Performance Analysis of a Hybrid Overset Multi-Block Application on Multiple Architectures

M. Jahed Djomehri* and Rupak Biswas†
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035
{djomehri,rbiswas}@nas.nasa.gov

Abstract

This paper presents a detailed performance analysis of a multi-block overset grid computational fluid dynamics application on multiple state-of-the-art computer architectures. The application is implemented using a hybrid MPI+OpenMP programming paradigm that exploits both coarse and fine-grain parallelism; the former via MPI message passing and the latter via OpenMP directives. The hybrid model also extends the applicability of multi-block programs to large clusters of SMP nodes by overcoming the restriction that the number of processors be less than the number of grid blocks. A key kernel of the application, namely the LU-SGS linear solver, had to be modified to enhance the performance of the hybrid approach on the target machines. Investigations were conducted on cacheless Cray SX6 vector processors, cache-based IBM Power3 and Power4 architectures, and single system image SGI Origin3000 platforms. Overall results for complex vortex dynamics simulations demonstrate that the SX6 achieves the highest performance and outperforms the RISC-based architectures; however, the best scaling performance was achieved on the Power3.

Keywords: Parallel performance, multi-level paradigm, cache-based architectures, vector machines, computational fluid dynamics

1 Introduction

A large fraction of high performance computing (HPC) platforms today use cache-based microprocessors, which are assembled as systems of symmetric multi-processing (SMP) nodes. Additionally, in order for many important scientific "legacy codes" originally developed for vector machines to perform efficiently on cache-based architectures, some significant changes in their algorithmic structures must be made. Recent development of vector-based SMP systems [1] offers a new HPC alternative for many of these legacy codes. To evaluate and compare the performance of state-of-the-art cache- and vector-based architectures in conjunction with practical scientific problems, we have selected a high-fidelity NASA production Navier-Stokes CFD application, called OVERFLOW-D [3, 7], that is based on multi-block overset grid methodology.

*Computer Sciences Corporation
†NASA Advanced Supercomputing Division

The overset grid approach falls into the general class of Schwartz decomposition methods [9]. The solution process resolves the geometrical complexity of the problem domain by generating and using overlapping multi-block structured discretization grids. This approach typically employs a Chimera interpolation technique [10] to periodically update and exchange inter-grid boundary information. A brief overview of the OVERFLOW-D methodology is given in Section 2.

This paper describes our performance analysis of a hybrid MPI+OpenMP programming paradigm implementation of OVERFLOW-D, and tested on multiple computer architectures. The approach consists of two levels of parallelism: the first is coarse-grained based on MPI message passing while the second is fine-grained based on OpenMP directives [6]. One major advantage of the combined paradigms is that it extends the applicability of multi-block applications to large clusters of SMP nodes. Details of the hybrid model as applied to OVERFLOW-D is presented in Section 3.

Our hybrid approach is conceptually similar to the "Shared Memory Multi-Level Parallelism" (MLP) [13] model that was initially developed at NASA Ames Research Center. However, the MLP method uses a fundamentally different strategy for data exchange among processors. It exploits the underlying shared memory for all data communication via direct memory referencing instructions and is more efficient than message passing; but, its applicability is limited to pure shared memory machines.

Furthermore, we describe the modifications that were made to a key numerical algorithm of the application, namely the LU-SGS linear solver, in order to enhance the parallel performance of the hybrid implementation. These modifications are reported in Section 4.

All performance evaluation experiments were conducted on four parallel machines: the Cray SX6 vector system, the cache-based IBM Power3 and Power4 machines, and the shared memory SGI Origin3000 (O3K) platform. A brief description of these architectures and the compiler flags used are given in Section 5. Performance results obtained using complex vortex dynamics simulations of a practical problem of interest are presented in Section6. Overall results demonstrate that the SX6 outperforms the RISC-based architectures; however, the Power3 demonstrated the best scalability and the O3K achieved the highest sustained floating-point performance (relative to peak).

2 Overset Methodology

In this section, we provide a brief overview of the high-fidelity multi-block overset grid application for Navier-Stokes simulations, called OVERFLOW-D [7].

2.1 Flow Solver

The overset grid application is popular within the aerodynamics community due to its ability to handle complex designs with multiple geometric components, whereby individual body-fitting grids are easily constructed about each component. OVERFLOW-D is explicitly designed to simplify the modeling of problems when components are in relative motion. At each time step, the flowfield equations are solved independently on each grid (also known as blocks or zones) in a sequential

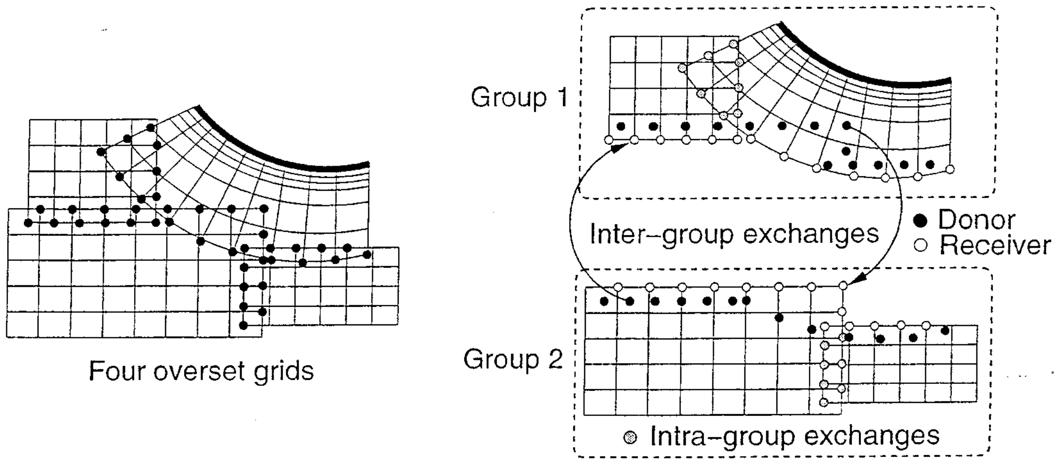


Figure 1: Overset grid intra-group and inter-group communication.

manner. Overlapping boundary points or inter-grid data is updated from previous solutions prior to the start of the current time step using a Chimera interpolation procedure [10]. The code uses finite differences in space, with a variety of spatial differencing and implicit/explicit temporal time-stepping. A domain connectivity program is used to determine the inter-grid boundary data.

The main computational logic at the top level of the sequential code consists of a "time-loop", a "grid-loop", and a "subiteration-loop". The last two loops are nested within the time-loop. Within the grid-loop, solutions are obtained on the individual grids with imposed boundary conditions, where the Chimera interpolation procedure successively updates inter-grid boundaries after computing the numerical solution on each grid. Convergence of the solution process is accelerated by the subiteration-loop. Upon completion of the grid-loop, the solution is automatically advanced to the next time step by the time-loop. The overall procedure may be thought of as a Gauss-Seidel iteration.

2.2 Grid Connectivity Interpolation

The Chimera interpolation procedure [10] determines the proper connectivity of the individual grids. Adjacent grids are expected to have at least a one-cell (single fringe) overlap to ensure the continuity of the solutions; for higher-order accuracy and to retain certain physical features in the solution, a double fringe overlap is sometimes used [11]. A program named Domain Connectivity Function (DCF) [8] computes the inter-grid donor points that have to be supplied to other grids (see Fig. 1). The DCF procedure is incorporated into the OVERFLOW-D code and fully coupled with the flow solver. All boundary exchanges are conducted at the beginning of every time step based on the interpolatory updates from the previous time step. In addition, for dynamic grid systems, DCF has to be invoked at every time step to create new holes and inter-grid boundary data.

3 Hybrid Programming Model

In the following subsections, we briefly describe a hybrid two-level parallel design [6], which implements a combined MPI+OpenMP model, into OVERFLOW-D. The combined implementation permits the execution of the application either in pure message passing mode or, as in the true hybrid model, with multiple threads per MPI process (or task).

3.1 Message Passing Parallelization

The first level of parallelization is based on the MPI message passing library using the single program multiple data (SPMD) paradigm. The MPI model has been developed around the multi-block feature of the sequential code, which offers a natural coarse-grain parallelism [14]. To facilitate parallel execution, a grouping strategy is required to assign each grid to an MPI process, and concurrently distribute the workload across the processors in a load-balanced fashion. The total number of groups, G, is equal to the number of MPI processes, M. Since a grid can only belong in one group, the total number of grids, Z, must be at least equal to M. If Z is larger than M, a group will consist of more than one grid. There are various, simple to sophisticated, grouping strategies [4] available for overset grid applications. In this paper, the bin-packing approach was used and is reviewed in Section 3.1.1 for the sake of completeness. The assignment of groups to processors is somewhat random, and is taken care of by the operating system, usually based on a first-touch strategy at the time of the run.

The logic in the MPI model differs slightly from that of the sequential case ($G = P = 1$) mentioned in 2.1. Here the grid-loop is subdivided into two procedures, a loop over groups ("group-loop") and a loop over the grids within each group. Since each MPI process is assigned to only one group, the group-loop is performed in parallel, with each group performing its own sequential grid-loop. The inter-grid boundary updates among the grids within each group (called intra-group updates) are performed as in the serial case. Chimera updates are also necessary for overlapping grids that are in different groups, and are known as inter-group exchanges (see Fig. 1). The inter-group donor points from grids in group G_i to grids in group G_j are stored in a send buffer and exchanged between the corresponding processes via MPI calls. These inter-group exchanges are transmitted at the beginning of every time step based on the interpolatory updates from the previous time step. The message passing is done by an efficient asynchronous communication model, discussed in Section 3.1.2, based on the MPI library.

3.1.1 Grouping Algorithm

The original parallel version of OVERFLOW-D uses a grid grouping strategy based on a bin-packing algorithm [14]. It is one of the simplest clustering techniques that strives to maintain a uniform number of "weighted" grid points per group while retaining some degree of connectivity among the grids within each group. Prior to the grouping procedure, each grid is weighted depending on the physics of the solution sought. The goal is to ensure that each weighted grid point requires the same amount of computational work. For instance, the execution time per point belonging to near-body grids requiring viscous solutions is higher than that for the inviscid solutions of off-body grids. The weight can also be deduced from the presence or absence of a turbulence model. The bin-packing algorithm then sorts the grids by size in descending order, and assigns a grid to every empty group. Therefore, at this point, the G largest grids are each in a group by themselves. The remaining $Z - G$ grids are then handled one at a time: each is assigned to the smallest group that satisfies the connectivity test with other grids in that group. The connectivity test only inspects for an overlap between a pair of grids, regardless of the size of the boundary data or their connectivity to other neighboring grids. The process terminates when all grids are assigned to groups.

3.1.2 Asynchronous Communication

Inter-processor communication can be synchronous or asynchronous, but the choice significantly affects the MPI programming model. The current version of OVERFLOW-D uses asynchronous message passing that relaxes the communication schedule in order to hide latency [5]. Asynchronous communication consists of non-blocking MPI send/receive calls. These pairs of non-blocking invocations place no constraints on each other in terms of completion. Receive completes immediately, even if no messages are available, and hence allows maximal concurrency. In general, however, control flow and debugging can become a serious problem if, for instance, the order of messages needs to be preserved. Fortunately, in the overset grid application, the Chimera boundary updates take place at the completion of each time step, and the computations are independent of the order in which messages are sent or received. Being able to exploit this fact allows us to easily use asynchronous communication within OVERFLOW-D.

3.2 OpenMP Implementation

The second level of parallelism in the hybrid approach is based on the OpenMP programming model, where explicit compiler directives are inserted into the code at the loop level. The logic is the same as in the pure MPI case, only the computationally intensive portion of the code (i.e. the grid-loop) is multi-threaded via OpenMP. In our current implementation, an equal number of OpenMP threads are spawned for each MPI task. The total number of processors used is the product of the number of MPI tasks and OpenMP threads.

The OpenMP thread initialization follows a fork/join procedure. Whenever a parallel region is encountered, one of threads acts as the master while the others behave as team members; otherwise the master executes alone while the others remain idle. Message passing is performed by the master thread only; in other words, there is no inter-group cross communication among the threads. Fig. 2 illustrates the schematic of the hybrid implementation for two MPI processes and four OpenMP threads. Master threads within each MPI task exchange inter-group boundary data in OVERFLOW-D.

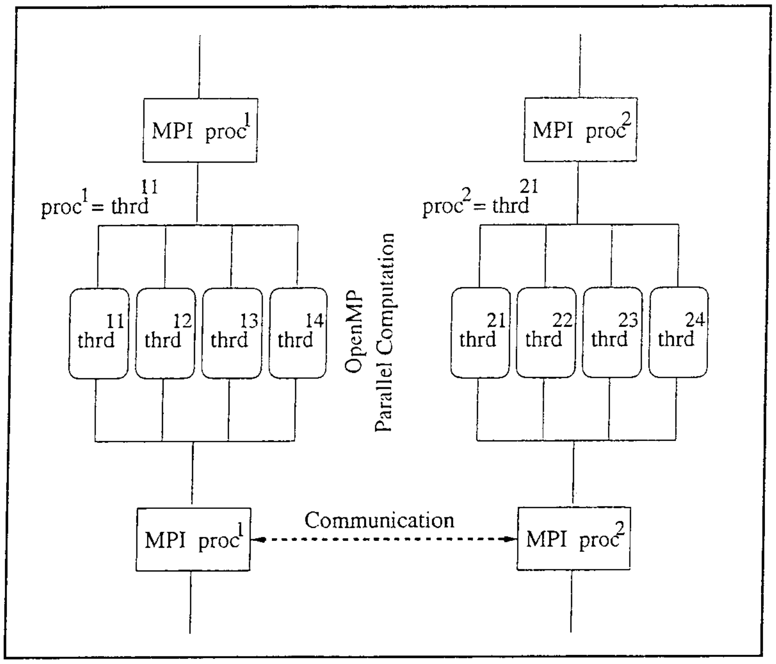


Figure 2: Schematic of the hybrid MPI+OpenMP implementation, master-thread MPI communication, and parallel OpenMP computation.

4 LU-SGS Reorganization

Both the pure MPI and hybrid programming models discussed above were developed based on the sequential (serial) version of OVERFLOW-D, the organization of which was designed to exploit vector machines, such as the Cray-YMP and C90. The same basic code structure is used on all machines, except for the LU-SGS [16], linear solver that required significant modifications to enhance efficiency. The LU-SGS solver combines the advantages of LU factorization and Gauss-Siedel relaxation to improve the numerical convergence rate. Unfortunately, the inherited data dependencies 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 cache-based machines, there are two main deficiencies of the algorithm: poor cache utilization and small communication granularity. In fact, a naive version of the OpenMP LU-SGS code performed very poorly on an O3K, achieving a speedup of only 1.2 on four CPUs for a small test case. The poor performance was a direct consequence of the original code structure which suggested the insertion of OpenMP directives into some of the inner loops.

A smart approach to parallelize the LU-SGS scheme is based on the pipeline algorithm described in [15]. Fig. 3 illustrates the pipeline method for 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 lower-left corner and works on one slice of the data for the first L value. Other threads wait for the data to become available. Once thread 0 finishes its job, thread 1 can start working on its slice for the same L

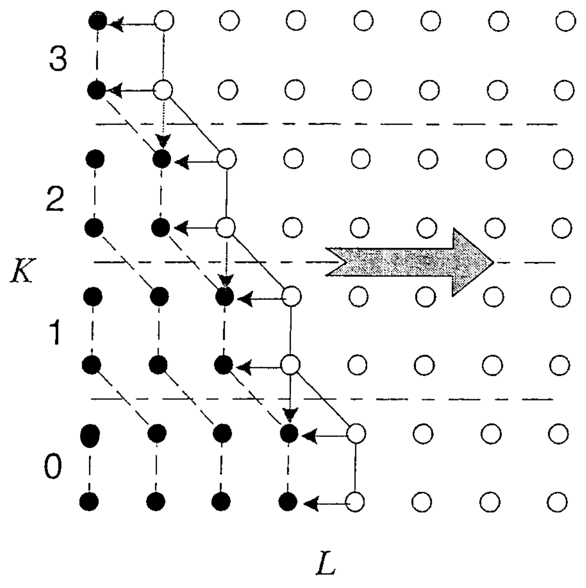


Figure 3: Illustration of a pipeline parallelization method for LU-SGS.

and, in the meantime, thread 0 moves onto the next L. This process continues until all the threads become active. Then they all work concurrently 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. The new parallel version improved the hybrid performance, with a speedup of 2.9 on 4 CPUs for the same test case mentioned earlier.

On the Power3/4 and O3K machines, this pipeline algorithm was implemented, while a vector strategy was executed on the SX6. Except for a few minor changes in several subroutines in an effort to meet the specific MPI/OpenMP compiler requirements on each machine, the LU-SGS program has been the only module to be reorganized to enhance efficiency.

5 Target Architectures

All experiments were performed on four state-of-the-art parallel machines: the Cray SX6 system at Arctic Region Supercomputing Center (ARSC), the IBM Power3 at Lawrence Berkeley National Laboratory (LBNL), the IBM Power4 temporarily installed at NASA Ames Research Center (ARC), and the SGI Origin3000 (O3K) also at ARC. We give a brief overview of each platform as well as the compiler flags that were used to run OVERFLOW-D.

5.1 System Specifications

The cacheless SX6 uses vectorization to exploit regularities in the computational structure, thereby expediting uniform operations on independent data sets. Memory latencies are masked by overlapping pipelined vector operations with memory fetches. The system at ARSC is a single SMP node consisting of eight 500 MHz processors, each with a peak performance of 8 Gflops/s. The processors contain 72 vector registers, each holding 256 64-bit words. For non-vectorizable instructions, the SX6 contains a scalar processor with a 64KB instruction cache, a 64KB data cache, and 128 general-purpose registers. Since the SX6 vector unit is significantly more powerful than the scalar processor, it is critical to achieve high vector operation ratios, either via compiler discovery or explicitly through code (re-)organization.

The Power3 system at LBNL, part of IBM's RS/6000 series, has 380 SMP compute nodes. Each 375 MHz processor contains two floating-point units (FPUs) that can issue a multiply-add (MADD) per cycle for a peak performance of 1.5 GFlops/s. The out-of-order architecture uses prefetching to reduce pipeline stalls due to cache misses. The CPU has a 32KB instruction cache and a 128KB 128-way set associative L1 data cache, as well as an 8MB four-way set associative L2 cache with its own private bus. Each SMP node consists of 16 processors connected to main memory via a crossbar. Multi-node configurations are networked via the IBM Colony switch using an omega-type topology.

The IBM Power4 pSeries 690 is the latest generation of IBM's RS/6000 series. The temporary system at ARC was composed of two 32-way SMP nodes, coupled together via the Colony switch. Each 32-way SMP consists of 16 Power4 chips (organized as 4 MCMs), where a chip contains two 1.3 GHz processor cores. Each core has two FPUs capable of a fused MADD per cycle, for a peak performance of 5.2 Gflops/s. Each processor contains its own private L1 cache (64KB instruction and 32KB data) with prefetch hardware; however, both cores share a 1.5MB unified L2 cache. The directory for the L3 cache is located on-chip, but the memory itself resides off-chip. The L3 is designed as a stand-alone 32MB cache, or to be combined with other L3s on the same MCM to create a larger interleaved cache of up to 128MB. All our Power4 experiments reported in this paper were obtained within one compute node.

The SGI O3K is a scalable, hardware-supported cache-coherent nonuniform memory access (CC-NUMA) system, with an aggressive communication architecture. The hardware makes all memory equally accessible from a software perspective by sending memory requests through routers located on the nodes. Memory access time is nonuniform, depending on how far away the word lies from the processor. The interconnection network is a hypercube, bounding the maximum number of memory hops to a logarithmic function of the number of processors. Results presented in this paper were obtained on the 512-processor system at ARC. Each O3K node is an SMP containing four 400 MHz MIPS R12000 IP35 processors and 512 MB of local memory. Each processor, with a peak performance of 0.8 GFlops/s, also has separate 32 KB L1 instruction and data caches, and a 2-way set-associative 8 MB L2 cache where only it can fetch and store data.

5.2 Compiler Flags

The following compiler options were used in conjunction with the SX6 f90 and C compiler in building 64-bit executables for MPI and hybrid applications. The `-P openmp` option was turned off in building the the pure MPI executable.

- `FFLAGS = -C vsafe -f0 -P openmp -size_t64`

- `CFLAGS = -h size_t64`
- `LDFLAGS = -Wl,"-h size_t64"`

The following compiler options were used on IBM Power3/4 systems. Here, the 64-bit executables were built for the MPI and hybrid applications, using Fortran and C compiler scripts, mpxlf_r and xlc_r, respectively.

- `FFLAGS = -O3 -g -q64 -qsmp=omp -qfixed -qnosave`
- `CFLAGS = -O -g -q64`
- `LDFLAGS = mpxlf_r -q64 -qsmp`

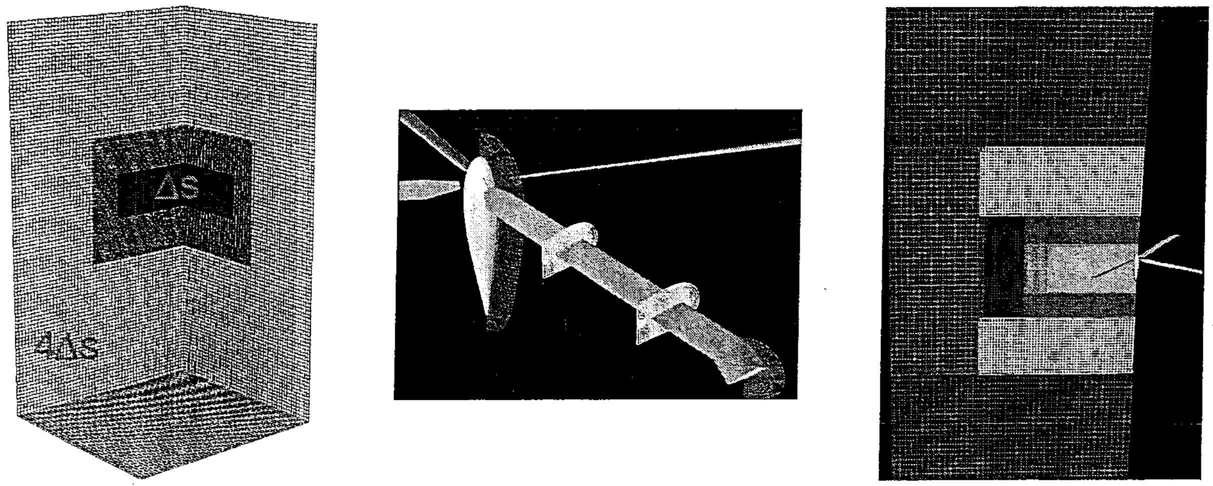
The following compiler options were used on the SGI O3K system to build the 64-bit executables for MPI and hybrid applications using f90 and C compilers, respectively. The OpenMP option `-mp` was turned off for the MPI application.

- `FFLAGS = -O3 -64 -mips4 -r10000 -mp`
- `CFLAGS = -O3 -mips4 -r10000 -64`
- `LDFLAGS = -O3 -64 -mips4 -mp`

6 Performance Results

The CFD problem used for the experiments in this paper is a Navier-Stokes simulation of vortex dynamics in the complex wake flow region for hovering rotors. Figure 4 shows sectional views of the test application grid system. The Cartesian off-body wake grids surround the curvilinear near-body grids with uniform resolution, but become gradually coarser upon approaching the outer boundary of the computational domain. Specifically, the spacing of the off-body grid nearest the rotor blade is Δs, that for the next surrounding level is $2\Delta s$, and so on for every successive level. Figure 5 shows a cut plane through the computed vortex wake system including vortex sheets as well as a number of individual tip vortices. A complete description of the underlying physics and an extensive analysis of the numerical simulations pertinent to this test problem can be found in [12]. Our overset grid system test case consisted of 41 blocks and approximately 8 million grid points.

Tables 1, 2, and 3 show total execution timings, T_{exec}, on the Cray SX6, IBM Power3 and Power4, and SGI O3K systems, respectively. T_{exec} is the time required to solve every iteration of the application (averaged over 20 iterations), and includes the computation, communication, Chimera interpolation, and processor idle times. T_{exec} is reported for both the MPI and hybrid paradigms to demonstrate the impact of the second level of parallelism introduced by OpenMP. The hybrid runs with M MPI tasks and one OpenMP thread are conceptually equivalent to pure MPI runs with M tasks; however, due to procedural differences, the timings may be somewhat different. A dash (—) entry in these tables indicates that data was either, "not available" or "not applicable".



(a) (b) (c)

Figure 4: Sectional views of the test application grid system: (a) off-body Cartesian wake grids, (b) near-body curvilinear grids, and (c) cut plane through the off-body wake grids surrounding the hub and rotors.

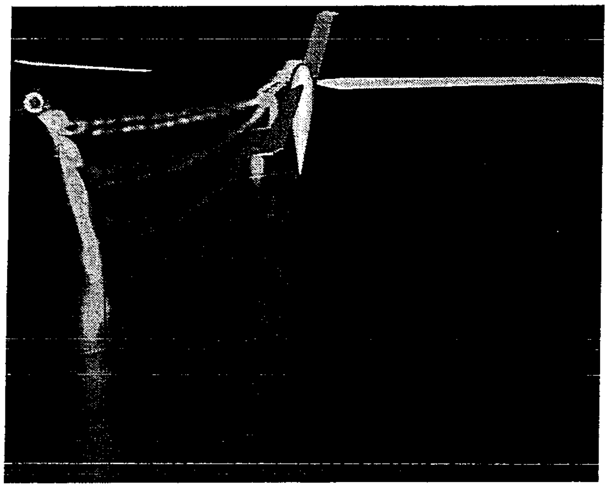


Figure 5: Computed vorticity magnitude contours on a cutting plane located 45° behind the rotor blade.

Performance results on the SX6 are presented in Table 1. The MPI and hybrid paradigms are appended with a –V or –NV to indicate whether or not the code was vectorized, with respect to the LU-SGS linear solver. The table includes data regarding floating point operations per second (Mflops/s), the average vector length (AVL), and vector operation ratio (VOR). AVL and VOR were measured using the SX6/f90 compiler option, `-ftrace`. Observe that T_{exec} for runs with the vectorized version of LU-SGS are smaller than the non-vectorized ones by at least a factor of 3X, signifying the performance improvement gained by vectorizing the solver. The relatively small AVL

Table 1: Performance results on the Cray SX-6 system

P	*MPI Tasks*	*OpenMP Threads*	*Paradigm*	T_{exec} *(sec)*	*Mflops/s*	*AVL*	*VOR (%)*
2	2	—	MPI-NV	16.4	760	83	77
2	2	—	MPI-V	5.5	2265	87	80
2	2	1	Hybrid-NV	16.7	746	83	77
2	2	1	Hybrid-V	5.6	2492	84	77
4	4	—	MPI-NV	9.1	1369	74	68
4	4	—	MPI-V	2.8	4450	84	76
4	4	1	Hybrid-NV	9.1	1369	74	68
4	4	1	Hybrid-V	2.8	4450	83	71
4	2	2	Hybrid-V	3.6	3461	80	71
6	6	—	MPI-NV	5.7	2185	75	67
6	6	—	MPI-V	2.0	6230	81	73
6	6	1	Hybrid-NV	5.9	2111	75	67
6	6	1	Hybrid-V	2.1	5934	79	68
6	2	3	Hybrid-V	3.0	4153	77	66
8	8	—	MPI-NV	5.9	2111	75	61
8	8	—	MPI-V	1.6	7787	79	69
8	8	1	Hybrid-NV	6.1	2042	75	60
8	8	1	Hybrid-V	1.6	7787	76	69
8	2	4	Hybrid-V	2.5	4984	77	67
8	4	2	Hybrid-V	1.8	6922	79	68

and limited VOR explain why the code achieves a maximum of only 7.8 Gflops/s on 8 processors (12% of peak). Reorganizing OVERFLOW-D would achieve higher vector performance; however, extensive effort would be required to modify this production code.

Except for $P = 8$, the hybrid paradigm slightly underperforms MPI due to the overhead associated with OpenMP thread management. For a given total number of processors, runs with larger numbers of OpenMP threads appear to be less efficient than those with fewer threads. This is also due to OpenMP overheads. However, the primary advantage of using the hybrid paradigm for overset grid applications is that it allows execution on larger processor counts. The performance scalability for both paradigms is almost identical but is expected to suffer for large numbers of MPI tasks due to workload imbalance.

Timing results and Mflops/s on the IBM Power3 and Power4 systems are shown in Table 2; in addition, the L1 cache hit rate and TLB misses per cycle are listed for the Power3. These data could not be obtained on the Power4 due to its short temporary duration at ARC. As expected, the Power4 outperforms the Power3 over the entire range of processors, from two to 32. Note that for $P = 32$, the Power3 runs were split across two SMP nodes communicating via Colony switches; whereas all runs on the Power4 were executed on one SMP node enjoying fast intra-cabinet interconnects. Nevertheless, the Power3 results are impressive. For small numbers of processors

Table 2: Performance results on the IBM Power3 and Power4 systems

	MPI	*OpenMP*		*Power3*				*Power4*	
P	*Tasks*	*Threads*	*Paradigm*	T_{exec} *(sec)*	*Mflops/s*	L_1 *(%)*	*TLB (%)*	T_{exec} *(sec)*	*Mflops/s*
2	2	—	MPI	46.7	266	93.3	0.245	15.8	788
2	2	1	Hybrid	28.9	431	98.1	0.123	18.2	684
4	4	—	MPI	26.6	468	95.4	0.233	8.5	1465
4	4	1	Hybrid	14.9	838	98.5	0.215	10.1	1233
4	2	2	Hybrid	15.2	819	98.1	0.123	10.5	1186
8	8	—	MPI	13.2	943	96.6	0.187	4.3	2897
8	8	1	Hybrid	7.4	1683	99.0	0.101	6.0	2076
8	2	4	Hybrid	9.2	1354	98.2	0.112	5.9	2111
8	4	2	Hybrid	8.0	1557	98.9	0.188	6.2	2009
16	16	—	MPI	8.0	1557	98.2	0.143	3.7	3367
16	16	1	Hybrid	4.6	2708	99.5	0.054	4.5	2768
16	8	2	Hybrid	4.1	3039	99.2	0.084	3.9	3194
16	4	4	Hybrid	4.8	2595	99.0	0.176	3.7	3367
16	2	8	Hybrid	7.6	1639	99.0	0.080	4.0	3115
32	32	—	MPI	4.5 (2 nodes)	2768	98.7	0.108	3.4 (1 node)	3664
32	32	1	Hybrid	4.7 (2 nodes)	2651	99.7	0.044	2.8 (1 node)	4450
32	16	2	Hybrid	2.4 (2 nodes)	5191	99.5	0.039	3.6 (1 node)	3461
32	8	4	Hybrid	2.6 (2 nodes)	4792	99.2	0.071	2.7 (1 node)	4614
32	4	8	Hybrid	3.8 (2 nodes)	3461	99.3	0.100	2.8 (1 node)	4450
32	2	16	Hybrid	14.1 (2 nodes)	883	99.3	0.046	3.4 (1 node)	3664

($P = 2$ and $P = 4$), the Power4 timings are significantly better than those for the Power3; this is due to the Power4's faster clock and complex but effective data locality system implemented via the architectural association of the L1, L2, and L3 caches.

For $P = 8$, both systems achieve about 7% of peak performance for the pure MPI runs; however, the hybrid paradigm on the Power3 runs at more than 14% of peak. For $P = 32$, the Power3 remains scalable (achieving 11% of peak), whereas the Power4 performance deteriorates significantly. This is probably because of the complex architecture of the Power4 which we were unable to fully exploit. Timing comparisons between the pure MPI and hybrid paradigms on the Power3 show that the latter outperforms the former, and for some cases, by a factor of nearly 2X. On the Power4, the same comparison shows that MPI performs better than the hybrid strategy for $P < 16$; however, the reverse is true for $P = 32$. As on the SX6, runs with larger numbers of OpenMP threads beyond an optimal value is less efficient, for a fixed value of P.

The L1 hit rate and TLB misses on the Power3, although reasonable for small P, improve significantly with the number of processors used. For example, TLB misses for $P = 32$ shows an improvement of 4X relative to $P = 4$. On the Power3, the timing for $P = 32$ with two MPI tasks and 16 OpenMP threads is extremely poor; the reasons being a lack of data locality and that the job is split across two SMP nodes. However, the most important cause for poor performance

Table 3: Performance results on the SGI O3K system

P	MPI $Tasks$	$OpenMP$ $Threads$	$Paradigm$	T_{exec} (sec)	$Mflops/s$
2	2	—	MPI	39.7	313
2	2	1	Hybrid	39.9	312
4	4	—	MPI	21.8	571
4	4	1	Hybrid	22.0	566
4	2	2	Hybrid	21.5	579
8	8	—	MPI	11.2	1112
8	8	1	Hybrid	11.3	1102
8	2	4	Hybrid	14.1	883
8	4	2	Hybrid	13.7	909
16	16	—	MPI	6.1	2042
16	16	1	Hybrid	6.3	1977
16	8	2	Hybrid	7.6	1639
16	4	4	Hybrid	12.2	1021
16	4	4	Hybrid-PIN	6.6	1887
16	2	8	Hybrid	12.7	981
16	2	8	Hybrid-PIN	8.3	1501
32	32	—	MPI	3.8	3278
32	32	1	Hybrid	3.8	3278
32	8	4	Hybrid	7.9	1577
32	8	4	Hybrid-PIN	3.4	3664
32	4	8	Hybrid	12.6	988
32	4	8	Hybrid-PIN	4.7	2651
32	2	16	Hybrid	16.5	755
32	2	16	Hybrid-PIN	8.8	1416

is the overhead associated with the OpenMP thread management procedures which negate the computational benefits.

Timing results and floating point operations per second on the SGI O3K are shown in Table 3. Results for the hybrid paradigm are presented in two fashions for certain runs: one is the basic hybrid strategy and the other is appended with a –PIN. Hybrid–PIN uses a special SGI O3K O/S system call, called "pin-to-node". Under current IRIX scheduling, the placement of the MPI processes (tasks) and spawned OpenMP threads (for mixed MPI+OpenMP jobs) may have various possible permutations over the selected compute nodes. An optimal placement of threads and processes may be furnished via the pin-to-node function, which consists of low-level IRIX calls. In other words, pin-to-node prevents dynamic thread migration during the entire course of the computation. The pin-to-node procedure is part of the MLP library, and has been frequently used in the context of the single system image shared-memory programming model [13].

We have implemented the pin-to-node procedure in conjunction with our hybrid approach for enhanced performance. As seen in Table 3, the timing results for the hybrid-PIN paradigm exceed that for our standard hybrid method. The improvement factor varies with the number of threads. For $P = 32$, with four MPI tasks and eight OpenMP threads, hybrid-PIN outperforms hybrid by a factor of almost 3X. Surprisingly, when using two tasks and 16 threads, the performance of the O3K is extremely poor. Similar observations could also be made from Table 2. There are at least two possible reasons: lack of data locality and the overhead associated with OpenMP procedures (such as fork/join and synchronization). A maximum of 3.6 Gflops/s is achieved on 32 processors (14% of peak performance). As on the SX6 and Power4 machines, pure MPI results are slightly better than those with hybrid-PIN. Also, increasing the number of OpenMP threads does not help.

In terms of absolute timings (T_{exec}), the SX6 (when running the vectorized solver) outperforms the other three architectures. Results show that the best run time for 8 processors on the SX6 (1.6 secs) is more than 40% less than the best 32-processor Power4 number (2.7 secs). Scalability on the Power3 exceeds all others; the O3K ranks second for our test application. The O3K demonstrated the highest sustained performance (14% of peak on 32 processors).

The hybrid programming paradigm is the most complex as it combines two layers of coarse- and fine-grain parallelism. In general, it therefore requires more programmer effort; however, our results show that for the same total number of processors, the best hybrid run performs comparably as the pure MPI implementation. On the Power3 though, the hybrid results were significantly (and rather surprisingly) better than MPI. Adding more OpenMP threads beyond an optimal number, depending on the number of MPI tasks, did not improve performance. However, the primary advantage of the hybrid paradigm for overset grid applications is that it extends their applicability to large clusters of SMP nodes. In other words, hybrid programming is particularly appropriate when the number of overset grids is less than the number of processors to be used, or when load balancing becomes difficult due to the wide disparity in grid sizes [4].

7 Summary and Conclusions

In this paper, we presented a detailed performance analysis of a high-fidelity multi-block Navier-Stokes application on multiple state-of-the-art computer architectures. We implemented and used a hybrid (MPI+OpenMP) programming paradigm to exploit both coarse and fine-grain parallelism and extend the application's applicability to large clusters of SMP nodes. We considered a practical CFD simulation of vortex dynamics in the flow region of a complex configuration and conducted our experiments on the cacheless Cray SX6 vector processors, the cache-based IBM Power3 and Power4 architectures, and the single system image SGI Origin3000 platforms.

We showed the important role of restructuring a key kernel of the application, namely the LU-SGS linear solver, to improve performance on the above architectures. We analyzed and compared the runtime results and performance scalability on each architecture for both pure MPI and hybrid paradigms. We showed that in terms of execution timings, the SX6 outperforms the other three architectures; in fact, the best run time for eight processors on the SX6 is more than 40% less than the best 32-processor run on the Power4. We conclude that even though the pure MPI approach demonstrated a slight edge over the hybrid method, both paradigms still perform similarly for the same total number of processors, except for the Power3 where the hybrid results were significantly better. Finally, we showed that the hybrid scheme will be the more viable approach for extending multi-block applications to clusters of SMPs, for cases where the number of processors is comparable to or larger than the number of overset grids.

Acknowledgements

The authors would like to thank Dr. Leonid Oliker at Lawrence Berkeley National Laboratory for arranging access to the Cray SX6 and IBM Power3 machines. The first author was supported by NASA Ames Research Center under contract number DTTS59-99-D-00437/A61812D with AMTI/CSC.

References

[1] http://www.jamstec.go.jp. Earth Simulator Center.

[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 RNR-93-007, NASA Ames Research Center, Moffett Field, CA, 1993.

[3] P. G. Buning, D. C. Jespersen, T. H. Pulliam, W. M. Chan, J. P. Slotnick, S. E. Krist, and K. J. Renze. *Overflow User's Manual, Version 1.8g.* NASA Langley Research Center, Hampton, VA, 1999.

[4] M. J. Djomehri, R. Biswas, and N. Lopez-Benitez. "Load Balancing Strategies for Multi-Block Overset Grid Applications". In *Proc. 18th Intl. Conf. on Computers and Their Applications*, pages 373–378, 2003.

[5] M. J. Djomehri, R. Biswas, M. Postdam, and R. C. Strawn. "An Analysis of Performance Enhancement Techniques for Overset Grid Applications". In *Proc. 17th Intl. Conf. on Parallel and Distributed Processing Symposium*, 2003.

[6] M. J. Djomehri and H. H. Jin. "Hybrid MPI+OpenMP Programming of an Overset CFD Solver and Performance Investigations". Technical Report NAS-02-002, NASA Ames Research Center, Moffett Field, CA, 2002.

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

[8] 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.*, number AIAA-99-3302, 1999.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, MA, 1996.

[10] J. Steger, F. Dougherty, and J. Benek. "A Chimera Grid Scheme". *ASME FED*, (5), 1983.

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

[12] R. C. Strawn and M. J. Djomehri. "Computational Modeling of Hovering Rotor and Wake Aerodynamics". *AIAA Journal of Aircraft*, 39:786–793, 2002.

[13] J. R. Taft. "Achieving 60 GFLOP/s on the Production CFD code OVERFLOW-MLP". *Parallel Computing*, 27:521–536, 2001.

[14] 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, 1998.

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

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