Performance of a new CFD flow solver using a hybrid programming paradigm

M.J. Berger[a, b, *], M.J. Aftosmis[b], D.D. Marshall[c], S.M. Murman[d]

[a] *Courant Institute, New York University, New York, NY 10012, USA*
[b] *NASA Ames Research Center, MS T27B, Moffett Field, CA 94035, USA*
[c] *Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA*
[d] *ELORET, MS T27B, Moffett Field, CA 94035, USA*

Abstract

This paper presents several algorithmic innovations and a hybrid programming style that lead to highly scalable performance using shared memory for a new computational fluid dynamics flow solver. This hybrid model is then converted to a strict message-passing implementation, and performance results for the two are compared. Results show that using this hybrid approach our OpenMP implementation is actually marginally faster than the MPI version, with parallel speedups of up to 599 out of 640 using OpenMP and 486 with MPI.

Keywords: Parallel programming; Shared address space; Message passing; Space-filling curves

1. Introduction

One of the first choices to make when developing a new parallel application is whether to use a message-based distributed-memory model with MPI or a shared-memory programming model using parallel directives such as OpenMP. Both approaches have their advantages and disadvantages. MPI is the most portable across multiple platforms, since it runs on both distributed and shared-memory machines. With the number of disappearing languages and architectures in the last 20 years, it is the safer way to insure longer lasting software. On the other hand, since MPI involves an assembly-like attention to buffers and memory management, the MPI route is generally recognized as the more tedious approach. Shared-memory models offer a simpler path for code development. Often a developer of a shared-memory application will start with a serial code and put in simple loop-level OpenMP directives, obtaining incremental parallel improvements. Unlike MPI, when the parallel performance is sufficiently high, the parallelization effort can stop.

However, as has been reported in the literature, incremental parallelization does not typically yield good parallel scalability on large numbers of processors [14,20]. It relies too heavily on the compiler for loop level parallelization and does not pay enough attention to memory locality. Compiler directives and other hints are insufficient to transform a serial code into a high performing parallel code without more attention to memory issues early in the design process.

In this paper, we report on the development of a new computational fluid dynamics solver using a hybrid of these two approaches. Our paradigm is to do explicit memory management, complete with domain decomposition, duplicated variables, and explicit communication steps (in other words, close to a distributed-memory model code), but implemented in shared memory. This approach yields much higher performance than typically reported for shared memory, while still being easier to implement and debug than MPI code. It provides a simple path for a second step of

conversion to MPI for distributed-memory machines once the code has been tested and debugged. Experimental results with both of these implementations will be reported in this paper. By using a subset of OpenMP directives with careful attention to memory location, we believe that this approach will make it possible to avoid an MPI conversion, at least a manual one. Instead, we believe that by using a hybrid programming paradigm it should be possible to use software shared-memory layers that sit on top of distributed-memory machines and still obtain good performance.

In the rest of this section we present some background material describing what a flow solver looks like for the non-expert in computational fluid dynamics. We also review some material on space-filling curves, which we make heavy use of in the rest of this paper. In Section 2, we describe the new OpenMP parallel flow solver, including its most important aspects of domain partitioning, load balancing, and the data structures for communication. Several algorithmic innovations that contribute to the scalability, in addition to the hybrid programming paradigm, are described. Section 3 presents the small modifications needed to convert the code to MPI. Most data structures were completely suitable (and highly successful) for both approaches. Computational experiments for several realistically large computational examples, on both shared and distributed-memory machines, are presented in Section 4. Conclusions are in Section 5.

1.1. Background

In this section, we describe the salient features of our flow solver that affect its parallelization. We then review some of the basic properties of space-filling curves.

The flow solver was developed to solve the inviscid steady state Euler equation on multi-level Cartesian grids with embedded boundaries. Such grids have recently become popular largely because of the ease of grid generation around complicated geometries, along with their robustness and automation [2,6,4]. Fig. 1 illustrates a two-dimensional example of embedded boundary grids for purposes of discussion. Unlike typical body-fitted structured or unstructured grids, with embedded boundary grids the geometry simply intersects the underlying Cartesian grid in an essentially arbitrary way, creating general polyhedral cells next to the solid body which need special discretizations. However the bulk of the grid contains regular Cartesian cells, so finite volume schemes can be accurately and efficiently implemented. An essential ingredient for such methods is the use of a multi-level grid, so that cells at different levels of refinement can be used to accurately discretize both the geometry and the solution. This means some cells have more than one face in a given coordinate direction, but a mesh ratio of 2:1 is strictly enforced. Details of the grid generation can be found in [2].

The flow solver uses a finite volume discretization, where the flow quantities are stored at the centroid of each cell. Each iteration proceeds by computing the flux F_{ij} between

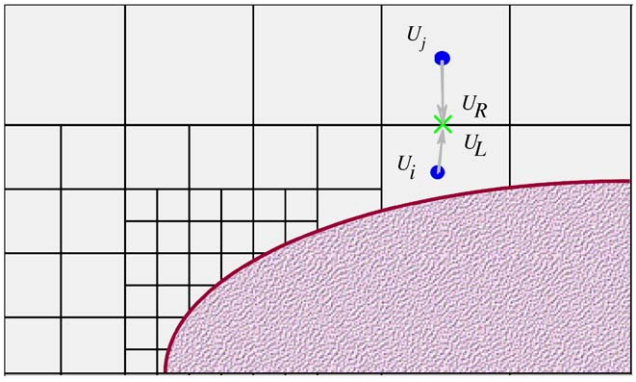

Fig. 1. Illustration in two dimensions of Cartesian mesh with embedded boundaries. Also shown are cells of different refinement levels, but adjacent cells are always 2:1.

cells i and j, at the cell edges, using an equation of the form

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{V_j} \sum_{\text{faces of cell } j} F_{ij}(u_{\mathrm{L}}, u_{\mathrm{R}}) A_{ij}. \qquad (1)$$

Here V_j is the volume (in three dimensions) of cell j, and A_{ij} is the "projected normal area" of the face between cells i and j. As is typical, the explicit iteration uses a multi-stage Runge–Kutta scheme, and the iterations continue until *steady state*, when the solution stops changing. Since the stencil is small and the scheme is explicit, communication takes place only between nearest neighbors. The value of the flux at each edge is determined from the solution as *reconstructed* from each adjacent cell, the left and right states u_{L} and u_{R}, and a non-linear Riemann problem produces the single *upwinded* state $u(u_{\mathrm{L}}, u_{\mathrm{R}})$ where the flux F is then evaluated. For a second order method, the value u_{L} is computed from the cell-centered value u_i and the cell's gradient ∇u_i. Computing the gradient is again a local operation, since it is based on solution values from only the nearest neighbors that share an edge with the cell. Another important component of the flow solver is the multi-grid acceleration scheme. The essence of this algorithm is to *restrict* the solution from the fine grid to a coarser one, where a solution is cheaper to compute and for technical reasons much of the error is reduced faster than on the fine grid. The correction to the solution is then *prolonged* back to the fine grid. This idea is recursively applied, with three or more levels often used in a multigrid hierarchy. For the numerical details of the flow solver see [1,3]. Note that in this application, some of the grid cells are full Cartesian hexahedra while others are cut by the embedded geometry, but this aspect is orthogonal to the parallelization efforts described below.

Although the grid generation for this type of grid has only recently been developed, the algorithms and data structures used in the flow solver follow established methods [5]. Typical light-weight data structures for these kinds of irregular grids use an *edge-based* data structure, which makes heavy

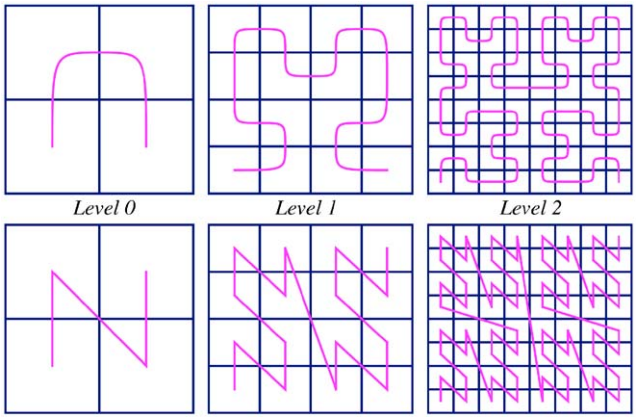

Fig. 2. Illustration of space-filling curves in two dimensions, both the Peano–Hilbert ("U") and Morton ("N") orderings. Three levels of meshes are shown.

use of indirect addressing. This consists of an array of cells and an array of faces. Cell-based information, for example includes the solution vector at each centroid, consisting of density, velocity and pressure. A cell does not know its nearest neighbor in this scheme however, since it is not stored using multi-dimensional rectangular indexing. Instead, the cells are ordered using space-filling curve indices, described next. The array of faces contains the index into the cell array of the adjacent left and right cells for each face. The only way cells exchange information with their neighbors is by sweeping over the face list. For the irregular cells adjacent to the embedded boundary, these arrays are augmented with additional information such as the surface normals, irregular cell centroids, which are also needed for Eq. (1). Faces are ordered in the face array according to the minimum index of their adjacent cells.

The face and cell arrays are ordered before the flow solution starts, for both good cache performance and to prepare for the domain partitioning described in Section 2. This ordering is performed using space-filling curves. We briefly review here some of their important properties; for more details see [8,9,18].

Space-filling curves (henceforth sfc) provide a linear ordering of a multi-dimensional Cartesian mesh. The basic building block of the Peano–Hilbert curve is a "U" shaped segment, which visits each cell in a 2 by 2 block, or an "N" shaped segment for a Morton ordering, as shown in Fig. 2. Subsequent levels replace the coarse cell with fine cells which are themselves visited consecutively by the basic curve. This implies the mesh is traversed in essentially the same order (physically in space) on both a coarse and fine grid. Note that a curve enters a cell from an adjacent cell through a common face, and leaves through another face to a different adjacent cell. A cell is thus connected to two neighbors in the one-dimensional ordering in the array of cells. This locality provides for good cache re-use.

2. Description of parallel flow solver

We illustrate the points raised in the introduction by discussing the choices made in developing a new flow solver for the inviscid steady-state Euler equations on multi-level Cartesian grids with embedded boundaries. Fig. 3 shows a three-dimensional grid around a realistically complex vehicle that will be used as an example throughout the remainder of the paper.

2.1. Domain partitioning via space-filling curves

The first step in implementing the flow solver using our hybrid distributed-memory programming model is to partition the mesh in a load-balanced fashion. This use of explicit domain decomposition and data replication is not a typical style for shared-memory machines, but is the essence of our hybrid approach. It is also much easier to implement using shared memory. As is common with domain decomposition, each processor is assigned a subdomain with a certain number of cells, and is responsible for updating those cells using the "owner computes" rule. Each domain is surrounded by one layer of "overlap" cells, so that a stencil update can be performed without communication. Each domain does not update its overlap cells, but receives the updated values from the subdomain that does own them after each update.

Since the underlying mesh is Cartesian, general purpose partitioners such as Metis [7] are unnecessarily expensive. Instead, a natural choice for partitioning these types of grids is to use space-filling curves [15–17]. Space-filling curves provide a one-dimensional ordering of a three-dimensional mesh, and guarantee that each cell is adjacent to at least two neighbors (see Fig. 4 for a two-dimensional illustration of this). Just prior to flow solution, the cells in the incoming Cartesian mesh are ordered using the space-filling curve ordering. To improve cache reuse, the faces are also lexicographically sorted according to the cell with the minimum sfc index. The total amount of "work" on a mesh is the sum of the work over the individual cells. In the Cartesian approach there are two distinct cell types: regular Cartesian-aligned hexahedra, and general "cut-cell" polyhedra adjacent to the body. These cell types require a different amount of computational work. Currently, the cut-cells are empirically determined to be 1.5 times the work of a full (uncut) cell. As with many codes, we do not directly account for the variation in number of faces, nor the communication work that each partition has (a function of the shared faces). This value was determined empirically using a linear least squares fit to the total execution time taken from serial computations with different size meshes.

One elegant feature of the sfc reordering is that when combined with the work estimates it is easy to partition the mesh into any number of subdomains at run-time. The number of processors is a run-time parameter (obtained from the OMP_NUM_THREADS environment variable),

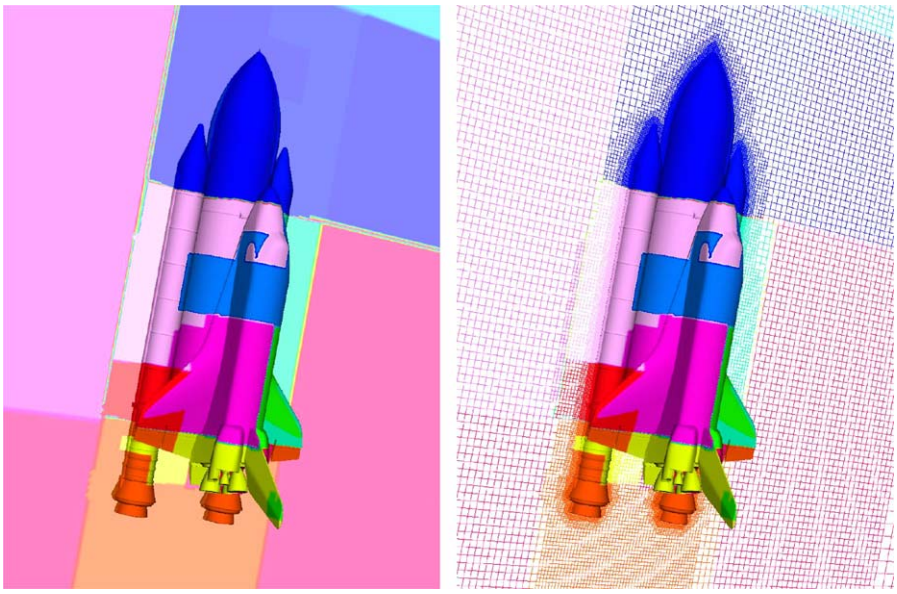

Fig. 3. Cartesian mesh with embedded geometry representation for space shuttle example. The domain is partitioned into 64 subdomains using Peano–Hilbert space-filling curves. Mesh and geometry are colored by partition number.

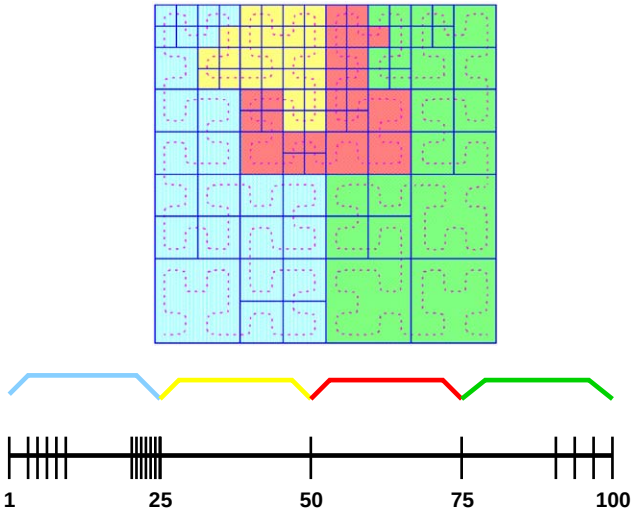

Fig. 4. Sample two-dimensional space-filling curve on a mesh with total amount of work=100 work units.

with the mesh partitioned using an on-the fly domain decomposition as it is input. This alleviates the need to determine in advance the number of partitions, or to re-partition if the number of available CPUs changes. Cells are assigned sequentially to the next processor until each node's assigned quota has been filled. (The assigned workload is the total work on the mesh divided by the number of processors). This can be thought of using a garden hose analogy—point the hose to the first partition; when it is full, move the hose to the second partition, etc. This is the step that transforms the global mesh into its partitioned counterpart, relying on information which straddles both views of the mesh. It is clearly much easier to implement using shared memory than MPI directives. At start-up time, we also ensure that memory associated with a sub-domain resides on the intended CPU, by initializing it right after allocating it, since some OS implementations wait until the first touch to allocate. After the read, the rest of the initialization and setup work is done in parallel.

The work estimates themselves can be exceedingly well balanced, with typical differences between the maximum and minimum load on the order of .0001%. This is easy to do, since for unstructured meshes the granularity of the partitioning is plus or minus one cell. Of course the work estimates are only a guess at the actual computational load, but the numerical experiments show these to be quite accurate.

The partitioning of the faces follows the cells. As the faces are input, if a face points to adjacent cells on the same partition, that partition owns the face. If the adjacent cells belong to two partitions, the face is duplicated, and the respective cells are put on the partition's overlapping cell list. Each domain explicitly copies its overlapping cells from the owner, so that a residual calculation can be done without inter-partition communication. This architecture follows the standard message passing template, except in this case it is implemented in shared memory rather than explicitly packing messages into paired sends and receives. Fig. 3 shows a domain partitioned into 64 subdomains, along with a cutting plane through the mesh colored by partition number as well.

Table 1 presents the number of overlapping cells as a function of the number of partitions and the size of the mesh. As can be seen in Fig. 5, space-filling curves have overlapping

Table 1
Statistics on the average number of overlap cells and the average number of neighbors the partition communicates with for three different meshes using the space-filling curve partitioning

# Parts	1.0 M Cells		4.7 M Cells		9.0 M Cells	
	# olap cells	# nbors	# olap cells	# nbors	# olap cells	# nbors
8	76 656 (7.5%)	3.7	267 558 (5.6%)	6.2	448 528 (5.0%)	—
16	127 159 (12.5%)	6.2	362 724 (7.6%)	8.2	595 177 (6.7%)	7.9
32	184 418 (18.1%)	7.5	458 174 (9.6%)	9.1	814 642 (9.0%)	8.9
64	249 929 (24.5%)	8.7	618 866 (13.0%)	9.6	1 070 983 (11.8%)	8.7

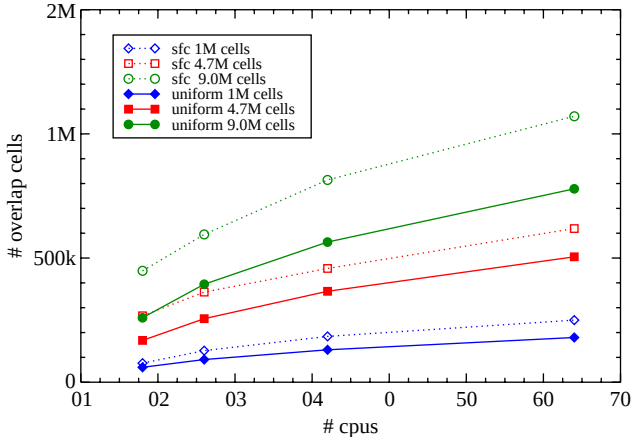

Fig. 5. Comparison of the number of overlap cells using the space-filling curve partitioning on a multi-level mesh (see Table 1) versus an idealized uniform mesh with the same total number of cells.

statistics that are close to the statistics one expects from a regular Cartesian mesh. On meshes from between 1 and 9 million cells, the number of overlapping cells ranges from a few percent on small numbers of processors to up to 25% of the cells. Note however that in this last case, with 64 CPUs there are only 15 K cells per cpu. As an additional bonus, space-filling curves have locality properties that are beneficial for good cache performance on each processor of a parallel machine.

Once the mesh is partitioned, the time stepping proceeds in SPMD fashion. Each processor computes the gradient for each cell, copies the gradient for the overlap cells from the adjacent processors, computes a residual, updates the cells it owns, and copies the new overlap cell values from adjacent processors. Since this is an explicit finite volume code, large chunks of code are executed in parallel with a coarser granularity than a typical fine-grained loop level parallelization. For example, one of the large chunk includes calculating the time step, computing the residual, and updating the solution. Another chunk is computing the gradient over the entire subdomain and computing the limiter for it. The type and number of cells on each domain varies, but as the numerical experiments show, the total time it takes to do these calculations is balanced across partitions.

After each computational chunk, synchronization followed by an explicit communication step is performed to update the copies of the overlap cells. Traditionally, these cells are not duplicated with shared-memory programming—one updates one's cells, and uses the neighboring cell as needed. However for performance with the hybrid paradigm, local memory is associated with each subdomain. Each computational node allocates and fills its own local memory for its partition, again not typical with shared-memory implementations. The alternative shared-memory strategy would have been to allocate one large array, and assign each partition a range of indices to update within the array. However, there is no way to enforce the memory locality for each subdomain with such a strategy. The communication step itself can be implemented in a very simple way using shared memory. For each overlap cell `j` the location to obtain the updated state is computed once and saved on the receiving processor. It is obtained using in essence a loop that looks like

```
// each partition performs the following procedure
  for (j=0;j<my_num_overlapCells;j++){
    p_myDomain_U[j]
      = p_otherDomain_U[j's_index_in_otherGrid]
  }
```

This shared-memory implementation requires a pointer *p_otherDomain_U*, determined at run-time, directly into the other domain. It is the only such pointer required for single grid calculations, with an additional shared-memory reference required for multigrid restriction and prolongation calculations. In MPI this step requires send/receive pairs and packing/unpacking of messages.

2.2. Multigrid via space-filling curves

A second big design decision for the new flow solver is how to implement the multigrid acceleration. The steps include creating the coarse meshes, partitioning them, and implementing the restriction and prolongation steps.

As with the sfc ordering performed with the initial mesh generation, the multigrid coarse meshes are generated

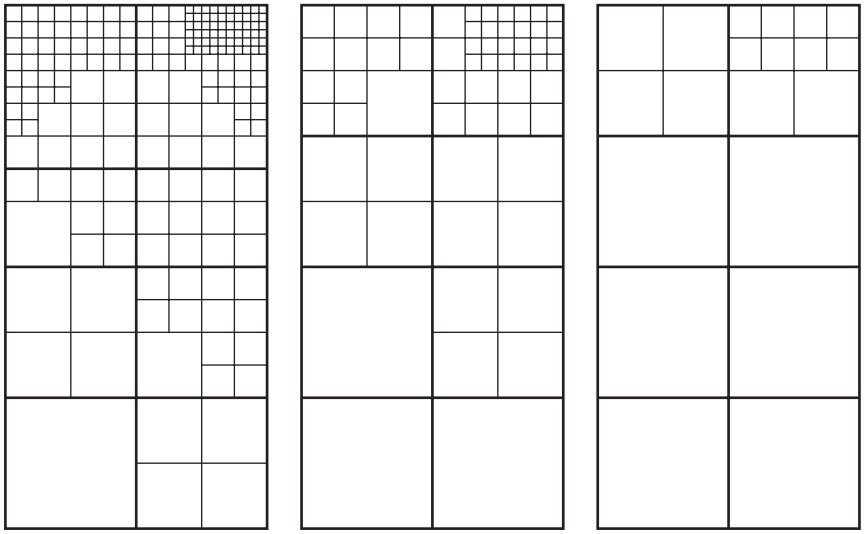

Fig. 6. Two-dimensional example of Cartesian mesh coarsening. Note that some cells do not coarsen because of the 1-irregular rule.

before the flow solution begins. Overall, the complete mesh preparation time, including initial mesh generation, space-filling curve ordering and multigrid mesh coarsening, typically takes only 2–3 min for meshes with several million cells, (using a 600 MHz SGI workstation), so we have not (yet) parallelized these steps. Since we use unstructured data structures to represent the mesh, it is not a trivial matter for a cell to coarsen: it must find its neighbors, check if they are coarsenable, and create the coarse face lists from the fine face information.

The coarse meshes are generated from the fine mesh in a novel way which uses the same sfc ordering as the partitioning. The key insight is to realize that this ordering places all sibling cells of a given parent sequentially in the sfc-ordered mesh. Thus, each coarse cell can be generated by agglomerating the (up to) eight fine cells in adjacent positions in the sfc-ordered mesh with a single pass. Cells are only allowed to coarsen if all the sibling cells are at the same level; otherwise, a cell is left alone, and a finer level cell will try to coarsen with its siblings by searching through the next eight cells. A second pass through the mesh checks that two adjacent cells are not more than one refinement level apart (the one-irregular rule), or the coarsening is disabled. Coarse mesh generation is thus a linear time algorithm in the number of cells in the mesh.

The mesh coarsening algorithm is illustrated in two dimensions in Fig. 6. A realistic example of the mesh coarsening is shown in Fig. 7 for a 4.5 million cell mesh. The coarsening ratios in generating four coarser meshes are 7.1, 6.5, 5.1 and 4.2, illustrating the retardation of the 8:1 coarsening as the mesh irregularity increases as a fraction of the total mesh. Nevertheless, the flow solver achieves excellent multigrid acceleration rates. Numerical experiments consistently show convergence rates of .85 to .95, comparable to the best 3D Euler solvers in the literature.

Once the coarse meshes are generated, they also need to be partitioned. The immediate decision is whether to partition the coarse meshes to conform with the fine meshes, which means the restriction step will have no communication, or whether to partition them independently so the coarse grid computations are load balanced. Since the grids are multilevel, they do not necessarily coarsen in the usual 8:1 ratio of uniform grids. Interfaces will retard the coarsening ratios, and the increasing percentage of cut cells in the mesh may lead to imbalanced work loads. Some multigrid implementations choose the partitioning induced by the finer mesh, so that the only communication comes from coarse cells with parent cells from two different meshes. Others partition the coarse mesh anew [11], and then look for maximal overlap with a fine partition in assigning the coarse partitions to a CPU. This is sometimes followed by a bartering algorithm where neighboring sub-domains exchange boundary cells to try to minimize the communication.

With sfc-ordered meshes, we can implement either approach directly, since the enumeration of the cells follows the same layout on the coarse and fine grids. We have chosen to partition the coarse mesh in a load balanced way. This may lead to additional communication from coarse cells that straddle one or more partitions on the fine grid (illustrated in Fig. 8). However numerical experiments show that this potential bottleneck is not encountered, since the multigrid scalability results are nearly as good as the single grid results.

In many implementations, if there is a lot of communication the fine grid would first coarsen itself, and then send the coarse values to the appropriate coarse partitions. However, since most of the cells are on the same partition, this preliminary coarsening step is not necessary, and the fine grid simply adds its value to the coarse data structure. Note that this step needs some kind of synchronization for coarse

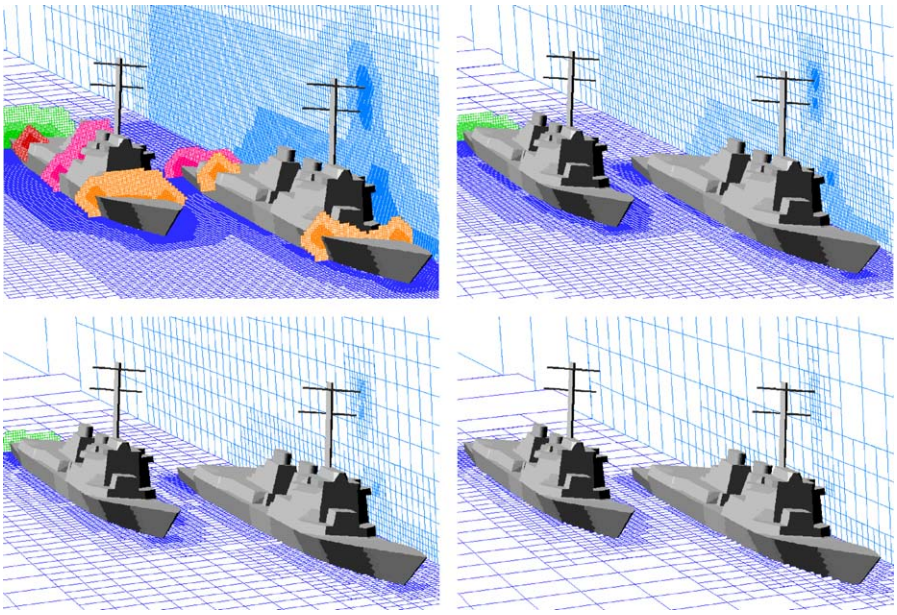

Fig. 7. Sequence of coarser grid levels for a 4.5 million cell mesh. The coarsening ratios are 7.1, 6.5, 5.1, and 4.2 in this example. (The last mesh is not shown.)

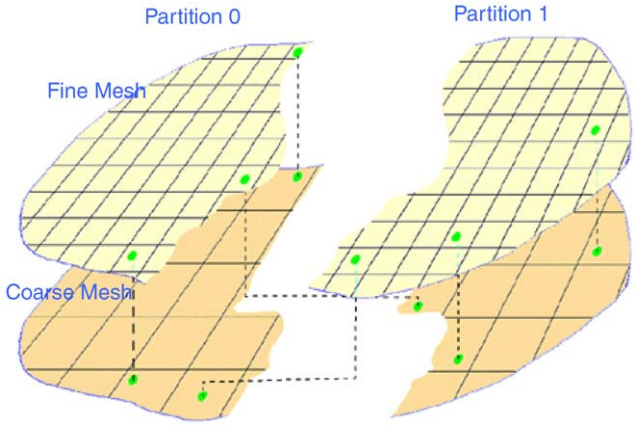

Fig. 8. Coarse and fine grids are separately partitioned using the sfc ordering. Note that fine cells on different partitions may restrict to the same coarse cell.

cells that are *shared* by two or more fine mesh partitions, as in Fig. 8. For the OpenMP implementation this is done using an atomic directive in the restriction loop.

3. Conversion to MPI

The conversion of the flow solver to MPI started with the OpenMP implementation. Due to the coarse granularity of the OpenMP parallelization, over half the code needed little (under 30 lines) or no modification. In fact, of the 11 000 lines of code organized into 18 files, only the three I/O files changed substantially, with an additional 1800 lines of MPI packing code, derived type assembly, etc.

One simple strategy made all the parallel loops work in both paradigms. The standard loop over a sub-domain is wrapped with a macro as follows:

```
# pragma omp parallel
{
  int myPartitionNumber
    = GET_MY_PARTITION_NUM;
  do_something_on_my_subdomain;
} /* -- end parallel region -- */
```

In OpenMP, the GET_MY_PARTITION_NUM macro becomes a call to the function `omp_get_thread_num()`. With MPI code however all processors are running all the time, and a parallel region does not need to be invoked like this. Every partition might as well think of itself as partition 1, which the macro evaluates to in MPI. Where needed, additional MPI `get_my_procnum()` calls are inserted. Of course, the reduction operations and all I/O routines needed to be completely rewritten.

While the data structures involved in flow computations on a domain were suitable for both paradigms, the data structures involved in any communication step had to be completely changed. This includes for example, multigrid restriction and prolongation, and sharing of overlapping cell information between sub-domains. A shared memory-based implementation is more naturally served by a GET operation: domain A *gets* its overlap information from the owner domain B. This is efficiently accomplished by domain A with the use of pointers to the overlap information in domain B. Translating this to message based communication would require two steps: first, domain A has to request the overlap information from domain B, then domain B has to

Table 2
Average number of cells on the fine mesh per partition that communicate to the coarse mesh on a different partition during the restriction step for a 4.7 M cell mesh

# partitions	Avg. # fine cells	Avg. # fine cells restricting to different partition
8	593 824	46 925 (7.9%)
16	296 912	36 206 (12.2%)
32	148 456	28 670 (19.3%)
64	74 228	24 284 (32.7%)
128	37 114	21 228 (57.2%)

return the overlap information. A more efficient implementation for message based communication is to use a SEND operation: domain B *sends* the information needed for the overlap to domain A. This eliminates the first step of the *get* operation and requires only one communication step.

Table 2 shows the number of cells on the fine grid that restrict the solution to a different partition on the coarser grid. As the number of CPUs increases, the number of cells per CPU decreases, but the percentage that communicate increases. It is clear that until very high numbers of CPUs are reached, the percentage of the fine and coarse grids on the same partition is very high, minimizing the communication bandwidth required for restriction and prolongation Note that on 64 CPUs there are only 75 K fine grid cells per node and 24 K coarse grid cells with this mesh, and the communication percentages are rising, yet the scalability results are not greatly affected by this. However, if our final goal were only an MPI implementation, a better choice may have been to partition the coarse grid to follow the fine grid (and also to ensure that the fine partitions ended on coarse cell boundaries). This was not investigated further.

4. Computational experiments

The scalability studies presented here use a variety of mesh sizes on two different machine architectures. The CPU times measure ten cycles, where the work per cycle includes a 5 stage Runge–Kutta scheme with gradient evaluations at every stage. The timing excludes the cost of the initial start-up and final I/O.

In the first experiment we compare the OpenMP and MPI versions of the flow solver for a single grid. This was run on the NASA Ames SGI Origin 3600, which has 600 MHz R14000 MIPS CPUs. This shared-memory machine has 1024 CPUs, of which subsets were reserved for this study. The machine was not in dedicated mode for these runs. For the OpenMP experiments, the environment variable OMP_DYNAMIC was set to false. The MPI implementation in this experiment is the native SGI version. Experiments show that MPICH is slower for these tests (see [10]) and it is not included in this set of experiments. The mesh contains 4.7 million cells, although also included here is an example with a smaller mesh of 1.6 million cells run

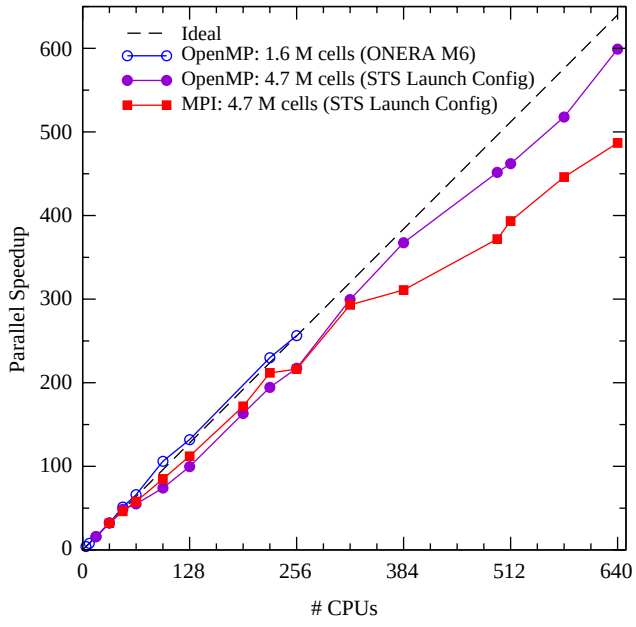

Fig. 9. Performance comparison of MPI and OpenMP code on SGI Origin 3600 for single grid computations.

on smaller numbers of nodes. Note that the 4.7 M cell mesh has fewer than 7200 cells per CPU on 640 processors, and the 1.6 million cell mesh has 6250 cell on 256 nodes. The smaller mesh speedup curve shows the superlinear speedup that comes from fitting more of the mesh in cache. On the larger mesh, speedups of 599 for the OpenMP version and 486 for the MPI version were obtained on 640 processors. Obviously the space-filling curve load balancing algorithm is doing a good job of partitioning the work to obtain this degree of scalability. The computation does not fit on one processor, so scalability is measured relative to timings on 8 CPUs.

As Fig. 9 shows, initially both MPI and OpenMP perform equivalently, but the MPI version begins to slow down on more then 320 CPUs. This seems to indicate slightly greater overhead in the MPI version. If this is the case, then the slowdown should be more pronounced when using multigrid, which has greater emphasis on communication.

The next set of experiments includes the multigrid acceleration scheme in comparing OpenMP and MPI. Fig. 10 shows results using the same 4.7 M cell mesh. We use a multigrid W-cycle with three grid levels and one pre- and one post-sweep per level. The MPI runs has a speedup of 392 on 640 CPUs. The OpenMP achieves 514 on 640 CPUs, measured relative to a 32 node baseline. For reasonable numbers of CPUs, the multigrid scalability is slightly worse than for a single grid, despite the fact that surface to volume ratio of the partitioning is increasing rapidly. The 4.7 million cell fine mesh has 700 000 cells in the first coarser mesh (for an average of approximately 1100 cells per node on 640 CPUs), and 105 K cells on the second coarser mesh (for an average of only 180 cells per node), but the fact that the multi-

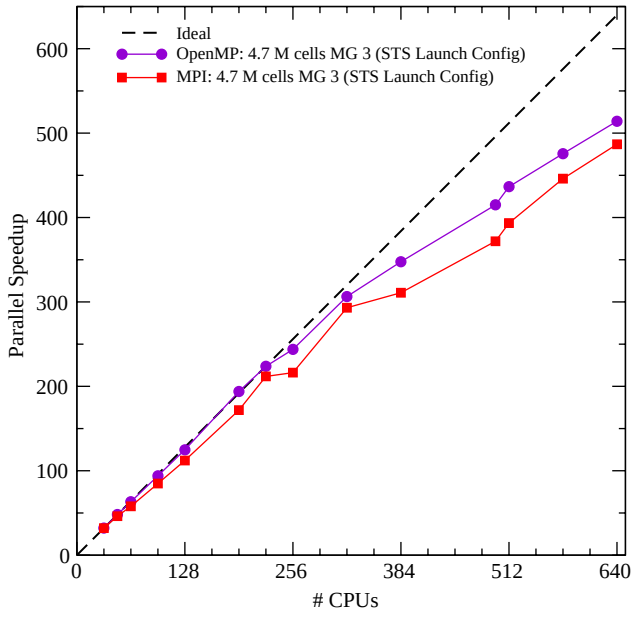

Fig. 10. Performance comparison of MPI and OpenMP code on SGI Origin 3600 using multigrid W-cycle with 3 grid levels.

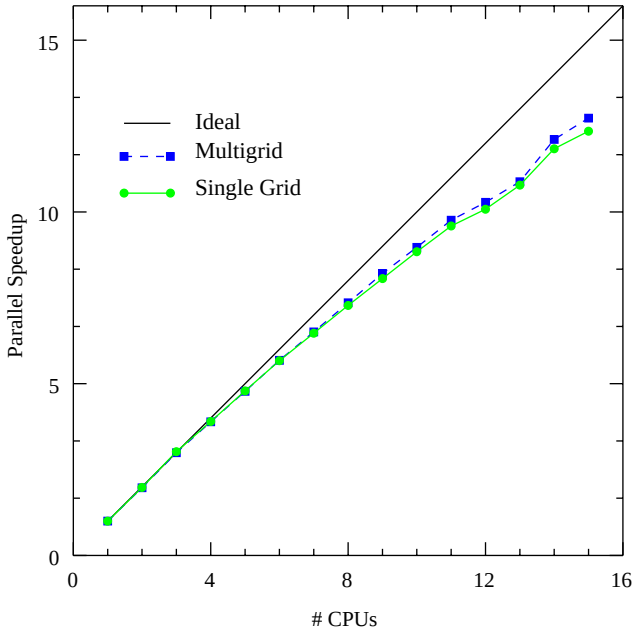

Fig. 11. Scalability study on Linux cluster using MPICH—single 4.7 M cell grid and multigrid with 3 levels.

grid communication overhead is creeping up (the results in Table 2) has little effect.

At first glance, these results appear surprising in that the OpenMP version outperforms the MPI version. This holds even when only the core components of the code are executed, i.e. no gradients, no limiters, and no multigrid. The OpenMP version is still somewhat more scalable. Detailed instrumentation reveals that the communication routine where the solution in the overlap cells is retrieved from neighboring partitions (OpenMP version), or buffered and sent to neighbors (MPI version), consistently takes between 2 to 3 times longer in MPI. This additional overhead is a small fraction of the runtime relative to the total integration time of the solution. However on larger numbers of CPUs this overhead becomes increasingly dominant as the rest of the code speeds up. This situation is exacerbated when gradients are included as part of the computation, since three times as much information needs to be communicated twice per solution update (once for the gradients themselves, and once for the limited gradients in the overlap cells).

Since we are using the native SGI MPI library, we can only speculate as to what is causing this slowdown. Analysis using Paraver [1] [19] shows that the cause is internal to the MPI calls, with an order of magnitude more instructions being executed inside the MPI calls than the OpenMP version. Variations using Send/Irecv or Isend/Recv combinations do not affect this outcome. Actually we believe that our MPI performance is quite good, and similar to other careful MPI applications; what is surprising is the scalability of the OpenMP code.

[1] Thanks to Gabriele Jost at NASA Ames for the Paraver results.

The last experiment, in Fig. 11, shows results of the MPI code using MPICH on a 16 node 2.4 GHz Pentium 4 Linux cluster with a Gigabit ethernet switch. The single grid and multigrid speedups are essentially identical, although the actual wall clock time of the multigrid is approximately 40% more than for a single grid. Since the timings are so close, it is clear that the additional communication of multigrid is easily handled by the network.

5. Conclusions

We have presented our approach to shared-memory programming using a hybrid paradigm. By adopting many of the explicit domain decomposition and memory management techniques of distributed-memory programming but implementing them with the substantially easier shared-memory constructs, we can demonstrate performance with some of the best scalability results for shared memory on a real application yet obtained. Our scalability results are especially significant since OpenMP results using loop-level constructs found in the literature have been so disappointing. Similar results were found in [13].

By careful attention to memory placement and locality, our OpenMP implementation is actually marginally faster than MPI. We believe this is because there is less time consumed packing and unpacking buffers, and because of the overhead of MPI. The one-sided communication available in MPI2 may help in this regard. Additional steps for better performance would be to investigate use of parallel I/O. Ideas along the lines of those in [12] could also be very useful in further scalability improvements.

The hybrid paradigm uses a restricted set of OpenMP functionality, and follows closely the architecture of MPI codes. Thus we believe that an automatic conversion tool from OpenMP to MPI should be possible to a large degree. Alternatively, a shared-memory layer sitting on top of distributed memory, such as found in Treadmarks, Hamster, DSM-THREADS and the like, might provide good performance with the hybrid paradigm. However, this software was either not available or insufficiently stable for us to include here.

Acknowledgments

Marsha Berger was supported by AFOSR grant F19620-00-0099 and by DOE grants DE-FG02-00ER25053 and DE-FC02-01ER25472. David Marshall was supported by NASA's Graduate Student Research Program.

References

[1] M.J. Aftosmis, M.J. Berger, G. Adomavicius, A parallel multilevel method for adaptively refined cartesian grids with embedded boundaries, AIAA Paper 2000–0808, Reno, NV, January 2000.

[2] M. Aftosmis, M. Berger, J. Melton, Robust and efficient cartesian mesh generation for component-based geometry, AIAA J. 36 (6) (June 1998).

[3] M.J. Berger, M.J. Aftosmis, G. Adomavicius, Parallel multigrid on cartesian meshes with complex geometry, In: Computational Fluid Dynamics: Trends and Applications, Elsevier, 2001; Proceedings of the Parallel CFD Conference, Trondheim, Norway, May 2000.

[4] E. Charlton, K. Powell, An octree solution to conservation laws over arbitrary regions (OSCAR), AIAA paper 97-0198, January 1997.

[5] A. Jameson, W. Schmidt, E. Turkel, Numerical solutions of the Euler equations by finite volume methods using Runge–Kutta time-stepping schemes, AIAA Paper 81-1259.

[6] S. Karman Jr., SPLITFLOW: A 3D unstructured crtesian/prismatic grid CFD code for complex geometries, AIAA paper 95-0343, January 1995.

[7] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, Technical Report, Department of Computer Science TR95-035, University of Minnesota, 1995.

[8] X. Liu, G. Schrack, The spatial U-order and some of its mathematical characteristics, Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Victoria BC, Canada, May 1995.

[9] X. Liu, G. Schrack, Encoding and decoding the Hilbert order, Software—Practice Exp. 26 (12) (December 1996).

[10] G.R. Luecke, M. Kraeva, L. Ju, Comparing the performance of MPICH with Cray's MPI and with SGI's MPI, Concurrency and Computation: Practice and Experience, 2002.

[11] D. Mavriplis, Three-dimensional high-lift analysis using a parallel unstructured multigrid solver, ICAS Report No. 98–20, May 1998.

[12] J. No, S. Park, J. Perez, A. Choudhary, Design and implementation of a parallel I/O runtime system for irregular applications, J. Parallel Distrib. Comput. 62 (2) (February 2002).

[13] L. Oliker, R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, IEEE Trans. Parallel Distrib. Systems 11 (2000).

[14] L. Oliker, X. Li, P. Husbands, R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Rev. 44 (3) (2002).

[15] M. Parashar, J.C. Browne, Distributed dynamic data-structures for parallel adaptive mesh refinement, Proceedings of the International Conference on High Performance Computing, 1995.

[16] J.R. Pilkington, S.B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, IEEE Trans. Parallel Distrib. Systems 7 (3) (March 1996).

[17] J.K. Salmon, M.S. Warren, G.S. Winckelmans, Fast parallel tree codes for gravitational and fluid dynamical N-body problems, Internat. J. Supercomput. Appl. 8 (2) (1994).

[18] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading MA, 1990.

[19] See http://www.cepba.upc.es/paraver.

[20] H. Shan, J.P. Singh, L. Oliker, R. Biswas, A comparison of three programming models for adaptive applications on the origin2000, J. Parallel Distrib. Comput. 62 (2) (February 2002).