Parallelism and OVERFLOW

Dennis C. Jespersen[1]

NAS Technical Report NAS-98-013 October 1998

jesperse@nas.nasa.gov
MS T27A-1
NASA/Ames Research Center
Moffett Field, CA 94035-1000

Abstract

The computer code OVERFLOW is widely used for the numerical solution of the Navier-Stokes equations. This report describes work toward the goal of parallelizing the code. The emphasis is on explicit message-passing. Two message-passing versions of OVERFLOW have been developed. The major difference between the two versions is that one of them allows zones to be split across processors while the other doesn't. The version that disallows zonal splitting was easier to code and is easier to maintain, but can suffer bottleneck problems if one zone is significantly larger than any other. The version that allows zonal splitting was harder to code and is harder to maintain but allows for better load-balancing. The floating-point performance of either parallel version is essentially determined by the single-node performance of the code.

[1]NAS Systems Division

1 Introduction

The computer code OVERFLOW is widely used in the aerodynamic community for the numerical solution of the Navier-Stokes equations. Recent trends in computer systems and architectures have been toward multiple processors and parallelism, including distributed memory. This report describes work that has been carried out by the author and others at Ames Research Center with the goal of parallelizing OVERFLOW on a variety of parallel architectures.

This report begins with a brief description of the OVERFLOW code. This description includes the basic numerical algorithm, some software engineering considerations, and some single-processor performance figures.

Next comes a description of a parallel version of OVERFLOW using PVM (Parallel Virtual Machine). This version of OVERFLOW, called OVERFLOW/PVM, uses the manager/worker style and is part of the standard OVERFLOW distribution.

Then comes a description of a parallel version of OVERFLOW using MPI (Message Passing Interface). This parallel version of OVERFLOW, called OVERFLOW/MPI, uses the SPMD (Single Program Multiple Data) style and MAY soon be part of the standard OVERFLOW distribution.

Finally comes a discussion of alternatives to explicit message-passing in the context of parallelizing OVERFLOW.

2 Basics of the code

OVERFLOW is a computer code for the numerical solution of the Navier-Stokes equations, oriented toward applications in aerodynamics [1],[2],[3]. OVERFLOW uses finite differences in space on structured meshes, implicit time-stepping, and general overlapping grids for dealing with complex geometries [4]. The implicit time-stepping is usually in the form of a three-factor ADI (Alternating Direction Implicit) scheme; the factors can be either scalar pentadiagonal, scalar tridiagonal, or block tridiagonal. A two-factor LU scheme is also available; with this option the implicit equations are approximately solved with a symmetric Gauss-Seidel iteration. The spatial differencing options include central differencing with artificial dissipation, and an implementation of Roe upwind differencing. For steady-state problems, time steps can be chosen locally based on a CFL (Courant-Friedrichs-Lewy) criterion, for faster convergence, and a multigrid convergence acceleration scheme is available.

Turbulence models are an essential part of the OVERFLOW code in practice. Available models include an algebraic turbulence model (Baldwin-Lomax), one-equation models (Spalart-Allmaras or Baldwin-Barth), and a two-equation (k-ω) model. The one- and two-equation models are field-equation models: a convective transport equation has to be solved for the evolution of a certain quantity or quantities. This equation is solved with finite differences and implicit time-stepping much as the transport of the conservative quantities is handled. The flow equations and turbulence transport equations are handled in a loosely coupled fashion: first one time step is taken for the turbulence field equation, then one time step is taken for the flow equation, using the latest turbulence model quantities.

Structured meshes can be easily wrapped around simple geometrical compo-

nents such as an isolated wing or fuselage. With more effort geometries such as a wing/fuselage combination can be gridded with a single structured mesh. But general complex geometries cannot be handled with a single structured mesh. OVERFLOW handles general geometries by allowing overset or "chimera" meshes. In this approach individual meshes are generated about geometrical components such as wings, nacelles, or pylons, and allowed to overlap one another or to cut holes in one another in an arbitrary fashion. Boundary data needed at outer or hole boundaries of a given component mesh is obtained by interpolation from another mesh. This overset mesh capability is essential for OVERFLOW in practical situations.

The overset idea was advocated by the late J. Steger. The chimera grid approach was jointly developed in the 1980's by the Air Force Arnold Engineering Development Center for store separation problems and for tunnel support interference evaluation, and also by researchers at NASA Ames Research Center and NASA Johnson Space Center for evaluating the Space Shuttle launch vehicle. The OVERFLOW code itself is authored by P. Buning. OVERFLOW has a demonstrated capability to solve flow problems with a large number of unknowns in complex geometries [5],[6].

Any discussions of the parallel performance of a code should be prefaced by an assessment of the single-processor performance of that code. The OVERFLOW code is written in a vector-oriented style, with the preferred architecture for execution of the program being a Cray-class supercomputer such as a Cray-C90. On such an architecture the code performs extremely well, as essentially all arithmetic operations vectorize. The computation rate for a typical test problem with 210000 grid points and a Spalart-Allmaras turbulence model was 423 Mflop/sec, this rate being given by the hardware performance monitor on the C90. This is a single-CPU performance number; multitasking will be discussed later.

The single-processor computation rates that will be given now were all determined by scaling relative to the C90 computation rate. The same problem was run on a variety of different machines and the execution time per step was noted. This was compared to the execution time per step on the Cray-C90, and an Mflop/sec rating for the non-Cray machine was defined by scaling the C90 Mflop/sec rating by the ratio of the execution times. For non-Cray machines, the code was compiled in double precision so that 64-bit arithmetic was performed in all cases. An effort was made to compile the code with maximal optimization on each platform. All the Mflop rates should be viewed as approximations that are only valid to within 5 or 10%.

Machine	Mflop/sec	Comments
Cray J-90	85	Vector architecture, 100 MHz
IBM SP2 node	50	RS6000/590 processor, 66.7 MHz
SGI Indigo^2	14	MIPS R4000 processor, 150 MHz
SGI Power Challenge	50	MIPS R8000 processor, 75 MHz
SGI Origin2000 node	88	MIPS R10000 processor, 195 MHz
Cray-T3E node	20	DEC Alpha processor, 300 MHz

Table 1: Single-Processor Performance of OVERFLOW

From the data in Table 1 and many other similar experiments one can conclude that OVERFLOW performs very well on vector architectures such as the Cray C-90 and Cray J-90, running at about 45% of peak performance there. The situation is much less favorable on machines with hierarchical memory. On such machines the cache becomes a bottleneck and OVERFLOW typically runs at (scaled) rates that are 10–20% of peak. The effect of the cache size can be dramatically illustrated by the difference in performance between the nodes of the Origin2000 and the Cray-T3E. The nodes of the Origin2000 studied here have a clock rate of 195MHz, with a 32 Kbyte primary cache and 4 Mbyte secondary cache. The peak performance is 390 Mflop/sec. The nodes of the Cray-T3E studied here have, by contrast, a clock rate of 300 MHz, peak performance of 600 Mflop/sec, and a primary cache of 8 Kbytes and secondary cache of 0.096 Mbytes. The difference in cache size accounts for the better performance of OVERFLOW on the Origin2000 node, despite the slower clock rate.

OVERFLOW is not unique in its disappointing performance on hierarchical memory machines. In fact, the performance of OVERFLOW is typical of CFD codes on cache-based machines [7].

OVERFLOW allows multitasking on shared-memory multiprocessor machines such as the Cray-C90 or SGI Origin2000. The multitasking is accomplished by two mechanisms. The first mechanism is explicit multitasking directives; different computational planes L = constant or K = constant are given to different processors. This multitasking can be thought of as a medium-grain multitasking, as it is above the simple do-loop level and below the zonal level. This is currently applicable to Cray and SGI shared-memory multiprocessor machines, but should be easily extendible to the OpenMP [8] syntax and so could be used on a wide variety of multiprocessor machines. The second mechanism is simply to let the compiler discover multitasking in certain subroutines which consist of a triple loop over all grid points. Care is taken so that subroutines which are called by subroutines with explicit multitasking directives are not themselves multitasked.

Considerations of software engineering play an important role in working with OVERFLOW. Maintainability, readability, portability, and ease of revision are all important criteria. OVERFLOW is a fairly large code. A recent snapshot of the code counted approximately 84000 lines of Fortran code (Fortran77) and 1500 lines of C code. Of the lines of Fortran code, about one-third are comment lines, and of these comment lines about one-half are empty comment lines, used to enhance readability. The snapshot revealed 884 Fortran subroutines spread over 867 source files.

OVERFLOW is coded in a very clean, straightforward Fortran style. To a large degree it can be considered an "object-oriented" code, as many of the numerical subroutines are designed to do just one simple operation on either the whole dataset or on a two-dimensional slice of the dataset. This style of coding is easy for humans to read and understand, and the modularity is in accordance with one of the key principles of software engineering. The performance is very good on vector architectures. The coding style may contribute to the disappointing performance on cache-based machines. On a cache-based machine, it is desirable to perform many arithmetic operations on data once that data has been fetched into cache. It would be possible to recode OVERFLOW to operate this way, but this would involve a large number

of changes and would probably degrade the modularity, readability, maintainability, and object-oriented nature of the code.

OVERFLOW is a user-driven code, that is, most of the features in OVERFLOW are there because of user request. This implies that parallelization efforts, to be widely useful, have to address most or all of the components of the code (turbulence models, boundary conditions, implicit solvers). Parallelization of a small subset of the code would be of limited use.

A further point to make is that OVERFLOW does not exist in isolation. There are at least 75 auxiliary programs that are helpful to the user of OVERFLOW in the areas of grid manipulation, data file manipulation, and pre- and post-processing of data. Most of these programs are modest-sized utility programs involved with grid manipulation, but the essential program Pegsus, used to "glue" together various component meshes, is itself a good-sized program of some 20000 lines. Also essential, but not included in the set of auxiliary programs considered above, is at least one grid generation program, which has the task of producing a volume grid from a given surface mesh. The point here is that there is more to OVERFLOW than just the code itself, there are a number of other tools that are needed to produce a useful package.

Finally, OVERFLOW is a dynamic package, it is not a fixed target. It is continually evolving, so additions and modifications to the package must be intelligible to others who make modifications. Ideally, modifications to the package intended for parallelism should be minimal and "orthogonal" to the bulk of the serial code.

3 Explicit message-passing with PVM

Essentially all practical problems given to OVERFLOW involve multiple meshes, or zones. In such problems, the natural unit of parallelism is the individual zone. With this in mind, the obvious way to parallelize the code involves assigning different zones to different processors. It should be noted right away that this implies a change to the basic OVERFLOW algorithm.

Specifically, the heart of the code, after initialization, is this loop:

```
Loop over zones
    Read chimera boundary information
    Take one time step
    Write chimera boundary information
End loop
```

Here, "chimera boundary information" means flow variable information from zones which supply boundary data to the current zone. Thus each zone uses the most recent chimera boundary information. An ordering is imposed upon the zones and the iteration can be thought of as a "Gauss-Seidel" ("successive updates") iteration, where the most recent information is used whenever possible.

The corresponding portion of the parallel code is logically equivalent to this:

```
Parallel loop over zones
    Read chimera boundary information
```

```
    Take one time step
End loop
Parallel loop over zones
    Write chimera boundary information
End loop
```

With this logic, all zones at time step N use chimera boundary data from time step $N-1$. This can be thought of as a "Jacobi"-type ("simultaneous updates") iteration. It can be argued that this is a more natural way of treating a multiple-zone problem, putting all the zones on an equal footing, while the serial code imposes an arbitrary ordering on the zones. In fact, if the zones are reordered then the serial code will give different answers (of course a steady state will not depend on the ordering of the zones, only the transient behavior will change) while the output from the parallel code will not change.

It is possible to imagine parallelizing the code strictly at the zonal level, that is, treating the zones sequentially and parallelizing only within each zone. Though this approach fails to exploit the natural zonal level of parallelism, it would have the advantage of exactly reproducing the results of the serial code. But there are some drawbacks to this approach. Suppose we have a case with 40 zones, 10 million grid points, the largest zone has 1 million grid points, and the smallest zone has 50000 grid points. Furthermore suppose that the processors under consideration have 128 Mbytes of memory each and that we are computing in 64-bit arithmetic. If we want enough memory to hold all the data, this case would require (OVERFLOW requires about 33 words of memory per grid point) at least 2.64 Gbytes of memory, or at least 21 processors; to be on the safe side we might want about 30 processors. Then either each zone will be worked on by all 30 processors, leading to a high communications overhead for the smaller zones (on the smallest zone each processor would have fewer than 2000 grid points), or else some processors could be idled when the smaller zones were being computing, leading to inefficient processor utilization. Alternatively, if we require only enough memory for the largest zone, then we would need 3 or 4 processors, which would necessitate reading and writing each zone's data from and to disk at each time step. This would probably seriously slow down the solution process. Despite these drawbacks, an early data-parallel version of OVERFLOW on the Connection Machine [9] used this approach with moderate success.

Returning to the model where each zone is treated independently and in parallel, it is important to emphasize that this changes the algorithm from a "Gauss-Seidel"-type algorithm to a "Jacobi"-type algorithm. This may or may not have significant consequences, depending on the particular problem and input parameters under consideration. For some cases the difference between the two algorithms is barely noticeable. For other cases [10] the "Gauss-Seidel" algorithm runs while the "Jacobi" algorithm with the same parameters diverges. It is easy to construct a simple example equation $du/dt + Au = 0$ with a 2×2 symmetric matrix A such that, using Euler explicit time stepping, the stability interval for Gauss-Seidel iteration is bigger than the stability interval for Jacobi iteration. This means that for some values of Δt the Gauss-Seidel algorithm is stable while the Jacobi algorithm is unstable. It is natural to suspect that for OVERFLOW there might be cases where the serial algorithm is just at the edge of some stability boundary, and switching to

the parallel algorithm might cause instability. In such cases one would hope that a slight decrease in the time step would restore stability to the parallel algorithm.

A pilot effort aimed at showing the feasibility of parallelizing OVERFLOW was begun in the early 1990's by M. Smith and C. Attwood of Ames Research Center. They used PVM and had in mind an environment of networked workstations (NOW, Network of Workstations). The basic idea was to distribute separate zones to separate processors, without splitting zones across processors. They coded the parallel algorithm in a manager/worker style. One important thing to note is that their code was an offshoot of the main OVERFLOW code and was not integrated back into the OVERFLOW package.

With that project as an example, this author began an effort at producing a parallel version of OVERFLOW with the following major characteristics.

1. The code would be integrated in the main OVERFLOW package and would automatically be distributed as part of OVERFLOW.

2. The code would be written in the manager/worker style exploiting parallelism at the zonal level.

3. The code would be fault-tolerant, in the sense that the code could recover from the failure of a worker.

4. There would be no splitting of zones across processors, and each worker would handle a single zone.

5. The various output files produced by the serial version of OVERFLOW (residuals, forces and moments, minimum density and pressure, turbulence residuals) would also be produced by the OVERFLOW/PVM code.

The rationale for item 3 above was that the target "machine" for this effort was a loosely coupled network of workstations at a company or research organization where control of the individual machines might not be centralized, and an individual workstation might be rebooted at any time. Given this, the implementation was in PVM since, at the time that project was begun, PVM supported dynamic process management while MPI did not. The first part of item 4 was also dictated by the loosely coupled target architecture, with possibly low bandwidth and high latency. Splitting zones across processors would necessitate much more interprocessor communication, specifically, linear systems distributed across processors would have to be handled. The practical effect of this "no splitting" criterion is that the new code amounted essentially to a "shell" around the outside of OVERFLOW and there was no modification of the lower-level internals of OVERFLOW. The second part of item 4 was for ease in coding. The last item was for user comfort: the parallel code should produce familiar output. This should tend to promote acceptance of the parallel version. The manager/worker style was natural given that fault-tolerance was a goal.

The goal of integrating the parallel code into the main OVERFLOW package was reached by modifying selected OVERFLOW subroutines with conditional compilation blocks ("`#ifdef PVM`") that would be activated only if compiling for the PVM target. A total of just 15 OVERFLOW subroutines were modified in this way. The

rest of the new code was separated into 60 subroutines with about 6500 total lines of code. The OVERFLOW subroutines that needed modification fell into three classes: startup, shutdown, and output. Of course startup and shutdown routines needed modification, and the routines dealing with output files needed to be modified (in accordance with item 5 above) so that the workers would send their information about residuals, forces, etc., to the manager; the manager process collects all this information and prints it out.

The low number of modified OVERFLOW subroutines and the absence of modification of the lower-level internal numerical subroutines greatly eases the burden of maintaining the OVERFLOW/PVM code as the OVERFLOW package is continually modified and upgraded.

The most serious problem with OVERFLOW/PVM stems from the prohibition on splitting zones across processors. If a zone has too many grid points for the memory of any of the processors, then either the code performance will suffer greatly due to swapping on the node that has too many grid points, or else the code will completely fail to run (if the operating system on the node lacks virtual memory). For a node with 128 Mbytes of memory, the maximum number of grid points that can in practice be used without causing swapping is about 400000. There are many practical cases in which one or more zones has in excess of 1 million grid points. In order for the OVERFLOW/PVM package to be used for such cases, manual regridding (a tedious and time-consuming procedure) is usually necessary. (A program called BREAKUP from Daniel Barnette, `dwbarne@cs.sandia.gov`, at Sandia National Laboratories may be able to aid in this task.)

Even if there is sufficient memory on all the nodes there is a fundamental problem having to do with zone imbalance and maximum possible speedup. Suppose, for example, that a given problem has one zone with half of the grid points, while the other half of the grid points are spread among several zones. Then no matter how many processors are used, the speedup for OVERFLOW/PVM cannot be more than 2. This is simply because the largest zone will always be a bottleneck if zonal splitting is not allowed. This indicates that the OVERFLOW/PVM approach is best suited for cases where the grid is such that (or can be generated such that) there is no one zone with a large plurality of the grid points.

There is an apparent difficulty with OVERFLOW/PVM which is not really a problem. This has to do with the problem of small zones. With a strict policy of one zone corresponding to one worker, it seems as if load balancing would be impossible if the geometry in question had one or more small zones along with one or more medium-size or large zones, since some workers would have only a small number of grid points while other workers would have many more grid points. This apparent difficulty is circumvented by assigning several copies of the worker program to a single node. In the NOW environment this is easy to do by explicitly starting up several copies of the worker program on a given named node; it is only slightly inefficient to have several copies, each with one zone, as opposed to one copy with several zones. In a homogeneous multiple-processor environment where the nodes are thought of as anonymous processors, this is harder but may not be impossible. For example, in an environment under the control of the Portable Batch Scheduler PBS [11] it is possible to learn the identities of the nodes given to a batch task and

to adjust the input file to start up multiple copies of the worker program on one or more of the nodes.

The question of how the workers should communicate their chimera boundary information among one another has at least two possible answers. The first approach has all chimera boundary information channeled through the master. The main worker routine in this approach is essentially this:

```
Loop on number of steps
    Receive chimera boundary data from manager
    Take one time step
    Send chimera boundary data to manager
End loop
```

This approach has the advantage of requiring minimal changes to the serial code. It has the disadvantages of channeling all chimera boundary information via the manager (potentially leading to communication contention) and of requiring twice as much total chimera communication per step (each piece of chimera data is sent from one worker to the manager and also sent from the manager to another worker).

A second approach was pioneered by C. Attwood. This approach, called "peer-to-peer" communication, sends chimera data directly from one worker to another. In this approach the main worker routine is essentially this:

```
Loop on number of steps
    Receive chimera boundary data from other workers
    Take one time step
    Send chimera boundary data to other workers
End loop
```

This approach has the advantage of reducing the total amount of communication required for a problem. It has the small disadvantage of requiring a preprocessing step to determine, for each worker, which other workers are to be sent information and which other workers will be sending information. This method is in OVERFLOW/PVM and is the method of choice.

Here are some examples of performance for OVERFLOW/PVM. To illustrate the bottleneck problem, consider a 6-zone wing/body case with 1 million grid points, the largest zone having 40% of the total. On an SGI Power Challenge machine, the serial code took just under one minute per time step. On 4 SGI Power Challenge nodes, the same problem ran at just under 0.5 minutes/step, a speedup of 2. Compare this with the maximum speedup of 2.5 for this problem (since the largest zone has 40% of the points).

Some more sample runs were made on a simplified aircraft geometry with about 2.2 million grid points. There were originally 6 zones, the largest zone having 720000 grid points. The grid was modified by hand into a 10-zone case, the largest zone having 280000 grid points, and also modified into a 28-zone case, the largest zone having 108000 grid points. These cases were run on an IBM SP2 and a network of workstations (SGI R4000 machines). For the 6-zone case, care was necessary to ensure that the two biggest zones (720000 and 470000 grid points, respectively) were assigned to processors with sufficient memory.

Case	Sec/step	Comments
SP2, 6 zones	72	Largest zone is bottleneck
SP2, 10 zones	29.5 sec/step	No peer-to-peer
SP2, 10 zones	27 sec/step	Peer-to-peer + barrier
NoW, 10 zones	105 sec/step	No peer-to-peer
SP2, 28 zones	7.5 sec/step	Peer-to-peer + barrier
NoW, 28 zones	70 sec/step	12 nodes, peer-to-peer + barrier
NoW, 28 zones	45 sec/step	28 nodes, peer-to-peer + barrier
NoW, 28 zones	35 sec/step	28 nodes, peer-to-peer, no barrier

Table 2: Performance of OVERFLOW/PVM

In Table 2 we have noted two refinements of the peer-to-peer strategy. In one case the processors are synchronized by an explicit barrier, in the other case the processors are synchronized but only implicitly. On a machine with a fast network, such as the SP2, there is not much difference between the explicit and implicit synchronization, but there is a significant difference between the two methods on the NoW.

Fault tolerance is implemented in OVERFLOW/PVM by using the standard capabilities of PVM. The manager is able to learn if any worker process exits (e.g., if a particular workstation is rebooted). If this should occur, the manager signals all the other workers to pause, starts up a new worker process on a new node or on the least-loaded node, and sends out restart data from the most recent checkpoint to all workers. If there is no checkpoint data then all workers have to roll back to the beginning.

It should be emphasized that aside from the fault-tolerance aspect, this parallel version of OVERFLOW could have as easily been coded using MPI as PVM. The term "OVERFLOW/PVM" is simply a convenient way to say "parallel implementation of OVERFLOW using the manager-worker style in which no zone is split across processors".

4 Explicit message-passing with MPI

The problem with the largest zone being a bottleneck for OVERFLOW/PVM motivated the work on OVERFLOW/MPI. This goals of this effort were as follows:

1. As with OVERFLOW/PVM, the code was to be integrated in the main OVERFLOW package and automatically to be distributed as part of OVERFLOW.

2. The code should be able to partition a zone across multiple processors, to alleviate the largest zone bottleneck.

3. The code should be able to cluster several zones onto a single processor, for good load balancing.

4. Every processor handles either one or more full zones, or else it handles just one part of a zone. Thus if a processor has part of a zone, it has no other full zones and no other partial zones.

5. The various output files produced by the serial version of OVERFLOW (residuals, forces and moments, minimum density and pressure, turbulence residuals) would also be produced by the OVERFLOW/MPI code.

The second and third items in the preceding list are controlled by input parameters. Thus it is the user's responsibility to determine a good partitioning for a given problem. It is possible to imagine the code itself doing the partitioning based on the number of processors and the size of each zone, and this may be added to OVERFLOW/MPI in the future.

One difficulty in the area of software engineering is that this was not an incremental project, where one could write a small piece of code and test it, progressing step by step. Rather it was necessary to write a lot of code before any testing could begin. Ferhat Hatay (formerly Research Scientist with MCAT, Inc., now System Engineer with HAL Computer Systems, Inc.) did much of the work in the initial phase of this effort.

We decided to adopt the SPMD (single program, multiple data) paradigm for this project. We also decided to adopt MPI, hoping for better bandwidth and latency on tightly-coupled multiprocessors. We decided to adopt a standard domain decomposition idea, using Cartesian decompositions of computational space.

It should be emphasized at the outset that this parallel version of OVERFLOW could have been coded with PVM. The term "OVERFLOW/MPI" is simply a convenient way to say "parallel implementation of OVERFLOW using the SPMD style and allowing zones to be split across processors".

For program debugging, we adopted a policy of "all bits equal". This means that when OVERFLOW and OVERFLOW/MPI, having been compiled on the same machine with the same compilation options, run a given problem on that machine, then they should produce results that agree exactly to every possible bit. This strict criterion is intended to avoid any subtle bugs.

It turns out that we needed to modify 58 subroutines of OVERFLOW (compared to the 15 that needed modification in the OVERFLOW/PVM project). The rest of the new code was separated into 144 subroutines with almost 20000 lines of code. Allowing for partitioning of a given zone across multiple processors accounts for the bulk of the increased complexity.

Why is it that allowing partitioning increases the complexity so much? The reasons are several, and some of them are nonobvious. Allowing an overlap of "halo" points and sending data from one processor to update a neighbor's halo points is an well-understood idea. OVERFLOW has the additional complexity of using implicit time stepping (explicit time stepping is not an option!), so code which solves linear systems (scalar pentadiagonal, scalar tridiagonal, block tridiagonal) spread across processors must be included. There are several algorithms for solving sparse banded linear systems spread across processors. We chose to implement pipelined Gaussian elimination, both one-way and two-way, as our solvers, and here there are tradeoffs that can be investigated in terms of the number of messages sent and the size of

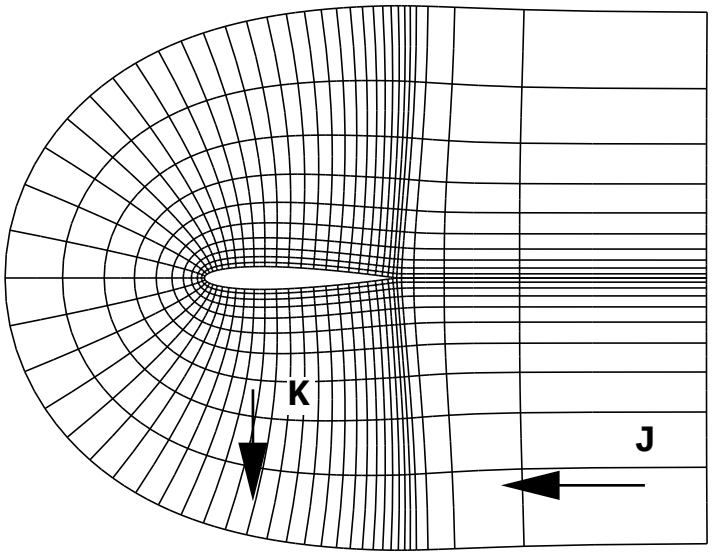


Figure 1: C-mesh around a two-dimensional airfoil

each message. Periodic solvers would add an additional level of complexity (these are not yet implemented into OVERFLOW/MPI).

One nonobvious reason for the increased complexity has to do with boundary conditions. A very common boundary condition in aeronautics applications is the wake cut boundary condition in a C-mesh. Figure 1 shows a schematic of a C-mesh around a two-dimensional airfoil. In this geometry, the J index starts at the right-hand boundary and increases proceeding clockwise, while the K index starts at the airfoil (and wake cut boundary) and increases proceeding outwards. With this setup, a single physical point on the wake cut line corresponds to two distinct computational points. The values of flow variable at these two computational points are, in OVERFLOW, determined by averaging the values from the adjacent points in physical space above and below the cut. Figure 2 shows the situation more explicitly. In Figure 2, the point A with physical coordinates (x_A, y_A) has two distinct computational points associated with it; the coordinates of these computational points are $(j, 1)$ and $(jmax + 1 - j, 1)$, where $jmax$ is the number of points in the J direction. The boundary condition in OVERFLOW for a flow variable q at point A is $q_A = (q_B + q_C)/2$, i.e.,

$$q(j, 1) = (q(jmax + 1 - j, 2) + q(j, 2))/2.$$

The boundary condition is nonlocal in computational space, as the point with coordinates $(jmax + 1 - j, 2)$ is not a neighbor of $(j, 1)$ in computational space. If the grid is partitioned so that the grid points in the lower part of the wake go to a different processor from the grid points in the upper part of the wake, then the wake cut boundary condition will require interprocessor communication. In fact, any boundary condition that is nonlocal in computational space may require interprocessor communication. There are other boundary conditions in OVERFLOW that are nonlocal in computational space, but they are not much used and we have not

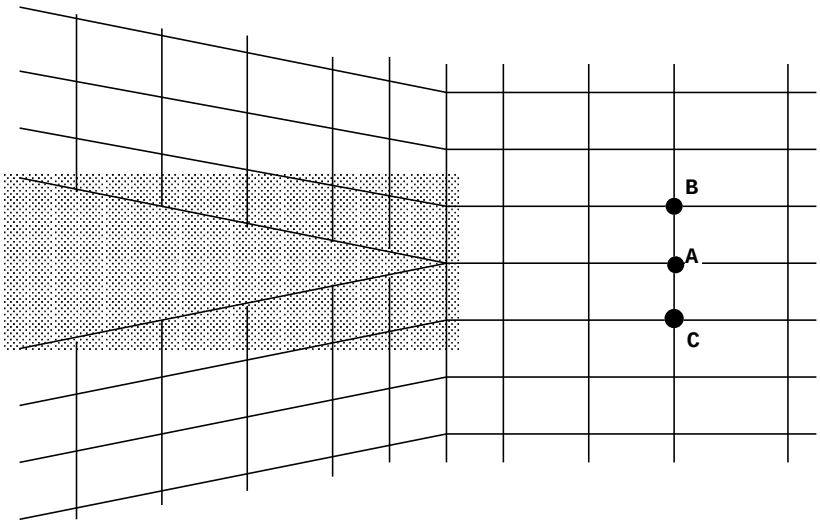


Figure 2: Closeup near trailing edge of airfoil

implemented them. We have implemented the C-mesh nonlocal boundary condition but none of the other nonlocal boundary conditions.

Another boundary condition often used in practical problems is the "read from file" boundary condition. Here a given file is used to determine the boundary values. What happens if the boundary face that corresponds to the boundary file is split across processors? We have implemented this boundary condition into OVERFLOW/MPI as follows. The given boundary file is preprocessed into two or more separate files, so each processor has its own boundary condition file to read. Of course, this can create complications on a true distributed memory architecture (the appropriate files need to be accessible by the appropriate processors).

The question of chimera boundary conditions arises for OVERFLOW/MPI as it did for OVERFLOW/PVM. For OVERFLOW/MPI there is no manager, so it is natural for the workers to communicate directly with one another. But what if a zone is partitioned across multiple processors and some of these processors need to read or write chimera boundary data? Our solution is to designate one of the processors as the "local master", and to have all chimera boundary data communication occur via local masters. Thus each local master has to collect chimera boundary data from processors in its group, send the appropriate pieces of it to certain other local masters, receive chimera boundary from some other local masters, and distribute the chimera data appropriately to the processors in its group.

It was also tedious to correctly implement the OVERFLOW multigrid algorithm if a processor is split across multiple zones. The main difficulty was to ensure that the halo points were correctly updated before every restriction (from a fine level to a coarser level) or interpolation (from a coarser to a finer level) operation. We have not worried about possible inefficient processor utilization in multigrid when the number of points on a level becomes small.

The code has been tested on SGI workstations (with the MPICH implementation), on the IBM SP2, Cray-T3E, Cray-J90, and on several SGI multiprocessor platforms (Power Challenge, Onyx2, Origin2000).

Now we consider performance for OVERFLOW/MPI. First we study speedup. The test case is a $69 \times 61 \times 50$ mesh (single zone, 210450 points). The case was run on 1,2,4, and 8 nodes on 3 parallel machines: IBM SP2, SGI Origin2000, and Cray T3E. With 2 nodes the mesh was partitioned $2 \times 1 \times 1$, with 4 nodes the mesh was partitioned $2 \times 1 \times 2$, and with 8 nodes the mesh was partitioned $4 \times 1 \times 2$ (the middle dimension has a periodic boundary condition and partitioning in a periodic direction is currently not allowed). The data are given in Table 3. They show respectable speedups for the code, especially in light of the fact that 210450 grid points are not too many to begin with, and that spreading them over 8 processors gives each processor about 26300 grid points, quite a small number.

Machine	No. of nodes	Sec/step	Speedup
SP2	1	14.74	
	2	8.55	1.72
	4	4.75	3.11
	8	2.81	5.24
Origin2000	1	7.99	
	2	4.73	1.69
	4	2.91	2.74
	8	1.83	4.38
T3E	1	20.15	
	2	10.90	1.85
	4	5.43	3.71
	8	3.42	5.90

Table 3: Speedup for OVERFLOW/MPI

In the second case for OVERFLOW/MPI performance, we show scaled speedup. Here we start out with the same original $69 \times 61 \times 50$ mesh, and for the same geometry generate new meshes with 2,4, and 8 times the number of mesh points. Thus the meshes have approximately 210K, 420K, 840K, and 1680K points. These meshes are run with 1,2,4, and 8 processors, respectively, so the number of grid points per processor remains constant. In Table 4 efficiency is defined as $T(1)/T(N)$, i.e., time for 1 processor divided by time for N processors.

Table 4 indicates that the SP2 scaled efficiency degrades gracefully as the number of nodes is increased. Also, the T3E efficiency degrades slowly but the single-node performance of the T3E is significantly slower than the single-node performance of the SP2 or the Origin2000. Finally, the Origin2000 efficiency suffers a sharp drop in going from 2 to 4 processors, probably because of the architecture of that machine which features 2 processors attached to a shared memory as a single "node".

5 Beyond explicit message-passing

In this section I will try to take a look at explicit message-passing as a general parallelization tool in the context of OVERFLOW, consider its strengths and weaknesses,

Machine	No. of nodes	Sec/step	Efficiency
SP2	1	14.74	
	2	17.64	0.84
	4	17.80	0.83
	8	20.32	0.73
Origin2000	1	7.99	
	2	9.38	0.85
	4	14.01	0.57
	8	26.74	0.48
T3E	1	20.15	
	2	22.33	0.90
	4	24.81	0.81
	8	28.72	0.70

Table 4: Scaled Speedup for OVERFLOW/MPI

and consider possible alternatives.

The book [12] lists as one of the strengths of explicit message-passing:

> The most compelling reason that message passing will remain a permanent part of the parallel computing environment is performance. ... Message passing provides a way for the programmer to explicitly associate specific data with processes and thus allow the compiler and cache-management hardware to function fully.

Consider the following slightly modified version of that point:

> The most compelling reason that assembly language will remain a permanent part of the computing environment is performance. ... Assembly language provides a way for the programmer to explicitly allow the hardware to function fully.

This may well have been asserted 20 (say) years ago, but assembly language is rarely hand-written nowadays. The tedious job of creating assembly language has been handed to compilers, which are generally very good at it.

It is my contention that writing explicit message-passing code is akin to writing assembly language and that this is usually not an appropriate use of a programmer's time and talent. If explicit message-passing is to be used it should be produced automatically, either as part of some higher-level programming language or else via some automatic tool with user input. At least one such tool is being developed (CAPTools [13]).

Now consider current trends in high-performance computer systems. The trend seems to be away from pure distributed memory systems and toward some form of shared memory, perhaps distributed shared memory where the memory is physically distributed across processors but logically shared. There are even software systems that can run on a set of workstations and make the separate workstations appear to have a shared memory ([14],[15]).

The partitioning of OVERFLOW/MPI is necessary for good performance, but experience shows that the partitioning creates many headaches in terms of code reliability and maintability. So it would be fruitful to look for another way to exploit parallelism.

Jim Taft has proposed such a method [16]. In this method, which he calls "multilevel parallelism", there is no explicit message-passing, yet parallelism can be exploited at the zonal level and at the intrazonal level. The zones are clustered into groups (a group consists of one or more zones) and a "master" process is forked for each group. Each forked process also has one or more threads associated with it. The masters proceed in parallel, and each master utilizes its given number of threads along with the medium-grain multitasking directives already in OVERFLOW. Thus there is parallelism at the level of the masters and parallelism beneath each master. Some sort of distributed shared memory is assumed for the underlying architecture.

This idea has some distinct advantages. It avoids all explicit message-passing and it uses the existing multitasking features of OVERFLOW. There is less low-level code modification necessary, even less than in OVERFLOW/PVM; only the low-level portions of the code that deal with writing files of residuals, etc., need to be modified. This idea should be portable to a variety of machines, especially since the medium-grain multitasking is compatible with OpenMP and since some sort of distributed shared memory seems to be more and more common.

6 Summary

The OVERFLOW code has been parallelized with explicit message-passing using two distinct methods. The first method (OVERFLOW/PVM) uses a manger/worker style and allows no splitting of zones across nodes. The second method (OVERFLOW/MPI) uses an SPMD style and allows zonal splitting and coalescing. The former version is part of the standard OVERFLOW distribution, and the latter version is ready to be part of the standard OVERFLOW distribution.

OVERFLOW/MPI, because it allows zonal splitting, is more efficient than OVERFLOW/PVM, but this efficiency comes at a high price in terms of software development cost and code maintainability. This seems to be inherent in explicit message-passing.

Newer approaches using distributed shared memory and multitasking directives have great promise and should be vigorously pursued.

Finally, a major performance issue with OVERFLOW today is the suboptimal performance on hierarchical memory machines. There is no obvious easy way to substantially improve the performance of OVERFLOW on cache-based machines without drastic recoding, which would probably have significant adverse impact on the modularity, readability, maintainability, and object-oriented nature of the code.

References

[1] P.G. Buning, I.T. Chiu, S. Obayashi, Y.M. Rizk, and J.L. Steger, "Numerical Simulation of the Integrated Space Shuttle Vehicle in Ascent", AIAA-88-4359-

CP, AIAA Atmospheric Flight Mechanics Conference, August 1988, Minneapolis, MN.

[2] K.J. Renze, P.G. Buning, and R.G. Rajagopalan, "A Comparative Study of Turbulence Models for Overset Grids", AIAA-92-0437, AIAA 30th Aerospace Sciences Meeting, Reno, NV, Jan. 6–9, 1992.

[3] M. Kandula and P.G. Buning, "Implementation of LU-SGS Algorithm and Roe Upwinding Scheme in OVERFLOW Thin-Layer Navier-Stokes Code", AIAA-94-2357, AIAA 25th Fluid Dynamics Conference, Colorado Springs, CO, June 1994.

[4] J.A. Benek, P.G. Buning, and J.L. Steger, "A 3-D CHIMERA Grid Embedding Technique", AIAA-85-1523-CP, July 1985.

[5] J.P. Slotnick, M. Kandula, and P.G. Buning, "Navier-Stokes Simulation of the Space Shuttle Launch Vehicle Flight Transonic Flowfield Using a Large Scale Chimera Grid System", AIAA-94-1860, AIAA 12th Applied Aerodynamics Conference, Colorado Springs, CO, June 1994.

[6] L.M. Gea, N.D. Halsey, G.A. Intemann, and P.G. Buning, "Applications of the 3D Navier-Stokes Code OVERFLOW for Analyzing Propulsion-Airframe Integration Related Issues on Subsonic Transports", ICAS-94-3.7.4, Proceedings of the 19th Congress of the International Council of the Aeronautical Sciences (ICAS 94), Anaheim, CA, Sept. 1994, pp. 2420–2435.

[7] M. Yarrow, personal communication.

[8] `http://www.openmp.org/.`

[9] D.C. Jespersen and C. Levit, "A Computational Fluid Dynamics Algorithm on a Massively Parallel Computer", Int. J. of Supercomputer Applications, Vol. 3, No. 4, Winter 1989, 9—27.

[10] M.J. Djomehri and K. Gee, personal communication.

[11] `http://parallel.nas.nasa.gov/Parallel/PBS/index.html`

[12] W. Gropp, E. Lusk, and A. Skjellum, "Using MPI", The MIT Press, 1994.

[13] `http://www.gre.ac.uk/~captool/`

[14] `http://suif.stanford.edu/~scales/sam.html.`

[15] `http://www.cs.rice.edu/~willy/TreadMarks/overview.html.`

[16] J. Taft, "OVERFLOW Gets Excellent Results on SGI Origin2000", NAS Newsletter,`http://science.nas.nasa.gov/Pubs/NASnews/98/01/.`