Hybrid MPI+OpenMP Programming of an Overset CFD Solver and Performance Investigations

M. Jahed Djomehri
Computer Sciences Corporation

Haoqiang H. Jin
NAS Division

NASA Ames Research Center Moffett Field, CA 94035
{djomehri, hjin}@nas.nasa.gov

Abstract

This report describes a two level parallelization of a Computational Fluid Dynamic (CFD) solver with multi-zone overset structured grids. The approach is based on a hybrid MPI+OpenMP programming model suitable for shared memory and clusters of shared memory machines. The performance investigations of the hybrid application on an SGI Origin2000 (O2K) machine is reported using medium and large scale test problems.

1 Introduction

The advent of multiprocessing hardware and software technologies has introduced new challenges into high-performance computing (HPC). Various parallel programming paradigms have been developed on distributed memory (DM) and distributed-shared memory (DSM) systems. Some widely used models are programs based on message passing, such as MPI [20], suitable both on DM and DSM architectures. Other popular paradigms are based on parallel compiler directives, such as OpenMP [11] and HPF [5], where the former exploits shared memory parallelism, and the latter takes advantage of data parallelism. Most of the parallel application codes based on the above approaches use only a single level of parallelism.

With the current trend in HPC hardware towards clusters of shared-memory symmetric multi-processor (SMP) compute nodes, hybrid programming techniques have been introduced that exploit parallelism beyond a single level. The main thrust of these methods is to combine coarse and fine grain parallelism; this is obtained by a domain decomposition and loop-level parallelism, respectively. Communication of data across SMP clusters is achieved via message passing or shared memory referencing, and loop level parallelization by compiler directives. Hybrid approaches have also been applied to various other scientific disciplines, (see e.g. [15] and [8] for hybrid MPI+OpenMP approaches). Our main effort here is to identify multi-level parallelism in a given sequential code, and to incorporate the MPI and OpenMP programming models and interfaces into that code. This effort is substantially less programming intensive for applications whose computational regions already enjoy a natural domain decomposition structure.

A hybrid methodology, named "Shared Memory Multi-Level Parallelism" (MLP) [19] and developed at NASA Ames, uses a fundamentally different approach. MLP exploits shared memory for all data communication via direct memory referencing instructions. MLP has been incorporated into two of NASA's multi-zonal CFD solvers and into a climate modeling code. The performance efficiency of MLP codes has been reported to be very successful on NASA's SGI single image system, 512 CPU R12K processors. The implementation of MLP programming is significantly simpler than the hybrid MPI+OpenMP, in that no extensive message passing library functions are used. In contrast, the MLP library, also developed at NASA Ames, consists of a few routines based on UNIX calls. A comparison of OVERFLOW performance assessment, using MLP versus the MPI-based code, can be found in references [4] and [3]. MLP is currently limited to shared memory architectures.

This report describes the implementation of a hybrid MPI+OpenMP programming model into NASA's high fidelity overset-grid CFD solver, OVERFLOW-D [9, 10]. The objective is to study the scalability of various parallel schemes implemented in this code on different architectures, shared memory and SMP cluster systems, such as O2K and IBM SP. OVERFLOW-D is based on a version of the aerodynamic flow solver OVERFLOW [12], which is designed for complex configurations with static grid systems. The former was primarily developed for moving-body (dynamic) grid systems.

Implementation of our hybrid approach into either the dynamic or the static version of the code would have been similar, but we chose the dynamic version for reasons of availability on several CFD problems of interest at the start of this work. We want to study the performance of our application code as it is applied to large scale test cases, which are more representative of practical problems. Such problems consist of large scale grid systems. Generation of large scale grids for the static application would have been very costly and time consuming. OVERFLOW-D has the capability of automatically generating its own off-body (background) grids that are compatible with near-body grids.

The remainder of this report includes a brief overview of the numerical method, §2, and the hybrid parallelization strategy, §3. Some performance results are presented in §4. The test problem is related to the CFD simulation of complex vortex dynamics.

The results demonstrate the functionality of the hybrid algorithm implemented. The performance results, however, are preliminary and reflect our initial experiences with the new algorithm on an SGI O2K machine. A discussion of future work, a summary and a conclusion are given in §5.

2 Numerical Method

In this section we will briefly review the basics of the flow solver and domain connectivity on structured overset grids.

2.1 Flow Solver

The multi-block, or "multi-zone", overset CFD code, OVERFLOW [12], is popular for high-fidelity complex aerodynamic shapes consisting of multiple geometric components, in which individual blocks of body-fitting overlap grids can be constructed easily about each component. Grid blocks are either attached, near-body, or detached (off-body). The latter is also called a background or a wake grid. The union of near and off-body blocks covers the entire computational domain known as the "Chimera" [18] style decomposition, which falls into the general category of a Schwartz domain decomposition. The code is a Reynolds averaged Navier-Stokes software, augmented with a number of turbulence models. The dynamic code, OVERFLOW-D, simplifies the modeling of bodies in relative motion. For example, in typical rotory-wing problems, the near-field is modeled with one or more grids generated about the moving rotor blades. The code automatically generates cartesian wake grids, called bricks, that encompass the curvilinear near-body grids. At each time iteration, flowfield equations are solved independently on each grid zone in a sequential manner. Overlap boundary (intergrid) data is updated from previous solutions prior to the current time step. Updates are furnished by a Chimera interpolation procedure.

The code uses finite differences in space, and implicit time-stepping on structured grids. A variety of time-stepping and spatial differencing schemes are available. For steady-state problems, faster convergence is achieved by a spatially varying virtual time increment, which is chosen based on a CFL (Courant-Friedrichs-Lewy) number. The code offers a number of user-specified solution algorithms, such as three-factored block tridiagonal, penta diagonal, and Lower-Upper Symmetric Gauss-Siedel (LU-SGS) [17] schemes. Upgrades [13] have been incorporated into newer versions of the dynamic code to improve the temporal and spatial accuracy of the solution scheme. They consist of higher order spatial differencing and the addition of a subiteration scheme at each time-step to reduce factorization errors.

2.2 Overset-grid Connectivity

A domain connectivity program is used to determine the intergrid boundary data, (see Fig. 1), which consists of "holes" and outer boundary points on each grid. Holes are cut in grids which intersect solid surfaces, such as when a portion of an overset grid lies inside a physical body. Adjacent grids are expected to have one-cell overlap to ensure continuity; for higher order accuracy, a two-cell overlap is sought [13]. In the static version, the coordinates of interpolated data are accessed from a pre-processed data file prior to the start of the time step loop. Unlike the static case and due to the relative motion of the grids, a Domain Connectivity Function (DCF) [16] program within the dynamic version is used to compute intergrid donor points that will be supplied to other grids, creating "holes" as needed. The DCF procedure is fully coupled with the flow solver.

3 Hybrid Programming Model

In the following subsection we describe a simple flexible hybrid parallel design which implements a combined MPI+OpenMP model, a two level parallelism, into the OVERFLOW-D application code. The code is typical of multi-zonal CFD code; the approach discussed here can virtually be implemented on all such CFD or other scientific applications. The OVERFLOW-D solver has already been parallelized [21] via an MPI distributed style algorithm, adopting the SPMD (Single Program Multiple Data) style, and is suitable on both the DM and DSM platforms.

Efforts for development of the current hybrid approach primarily consist of integrating and interfacing the OpenMP programming into the OVERFLOW-D code. The task is straightforward, and involves a focused analysis to identify the potential loops in the application code suitable for OpenMP. For large codes, such as for our application, manual analysis to select such loops is time prohibitive and error prone. At the time of this work, a recently developed NASA compiler-based automatic parallelization tool, CAPO [7], has become available. In this work we have made use of this tool along with some manual effort to parallelize some of the loops in OVERFLOW-D.

The combined implementation permits the execution of the OVERFLOW-D code in pure message passing or, as in the true "hybrid" model, with multiple threads per MPI process. The former extends the code capability to run on clusters of SMP. In the following subsections §3.1 and §3.2, we briefly review the MPI and OpenMP programming models of the solver part of the OVERFLOW-D code.

3.1 MPI Implementation

MPI implementation has been developed specifically around the sequential version of OVERFLOW-D and around the multi-block feature of the code, which offers a natural coarse grain parallelism. The main computational logic at the top level of the sequential code consists of a "time-loop", "grid-loop", and a "subiteration loop". The last two loops are nested in the time loop, respectively. Solutions are obtained on the individual grids with imposed boundary conditions, where the Chimera intergrid boundaries are updated successively at the completion of the solution on each grid in the sequence. Upon completion of the grid-loop, the solutions are automatically advanced to next time-step. The overall procedure may be thought of as a "Gauss-Seidel" iteration.

To facilitate parallel execution, a strategy is required to cluster grid components into groups. Each group may contain several grid zones. The grouping is based on a bin-packing strategy that seeks to maintain an even number of total grid points per group, and at the same time retain a degree of connectivity among the grids within a group. This issue is subject to load balancing among parallel processors and will be addressed in another work.

The main change in the logic of the sequential code is to subdivide the grid-loop into two loops, a loop over groups and a loop over the grids within each group. The outer "group-loop" is done in parallel and contains the grid-loop. One MPI process is assigned to each group, with the total number of groups, $N_{MPI-proc}$, equal to the total number of MPI processors invoked at the execution of the code. The intergrid update among grids within each group, named intra-group, is done similarly to the serial case. Chimera updates are also neccessary for overlapping grids across group boundaries, known as inter-group data exchanges, (see Fig. 1). The supply of inter-group donor points from grids in, say, the n^{th} group to grids of the m^{th} group, $m = 1, ..., N_{MPI-proc}$, is stored in a "send" array that will be exchanged by MPI calls. The inter-group exchanges are done at the beginning of a current time-step based on the interpolation data of the previous one.

In the hybrid mode, with multiple threads assigned for each MPI process, only the master thread is responsible for data exchanges as discussed below. MPI processes are synchronized at the completion of the solution over each group. In addition, for problems with moving grids, the DCF program must be invoked to compute new donors prior to the following time-step.

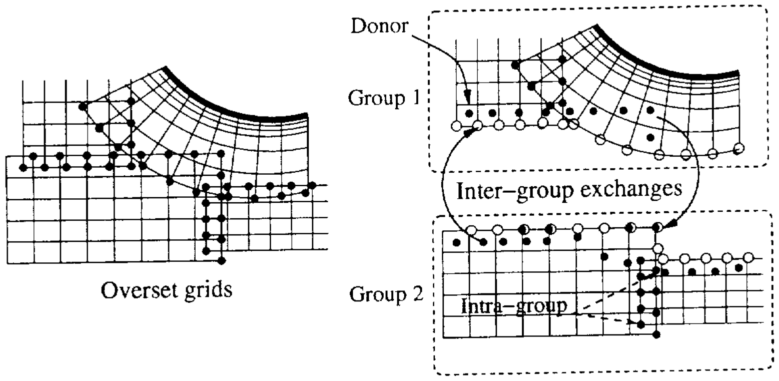

Figure 1: Overset grid connectivity, grouping, interpolation, and data exchanges.

3.2 OpenMP Implementation

The MPI-based scheme only accounts for one level of parallelism; a second level is exploited by fine parallelism using OpenMP compiler directives. Compilers on certain machines, such as O2K, have automatic parallelization options, but the loops chosen tend to be the very innermost loops. This tendency reduces the efficiency of multi-threading. The OpenMP [11] approach offers a superior alternative. The main effort of OpenMP implementation is first to identify parallel loops and parallel regions into the code, and then to insert OpenMP directives along with the proper list of the privatizable and shared variables.

Manual analysis would be tedious and very time-consuming for large programs such as OVERFLOW-D, which consist of over 100,000 lines of FORTRAN and approximately 1000 subroutines. We have employed the automatic parallelization tool CAPO [7] to detect parallelism and insert directives in the code. CAPO is based on the CAPTools [6] toolkit and makes full use of CAPTools unprecedented interprocedural analysis for determining data dependence. CAPO allows a single level of OpenMP for multiple nested loops. The use of directives, such as *!$OMP PARALLEL DO*, are most effective if they can be inserted at the outermost loops, to ensure large granularity and small overhead.

The hybrid code structure at the top is similar to the MPI code (see §3.1), consisting of the time-loop, group-loop and grid-loop. The group-loop runs in parallel via MPI, but the grid-loop, which contains the computationally intensive part of the code, is multi-threaded by OpenMP. Currently, an equal number of threads, N_{thrd}, are spawned per each MPI process. In the hybrid mode, for a total number of, N_{CPU}, processors, MPI spawns, $N_{MPI-proc}$, processors at the initial level of the main program, followed by N_{thrd} OpenMP threads, subject to the following constraint, $N_{CPU} = N_{MPI-proc} * N_{thrd}$. The OpenMP thread initialization follows a "fork/join" program. One of the threads amid N_{thrd} acts as the master thread and the others as team members. The master in each MPI process acts as the MPI processor. In the absence of a parallel construct, the master thread executes in serial mode while the

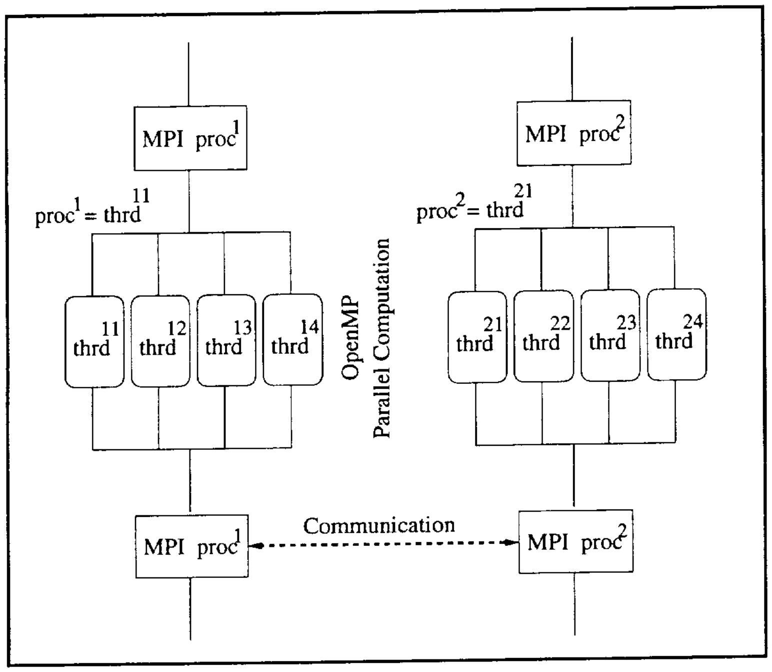

Figure 2: Schematic of the hybrid MPI+OpenMP implementation; master-thread MPI communication and parallel OPenMP computation.

other threads remain idle. The inter-group data exchanges across MPI processors are done only by the masters within a MPI process.

In the hybrid implementation, a sequential communication strategy is adopted. The packing and unpacking of messages are done in a thread-parallel fashion, but the master thread only sends/receives the messages using MPI calls. There is no inter-group cross communication among the threads. This strategy may be a source of sequential bottlenecks depending upon the load of communications. In addition to MPI synchronization calls, OpenMP threads ought to be synchronized at the end of every parallel construct. Fig. 2 illustrates the schematic of the hybrid MPI+OpenMP implementation for two MPI processes and four OpneMP threads in OVERFLOW-D. Mater-threads within each MPI process are exchanging inter-group data.

Load balancing in the hybrid code is static. It is furnished by the grouping strategy with the same characteristics as in the MPI code, §3.1, at start of the first time-step. A dynamic load balancing strategy can be sought, (not implemented in this work), by a varying number of N_{thrd} in proportion to the workload on the MPI processes. The workload can be evaluated by the computation time per group for the first few time-steps at runtime.

The overall activities required to generate the OpenMP constructs in OVERFLOW-D have been a result of a judicial usage of CAPO along with some manual analysis and modification. The hybrid-based code consists of over 1000 OpenMP parallel constructs, whose

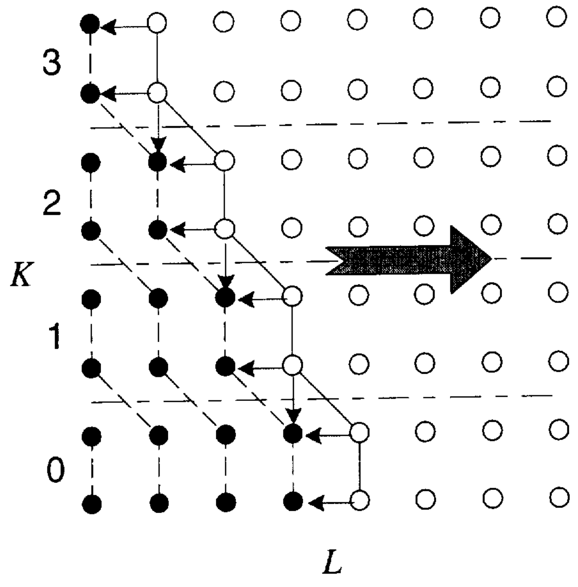

Figure 3: Illustration of a pipeline parallelization method for LU-SGS.

coding is spread out over nearly 400 subroutines. Some parallel constucts within the DCF part of the code had to be modified and/or removed for efficiency and/or debugging. The DCF portion of the code essentially runs in serial mode. Several parallel constructs in subroutines pertinent to the flow solver have been modified; for instance, certain data dependecies were removed in some "paralle do loop" constructs. In addition, parallel constructs in approximately 80 subroutines were found to be interfering with the memory allocation/freeing procedure. This occassionally caused a hung program when run on O2k machines with $N_{thrd} > 2$. All these subroutines were modified leading to stable hybrid execution.

One of the code sections that involved manual modifications is the LU-SGS linear solver. The LU-SGS scheme [17] combines the advantages of LU factorization and Gauss-Siedel relaxation to improve the numerical convergence rate. But the inherited data dependences in the scheme require the availability of the solution on the previous diagonal line for each diagonal line in the solution process. The "hyper-line" algorithm, similar to the "hyper-plane" algorithm [2], was used in the original code to achieve reasonable parallel performance on the vector machine. However, for a cache-based machine like O2K there are two main limitations on the algorithm: poor cache utilization and small communication granularity. In fact, our first version of the OpenMP LU-SGS code generated by CAPO performed very poorly, achieved a speedup of 1.2 on 4 CPUs for a small test case. The poor performance was a direct consequence of the original code structure for which CAPO was only able to insert OpenMP directives to some of the inner loops.

A better approach to parallelize the LU-SGS scheme is the pipeline algorithm described in [22]. To illustrate the pipeline method Fig. 3 shows a case of a 1-D pipeline in which the data grid is partitioned in the K dimension among four threads (or processors). Thread 0 starts from the low-left corner and works on one slice of data for the first L value. Other threads are waiting for data to be available. Once thread 0 finishes its job, thread 1 can start working on its slice for the same L and, in the meantime, thread 0 moves onto the next L. This process continues until all the threads become active. Then they all work conccurrently to the opposite end, as indicated by the large arrow in the figure. The pipeline algorithm has better cache performance and less communication cost than the hyper-plane algorithm. We restructured the LU-SGS code in a way that CAPO could automatically exploit pipelines with OpenMP directives. The new parallel version of LU-SGS not only improved the parallel performance, with a speedup of 2.9 on 4 CPUs for the same test case mentioned earlier, but also the sequential performance.

4 Performance Results

Several experiments have been conducted to demonstrate the functionality and accuracy of the hybrid OVERFLOW-D code, using performance results from two main test cases. The physics of one of the test problems is related to the Navier-Stokes simulation of vortex dynamics in a complex vortical wake flow of hovering rotors, whose solutions were extensively studied in reference [14]. All the performance computations have been obtained on an SGI O2K Shared Memory machine, using 512 MIPS R12K processors with a clock frequency of 400 MHz.

The test cases consist of a medium and of a large overset grid systems as follows:

- CASE 1 grid system consists of 41 grid zones, with a total of approximately 8 million grid points.
- CASE 2 grid system consists of 857 grid zones, with a total of approximately 68 million grid points. To the best of our knowledge, this test case is the largest MPI/OpenMP application currently being evaluated on the O2K platform.

Solutions are obtained over many time-steps to confirm the stabilty of the hybrid code. Residuals of the solutions are compared with the MPI code alone for accuracy. For the performance experiments, however, computations are only carried over 100 time-steps, and the timing mainly reflects flow computations. The DCF computation is only invoked at the initial time-step and its impact over 100 times steps is minimal.

Runtimes (in seconds), denoted by T_{exe}, are normalized per each time iteration step, and reflect the sum of computation and communication, respectively. Furthermore, timings are averaged across the number of $N_{MPI-proc}$ processes used in each run. The "speedup" and "efficiency" reported below are defined as a ratio of "base-runtime" to N_{CPU} runtime, and as a percentage of the actual speedup to the theoretical maximum speedup, respectively. The base-runtime for each test case here is considered to be the timing T_{exe}, for the smallest N_{CPU}.

The number of combinations of $N_{MPI-proc} * N_{thrd}$ that would yield the same N_{CPU} can be limited by the number of groups, which is the same as $N_{MPI-proc}$. This can not be larger than the total number of grid zones in the problem, otherwise some group would be empty. In addition, for reasons of load balancing, it may be necessary to keep the number of groups less than the number grid zones.

4.1 CASE 1

Table 1 shows performance results for the hybrid implementation of OVERFLOW-D on the O2K, using the CASE 1 test problem. Due to the application's memory space requirement, the least number of processors necessary to run the code is $N_{CPU} = 4$. Runtimes are reported for some combinations of $N_{MPI-proc} * N_{thrd}$, speedup and parallel efficiency data are given for the best of these combinations.

Overall speedup increases while the parallel efficiency is decreasing. For $N_{CPU} > 128$ the speedup is insignificant. The changes in the value of runtimes for combinations consisting of the same value for $N_{MPI-proc}$, but varying N_{thrd}, scales reasonably for $N_{thrd} \approx 4$. For instance for $N_{MPI-proc} = 16$, the ratio of runtimes for $N_{thrd} = 1$ to $N_{thrd} = 4$ is 2.5, which is $\approx$ 62% of the ideal value, whereas for $N_{thrd} = 8$, the ratio is $\approx$ 33%. Comparison of runtimes within N_{CPU} rows on the table, consisting of combinations of $N_{MPI-proc} * N_{thrd}$, shows efficiency increases for $N_{MPI-proc} > 8$. Furthermore, the best of hybrid runs with $N_{MPI-proc} > 8$, are in the same ball park as runs with $N_{thrd} = 1$, for the same N_{CPU}.

A detailed analysis can be complex, for there are certain parameters that have to be assessed. One is load balancing among the MPI processes. As discussed earlier, see §3, the current load balancing strategy is static. Load imbalance may stem from both computation and communication. For a run with $N_{MPI-proc}$, an optimum number of $N_{MPI-proc}$ should be sought. Computational load imbalance mainly depends upon the initial grouping strategy. However, in the hybrid mode, computation may further be imbalanced by the anomalies which may exist between CPU assignment and memory placement. The memory should be placed on the node on which the pertinent MPI assigned N_{thrd}, is running.

For combination runs, with $N_{MPI-proc} = 16$ and N_{thrd} =1,2,4, and 8, the relative speed up is nonlinear and reverses from 4 to 8. The nonlinear behavior is also verified based on the computational time (not shown on the table), the cause of which may head us to issues such as, memory placement, cache line optimization, or parallelization strategy. As discussed in §3.2, the main computational task in the hybrid code is performed under OpenMP par-

Table 1: Runtimes (in seconds) of the hybrid implementation on O2K using test problem, CASE 1.

	Hybrid				
N_{CPU}	$N_{MPI-proc}$	N_{thrd}	T_{exe}	Speedup	Efficiency
4	4	1	24.6	1.	100
8	2	4	17.2		
	8	1	14.2	1.73	87
16	2	8	14.6		
	4	4	12.8		
	8	2	10.5		
	16	1	9.6	2.65	64
32	4	8	10.1		
	8	4	6.4		
	16	2	5.9	4.17	52
	32	1	5.9		
48	8	6	5.8		
48	16	3	4.2		
64	8	8	4.0		
	16	4	3.8	5.86	49
	32	2	3.8		
128	8	16	5.8		
128	16	8	3.5		
	32	4	2.6	9.46	30

allelism. A large portion of computational time is consumed by the linear solution scheme, LU-SGS, in this case. The fraction of reduction of computational time, in this case, is not small due to the fact that the LU-SGS solver lends poorly to parallelization.

Table 2 compares runtimes of the hybrid code with $N_{thrd} = 1$ to the corresponding "MPI-alone" code. The MPI-alone refers to computations made by the MPI code with no OpenMP directives being invoked. Due to the overhead associated with the hybrid code, its corresponding runtime is expected to be 5 to 10% larger than for MPI-alone. For the most partthis is true for $N_{CPU} > 8$.

4.2 CASE 2

Table 3 shows performance results of the hybrid code for some combination of MPI processes and OpenMP threads. For the size of the problem in CASE 2, the base-runtime is obtained based on the smallest $N_{CPU} = 56$. Because of this, and since the maximum available processors on the O2K machine are 498 CPUs, the number of combinations of $N_{MPI-proc} * N_{thrd}$

Table 2: Runtimes (in seconds) comparison of the hybrid with $N_{thrd} = 1$ and MPI-alone for CASE 1.

	Hybrid	MPI-alone
N_{CPU}	T_{exe}	T_{exe}
4	24.6	31.4
8	14.2	15.4
16	9.6	9.0
32	5.9	5.3
41	5.9	5.3

for this experiment is limited. Speedup increases while the parallel efficiency is decreasing to 41% at $N_{CPU} = 448$. The hybrid implementation with $N_{thrd} > 1$ outperforms the hybrid with $N_{thrd} = 1$, for the same N_{CPU}. Some issues related to the hybrid runs with $N_{thrd} = 1$ as compared to MPI-alone are discussed below. Combinations with $N_{CPU} = 56$ and varying numbers of N_{thrd} show nonlinear speedup, with almost none between 4 to 8 threads. Again, as stated in CASE 1, this may relate to issues of memory placement, etc.

The number of messages and volume of communication per MPI process is quite large for this test case. The communication times (not shown here) are about 30 to 40% of the total execution times. Threads introduce additional overhead related to creation and synchronization, which increases the overall communication time.

Table 3: Runtimes (in seconds) of the hybrid implementation on the O2K using CASE 2.

	Hybrid				
N_{CPU}	$N_{MPI-proc}$	N_{thrd}	T_{exe}	Speedup	Efficiency
56	56	1	19.1	1.	100
112	56	2	14.6		
	112	1	14.0	1.36	68
224	56	4	9.5		
	112	2	9.2		
	224	1	8.9	2.14	63
336	56	6	9.3		
	112	3	6.6	2.89	48
	336	1	8.1		
448	56	8	9.1		
	112	4	5.8	3.29	41
	224	2	7.1		
	448	1	8.2		

Table 4 compares performance of the hybrid code with $N_{thrd} = 1$ to the with the MPI-alone, using CASE 2. As mentioned in §4.1, inclusion of OpenMP directives would amount to a 5 to 10% increase in the runtime. However, the data on this table shows a significant increase of about 70% in runtimes. The exact cause of this anomally is not known to us at this time, but it may be an indication of some strong interaction between MPI and OpenMP. It appears that the problem is magnified when large numbers and volumes of messages attempt to communicate across the processors. The O2K system utilty software "ssrun" has been used to profile both the hybrid and MPI-alone programs, using $N_{CPU} = 112$. An extensive one-to-one comparison of captured performance data between the two codes shows that timing on all MPI calls is almost doubled for the hybrid implementation. Both the hybrid and MPI-alone codes are identical, except in the compilation of the latter, no OpenMP "-mp" option is invoked.

Table 4: Runtimes (in seconds) comparison of hybrid with $N_{thrd} = 1$ and MPI-alone using CASE 2.

	Hybrid	MPI-alone
N_{CPU}	T_{exe}	T_{exe}
56	19.1	15.6
112	14.0	8.6
224	8.9	6.0
336	8.1	4.7
448	8.2	4.9

5 Conclusions and Future Work

Current parallel implementation on the multi-zonal CFD code, OVERFLOW-D, supports MPI and a dual-level hybrid MPI+OpenMP strategy on shared memory machines and clusters of SMP. Results were only tested on shared memory SGI O2K machines. The hybrid approach delivers about the same performance as the MPI-alone, for the same total number of processors. The hybrid implementation, however, can outperform the MPI-alone, when the MPI-base code can not run beyond a certain number of groups, and is limited by the number of grid zones and related load balances. This conclusion is verified for CASE 1, above.

With respect to CASE 2, more experiment is needed to understand the strong interaction which occurs between OpenMP and MPI. This interaction affects the overall timing on all MPI calls and is particularly noticable when results of the hybrid code, running with one thread, are compared with results of the MPI-alone, for the same number of MPI processes. These results may be dependent upon the hardware and vendor implementation of the parallel libraries.

Future work should focus on a more detailed analysis of the hybrid code scaling performance. Linear solution algorithms, other than LU-SGS, should be efficiently parallelized and tested. Certain OpenMP parallel constructs in the hybrid code need improvement that could not be visited under the current time constraints. On the SGI O2K platform, an attempt should further be made to investigate the effect of the placement of processes and memory, as discussed in §4.1. The cause of OpenMP interference with MPI on the O2K machine, discussed in §4, should also be verified. Cache optimization of fine level loops may significantly enhance the scaling performance of the hybrid code. It is essential that a dynamic load balancing technique be implemented into the hybrid code. A detailed analysis is required to understand the impact of various parameters on the parallel performance. Further future work should include porting and testing of the hybrid approach on other platforms, such as IBM SP.

References

[1] http://captools.gre.ac.uk.

[2] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga. "Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors". Technical Report NAS RNR-93-007, NASA Ames Research center, Mountain View, CA, 1993.

[3] M. J. Djomehri and Y. Rizk. "Performance and Application of Parallel OVERFLOW Codes on Distributed and Shared Memory Platforms". In *Proc. NASA HPCCP/CAS Workshop.* NASA Ames Research Center, August 1998.

[4] M. J. Djomehri and Y. Rizk. "Performance Assessment of OVERFLOW on Distributed Computing Environment". In *Proc. NASA HPCCP/CAS Workshop.* NASA Ames Research Center, February 2000.

[5] http://www.crpc.rice.edu/CRPC/softlib/TRs_online.html, January 1997. "High Performance Fortran Forum, High Performance Fortran Language Specification, CRPCTR92225".

[6] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. Leggett. "Computer Aided Parallelization Tools (CAPTools) Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes". *Parallel Computing,* (22):163–195, 1996. See also [1] for more information.

[7] H. Jin, M. Frumkin, and J. Yan. "Code Parallelization with CAPO –– A User Manual". Technical Report NAS-01-008, NASA Ames Research Center, 2001.

[8] D. J. Mavriplis. "Parallel Performance Investigations of an Unstructured Mesh Navier-Stokes Solver". Technical Report NASA/CR-2000-210088, ICASE No.2000-13, ICASE, Hampton, Virginia, 2000.

[9] R. Meakin. "A New Method for Establishing Inter-Grid Communication Among Systems of Overset Grids". In *Proc. 10th AIAA Computational Fluid Dynamics Conf.*, pages 662–676, June 1991. Paper 91-1586.

[10] R. Meakin. "On Adaptive Refinement and Overset Structured Grids". In *Proc. 13th AIAA Computational Fluid Dynamics Conf.*, 1997. Paper 97-1858.

[11] http://www.openmp.org. "OpenMP Fortran Application Program Interface".

[12] P. G. Buning and W. Chan and K. J. Renze and D.Sondak and I. T. Chiu and J. P. Slotnick and R. Gomez and D. Jespersen. *"Overflow User's Manual".* NASA Ames Research Center, Mountain View, CA, version 1.6au edition, 1995.

[13] R. C. Strawn and J. U. Ahmad. "Computational Modeling of Hovering Rotors and Wakes". In *Proc. 38th AIAA Aerospace Sciences Meeting & Exhibit,* January 2000. paper 2000-0110.

[14] R. C. Strawn and M. J. Djomehri. "Computational Modeling of Hovering Rotor and Wake Aerodynamics". In *Proc. American Helicopter Society, 57th Annual Forum and Technology Display*, Washington, D.C, May 2001.

[15] R. D. Loft and S. J. Thomas and J. M. Dennis. "Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models". In *Proceedings of SuperComputing*, Denver, Nov. 2001.

[16] R. Meakin and A. M. Wissink. "Unsteady Aerodynamic Simulation of Static and Moving Bodies Using Scalable Computers". In *Proc. 14th AIAA Computational Fluid Dynamics Conf.*, 1999. Paper 99-3302.

[17] S. Yoon and A. Jameson. "An LU-SSOR Scheme for the Euler and Navier-Stokes Equations". In *Proc. 25th AIAA Aerospace Sciences Meeting & Exhibit*, January 1987. Paper 87-0600.

[18] J. Steger, F. Dougherty, and J. Benek. "A Chimera Grid Scheme". In *ASME FED*, number 5, 1983.

[19] J. R. Taft. "Performance of the OVERFLOW-MLP CFD Code on the NASA Ames 512 CPU Origin System". In *NASA HPCCP/CAS Workshop*. NASA Ames Research Center, February 2000.

[20] W. Gropp and E. Lusk and A. Skjellum. *"Using MPI:Portable Parallel Processing with the Message-Passing Interface"*. The MIT Press, Cambridge, MASS, 1994.

[21] A. M. Wissink and R. Meakin. "Computational Fluid Dynamics with Adaptive Overset Grids on Parallel and Distributed Computer Platforms". In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1628–1634, Las Vegas, NV, 1998.

[22] M. Yarrow and R. Van der Wijngaart. "Communication Improvement for the NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes". Technical Report NAS RNR-97-032, NASA Ames Research center, Mountain View, CA, 1997.