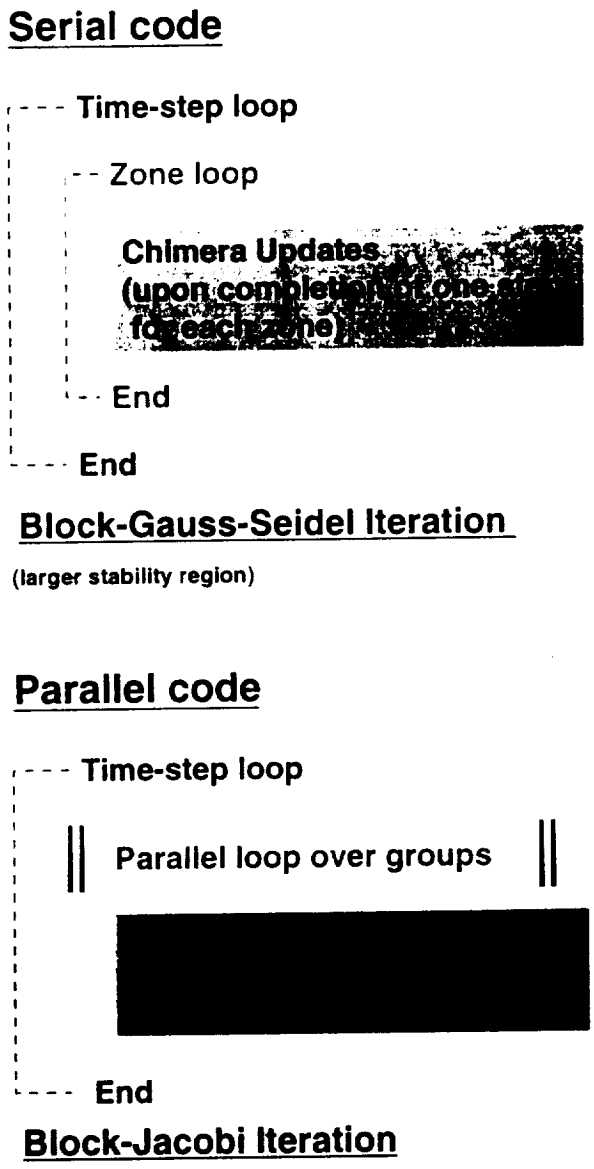

Figure 3: Logic flow for serial and parallel versions of OVERFLOW

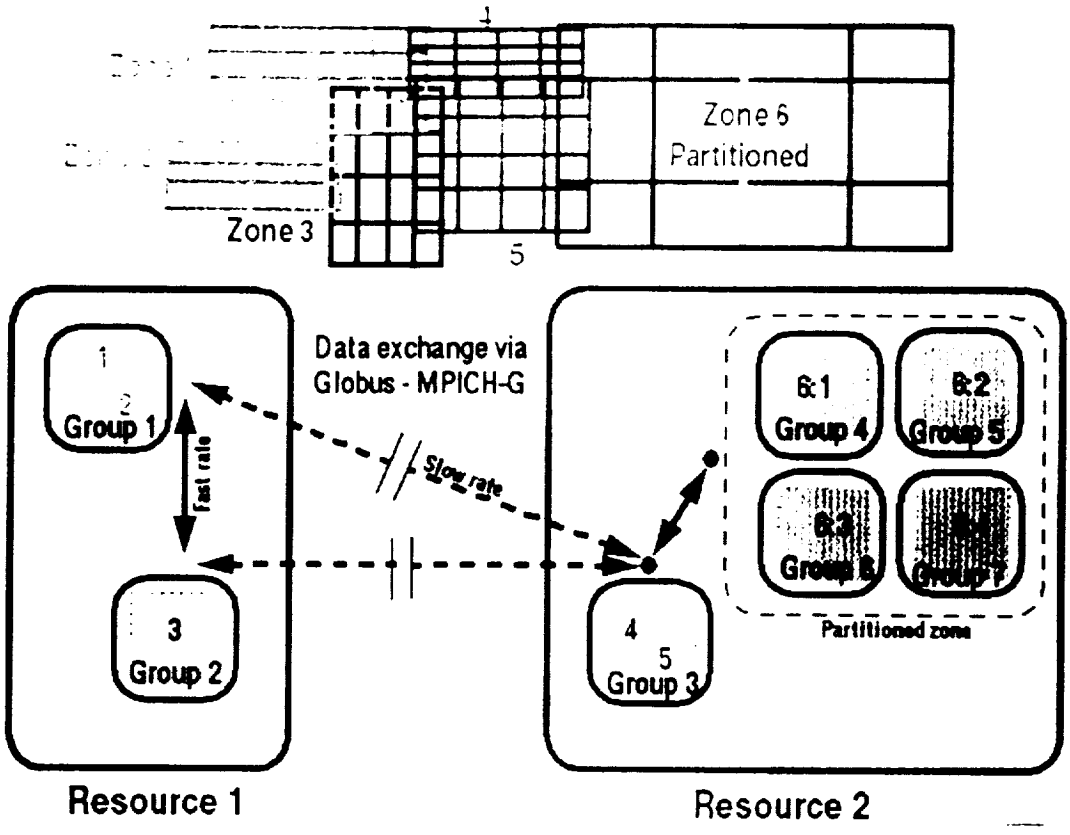

Figure 4: Schematic of OVERFLOW distributed computing.

Figure 5: Isometric view of wing-body overset grids in 12-Foot PWT.

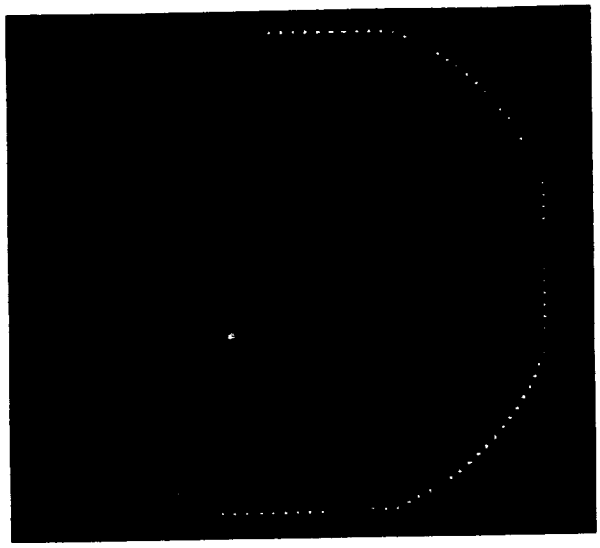

Figure 6: Isometric view of wind tunnel flow solution (pressure).

Machine	CPUs	Walltime (sec/step)	MFLOPS	Comm. time (sec/step)	
				Avg	Max
Origin 2000	4	68	540	3.8	11
	16	18	2055	3	8.8
	63	7.0	5257	1.6	3.8
	124	4.3	7667	1.1	2.7
Cray T3E	88	17.3	2127	3.5	12.
	271	7.2	5111	1.2	3.8
	370	7.9	4658	1.6	4.1
	492	6.8	5411	1.5	3.7

Table 1: Parallel performance for MEDIUM Test Case: 9 million grid points.

Machine	CPUs	Walltime (sec/step)	MFLOPS	Comm. time (sec/step)	
				Avg	Max
Origin 2000	16	52	2650	12	24.6
	48	28.9	4768	5	12.5
	96	19.7	6994	4.5	9.3
	124	20.1	6855	5.5	10.9
Cray T3E	203	34.2	4025	7.2	20.4
	299	31	4432	11.7	25.5
	400	22.4	6043	5.2	13.3
	510	18.1	7613	4.8	12.2

Table 2: Parallel performance for LARGE Test Case: 33 million grid points.

Number of Processors			Walltime (sec/step)	Comm. Time (secs/step)	
Evelyn (Ames)	Whitcomb (Langley)	Sharp (Glenn)		Min	Max
1	1	0	195	15	48
2	0	0	175	1	27
2	1	1	98	5	30
4	0	0	91	0.7	29
2	5	1	52	1	18
8	0	0	43	0.3	9
2	12	2	32	2	16
0	16	0	23	0.1	13
8	8	8	26	2	13
0	0	24	15	0.2	8

Table 3: Parallel distributed performance for MEDIUM test case, using IPG-Globus. Results on a single resource for the same number of processors are also presented for comparison purposes.