AIAA JOURNAL
Vol. 32, No. 3, March 1994

Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives

R. Das,* D. J. Mavriplis,† J. Saltz,‡ S. Gupta,§ and R. Ponnusamy¶
Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665

This paper is concerned with the implementation of a three-dimensional unstructured-grid Euler solver on massively parallel distributed-memory computer architectures. The goal is to minimize solution time by achieving high computational rates with a numerically efficient algorithm. An unstructured multigrid algorithm with an edge-based data structure has been adopted, and a number of optimizations have been devised and implemented to accelerate the parallel computational rates. The implementation is carried out by creating a set of software tools, which provide an interface between the parallelization issues and the sequential code, while providing a basis for future automatic run-time compilation support. Large practical unstructured grid problems are solved on the Intel iPSC/860 hypercube and Intel Touchstone Delta machine. The quantitative effects of the various optimizations are demonstrated, and we show that the combined effect of these optimizations leads to roughly a factor of 3 performance improvement. The overall solution efficiency is compared with that obtained on the Cray Y-MP vector supercomputer.

I. Introduction

THE relatively rapid growth in microprocessor technology over the last decade has led to the development of massively parallel hardware capable of solving large computational problems. Given the relatively slow increases in large mainframe supercomputer capabilities, it is now generally acknowledged that the most feasible and economical means of solving extremely large computational problems in the future will be with highly parallel distributed memory architectures. Although such hardware already exists and is rapidly progressing, the software required to solve large problems in parallel has proved to be a major stumbling block. The software problems are twofold. First, an efficient and inherently parallelizable algorithm must be devised for solving the problem at hand. Algorithmic efficiency relates to the ability to solve a problem with a minimum number of operations or iterations. Thus, simple explicit schemes are not generally suitable for large problems, and more complex implicit or multigrid schemes that propagate information more rapidly throughout the domain are preferred. A parallelizable algorithm is one that can be broken up into smaller components and executed on a parallel architecture without incurring substantial overheads, both in terms of reduced computational efficiency (increased number of operations to achieve the same level of convergence) and increased communication costs. Although simple explicit schemes are often easily parallelizable, they are not efficient. On the other hand, considerable difficulty has often been experienced in parallelizing more complex algorithms. This problem is compounded in the case of unstructured grids, since relatively few efficient algorithms have been devised, even for sequential (vector) architectures. The second software problem is an implementation issue. Most often, the programmer is required to explicitly distribute large arrays over multiple local processor memories and to keep track of which portions of each array reside on which processors. To access a given element of a distributed array, communication between appropriate processors must generally be invoked, unless the array element is resident in the processor responsible for the request. The programmer should be relieved of such machine-dependent and architecture-specific tasks. Ultimately, a parallelizing compiler should be capable of automatically distributing data and setting up inter-processor communication in an efficient manner, much as present-day coarse-grained shared memory supercomputers provide automated support for multiprocessing. Such low-level implementational issues have severely limited the growth of parallel computational applications, much in the same way as vectorized applications were inhibited early on, before the development of efficient vectorizing compilers.

This work represents the combination of two related efforts aimed at easing the software problem. On the one hand, an efficient three-dimensional unstructured solver has been developed that is highly parallelizable. The sequential version of this algorithm has previously been reported.[1] The data structures and solution strategy (multigrid) have been designed (or chosen) with parallel overhead issues in mind. (These also have a beneficial effect on the sequential code.) On the other hand, a set of parallelization software tools (primitives) has been developed to ease the task of implementing a broad class of unstructured and adaptive codes on parallel architectures. The development of these primitives has been heavily influenced by the requirements of the three-dimensional unstructured Euler solver described in this paper. Our runtime support has already been used in the implementation of several different Fortran-based prototype distributed memory compilers. These compilers are not yet capable of handling full applications codes, but preliminary performance results are encouraging.[2-4] In recent work, we have extracted procedures from both unstructured mesh Euler solvers and from molecular dynamics codes and parallelized these procedures (using realistic meshes and molecular data sets as input data). We currently see a performance difference of less than 10% when we compare the performance of hand parallelized codes with the performance we obtained from our compiler.[4] The development of these primitives, known as the PARTI primitives (parallel automated runtime toolkit at ICASE), as well as the compiler, has been under way for some time, and has been previously reported.[5] In this paper, we do not address the issues associated with the use of PARTI as the runtime support for a compiler; here we assume that the user ex-

Presented as Paper 92-0562 at the AIAA 30th Aerospace Sciences Meeting, Reno, NV, Jan. 6–9, 1992; received April 4, 1992; revision received May 15, 1993; accepted for publication June 27, 1993.

*Staff Scientist; currently Research Associate, Department of Computer Science, University of Maryland, College Park, MD 20742.
†Senior Staff Scientist. Member AIAA.
‡Lead Computer Scientist; currently Associate Professor, Department of Computer Science, University of Maryland, College Park, MD 20742.
§Research Fellow; currently Senior Applications Engineer, Intel Supercomputer Systems Division.
¶Research Fellow; currently Faculty Research Assistant, Department of Computer Science, University of Maryland, College Park, MD 20742.

plicitly invokes the primitives at the appropriate locations in the source code. Although this does not provide a completely automatic parallelization tool, it does relieve the user of the most burdensome low-level details of the implementation. This paper is less concerned with the individual development of the solver or the parallelizing primitives than with the interaction between these two efforts resulting from a specific application: the implementation of the unstructured Euler solver on the Intel iPSC/860. For example, the edge-based data structure employed in the solver was originally chosen to minimize and simplify communication between processors. Similarly, experience gained during the implementation was used to modify various primitives and even to create new primitives whose functionality had not been foreseen.

Parallel unstructured solver implementations have been performed in two-dimensions by various authors, on single instruction multiple data (SIMD) architectures,[6,21] and on distributed memory multiple instruction multiple data (MIMD) architectures.[7–9,22,23] Software environments for irregular problems have also been developed and applied to two-dimensional unstructured grid solvers,[8] and to two dimensional unstructured multigrid solvers. Although much use has been made of the concepts developed in previous work, new optimizations have also been devised and incorporated. This collection of techniques has been encapsulated into a set of software primitives that provides the interface between the parallel implementation and the sequential code.

In the next section, a brief description of the unstructured multigrid Euler solver is given. Thereafter, a description of the general parallelization approach and of the various optimizations is given, as well as a description of the underlying philosophy of software tool development. The goal of the results section is to illustrate the beneficial effects on performance of each of the individual optimizations and to compare the obtained performance with that of present-day supercomputers. Finally, the solution of a large practical unstructured grid problem is demonstrated and the parallel efficiency is compared with that obtained on the Cray Y-MP/8 vector supercomputer.

II. Three-Dimensional Multigrid Euler Solver

The basis for the implementation is a three-dimensional unstructured mesh Euler solver. Unstructured meshes provide a great deal of flexibility in discretizing complex domains and offer the possibility of easily performing adaptive meshing. However, unstructured meshes result in random data sets and large sparse matrices, which pose a significant challenge for parallelization issues.

The three-dimensional compressible gasdynamics equations are discretized on the unstructured mesh using a Galerkin finite element approach,[10] and piecewise linear flux functions are assumed over the individual tetrahedra of the mesh. The resulting spatially discretized equations can be recast as a summation at each vertex of contributions along all edges meeting at that vertex. Thus, the convective residuals may be assembled by performing a simple loop over the edges of the mesh. Artificial dissipation terms are required to stabilize the solution, and these are constructed as a blend of a Laplacian and biharmonic operator, the former being constructed as a single loop over edges and the latter as a double edge loop. The spatially discretized equations thus form a large set of coupled ordinary differential equations, which must be integrated in time to obtain the steady-state solution. This is achieved using a five-stage Runge-Kutta scheme. Enthalpy damping, local time stepping, residual averaging, and an unstructured multigrid algorithm are employed to accelerate convergence to steady state. In the multigrid algorithm, at each cycle a single time step is first performed on the finest grid of the sequence and the flow variables and residuals are then interpolated up to a coarser grid. This process continues recursively on successively coarser grids. When the coarsest grid is reached, the corrections are interpolated back to each successively finer grid, and a new cycle is initiated. In the context of unstructured meshes, it has proven useful to rely on sequences of independent nonnested coarse and fine meshes. To efficiently interpolate variables between such meshes, an efficient search algorithm must be invoked to determine the patterns (addresses and weights) for interpolation between any two consecutive meshes of the sequence. This is done in a pre-processing stage, on a sequential machine, before the flow computations. Alternatively, this may be viewed as a mesh generation postprocessing stage. The basic data structure for the Euler solver is based on the mesh edges. For each edge, we store the addresses of the two vertices on either end of the edge (similar to the coordinate-storage scheme for sparse matrices). This represents the minimum amount of information necessary to describe the unstructured grid. It also results in the minimum amount of data transfer between adjacent vertices within a residual evaluation operation, by avoiding duplicate transfers that are usually incurred by face-based and traditional finite element cell-based data structures. For parallel implementations, this results in the minimum amount of communication and enables a relatively simple implementation, since each edge can be shared by at most two processors. The multigrid interpolation procedures, in which the addresses and weights for interpolation have been precomputed, can be viewed as a simple gather / scatter of data from one array (grid) to another. Such operations are similar to the gather-scatter operations required on a given grid for assembling the residuals and can therefore be implemented with existing software primitives.

III. Parallelization (PARTI) Primitives

The PARTI primitives are designed to ease the implementation of computational problems on parallel architecture machines by relieving the user of the low-level machine specific issues. The research described in this paper began with the version of PARTI described in Ref. 11 and surveyed in this section. We then proceeded to identify ways in which the performance of unstructured codes could be optimized. The optimizations that involved reduction of communication overheads resulted in an improved version of PARTI. The optimizations that involved reduction of computation time were manually implemented.

The PARTI primitives enable the distribution and retrieval of globally indexed but irregularly distributed data sets over the numerous local processor memories. In distributed memory machines, large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called distributed arrays. Long-term storage of distributed array data is assigned to specific memory locations in the distributed machine. A processor that needs to read an array element must fetch a copy of that element from the memory of the processor in which that array element is stored. Alternately, a processor may need to store a value in an off-processor distributed array element. Thus, each element in a distributed array is assigned to a particular processor, and to be able to access a given element of the array, we must know the processor in which it resides and its local address in this processor's memory. We thus build a translation table that, for each array element, lists the host processor address. For a one-dimensional array of N elements, the translation table also contains N elements and therefore must be distributed itself over the local memories of the processors. This is accomplished by putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc., where P is the number of processors. Thus, if we are required to access the mth element of the array, we look up its address in the distributed translation table, which we know can be found in the $[(m-1)*P/N+1]$th processor. Alternatively, we could simply renumber all of the vertices of the unstructured grid to obtain a regular partitioning of arrays over the processors. However, the present approach can easily deal with arbitrary partitions and should enable a straightforward implementation of dynamically varying partitions, which may be encountered in the context of adaptive meshing. One of the primitives handles initialization of distributed translation tables, and another primitive is used to access the distributed translation tables.

PARTI carries out optimizations that reduce both the number of messages sent as well as the volume of data that must be communicated. In distributed memory MIMD architectures, there is typically a nontrivial communications latency or startup cost. For efficiency reasons, information to be transmitted should be collected

into relatively large messages. The cost of fetching array elements can be reduced by precomputing what data each processor needs to send and to receive. In irregular problems, such as solving partial differential equations on unstructured meshes and sparse matrix algorithms, the communications pattern depends on the input data. This typically arises due to some level of indirection in the code. This lack of information is dealt with by transforming the original parallel loop into two constructs called inspector and executor. During program execution, the inspector examines the data references made by a processor and calculates what off-processor data need to be fetched and where those data will be stored once they are received. The executor loop then uses the information from the inspector to implement the actual computation. The PARTI primitives can be used directly by programmers to generate inspector/executor pairs. Each inspector produces a communications schedule, which is essentially a pattern of communication for exchanging data.

Significant work has gone into optimizing the gather, scatter, and accumulation communication routines for the Intel iPSC/860. During the course of developing the PARTI primitives, we experimented with a large number of ways of writing the kernels of our communication routines. It is not the purpose of this paper to describe these low-level optimizations or their effects in detail; we will just summarize the best communication mechanism we have found. In all of the experimental studies reported in this paper, we use the optimized version of the communication routine kernels.

We communicate using FORCED message types (where explicit checking for free buffer space is not performed in the communication protocol). We use nonblocking receive calls (irecv); each processor posts all receive calls before it sends any data. Synchronization messages are employed to make sure that an appropriate receive has been posted before the relevant message is sent.

Communications contention is also reduced. We use a heuristic developed by Venkatakrishnan et al.[7] to determine the order in which each processor sends out its messages. The motivation for this heuristic is to reduce contention by dividing the communication into groups of messages such that within each group, each processor sends and receives at most one message. As pointed out in Ref. 7, this heuristic makes the tacit assumption that all messages are of equal length and in any event does not even attempt to eliminate link contention.

IV. Communications Optimizations

Our communication optimizations reduce the quantity of data that must be transmitted between processors. The optimizations also reduce the number of messages that must be sent.

In our unstructured mesh solver, we encounter a variety of situations in which the same data are accessed by several consecutive loops. For instance, consider a step of the Runge-Kutta integration. Flow variables are used in a sequence of three loops over edges followed by a loop over boundary faces. The flow variables are only updated at the end of each of the Runge-Kutta steps. We can obtain all of the off-processor flow variables needed at the beginning of the step. This makes it advantageous to develop methods that avoid bringing in the same data more than once. These methods can also reduce the number of communication startups.

Our new methods make it possible to track and reuse off-processor data copies. We do this by modifying our software so that we are able to generate incremental communications schedules. Incremental schedules obtain only those off-processor data not requested by a given set of pre-existing schedules. The pictorial representation of an incremental schedule is given in Fig. 1. In this figure, we depict a situation in which a loop over mesh edges (edgeloop) is followed by a loop over mesh boundary faces (faceloop). The schedule to bring in the off-processor data for the edgeloop is given by the edge schedule and is formed first. During the formation of the schedule to bring in the off-processor data for the faceloop, we remove the duplicates shown by the shaded region in Fig. 1. Removal of duplicates is achieved by using a hash table. The off-processor data to be accessed by the edge schedule are first hashed using a simple hash function. Next, all of the data to be accessed during the faceloop are hashed. At this point the information that exists in the hash table allows us to remove all of the duplicates and form the incremental schedule.

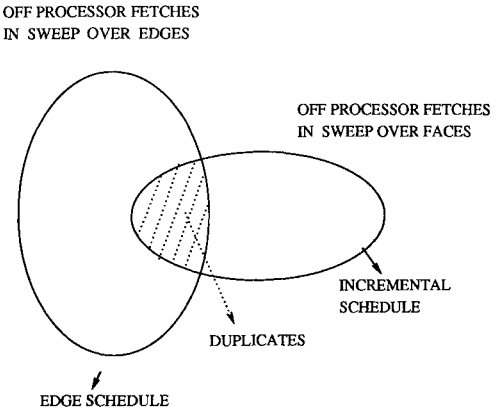

Fig. 1 Incremental schedule.

V. Reordering Computation

The performance of a single Intel iPSC/860 processor on computationally important loops over mesh edges ranges from 2.75 to 4.1 Mflops (million floating-point operations per second), in 64-bit arithmetic, depending on the unstructured mesh used. It seems likely that this relatively poor performance is due to the effects of irregular data access patterns on the Intel iPSC/860 memory hierarchy. The lowest level of iPSC/860 memory hierarchy consists of 32 integer and 32 floating point registers, each 32 bits wide. Data stored in registers can be used directly for computation without any memory overheads. At the next level of memory hierarchy, the iPSC/860 has an 8 kbyte data cache. The cache line size is 256 bits, and two double precision floating point numbers can be loaded from main memory into the cache simultaneously. The data stored in the cache can be accessed with a delay of one clock cycle. In cases where data are not available in the cache, it is loaded from memory, which causes a delay of several cycles. Other than possibly poor utilization of registers and cache, highly irregular data access patterns can potentially cause severe performance degradation due to overheads associated with the iPSC/860's mechanism for handling virtual memory. The virtual memory of the iPSC/860 is divided into 4 kbyte pages. Whenever a new page is accessed, a substantial overhead is incurred, associated with locating the page in physical memory. This overhead can be saved if the page has recently been accessed. The details of each component of iPSC/860 memory hierarchy can be found in Ref. 12.

The single processor performance can be improved by reordering the data and by restructuring the code associated with unstructured mesh computations. Because we employ an edge-based data structure in our codes, most of the computational work in the code is found in loops over edges so we concentrate our efforts on such loops. We investigated methods that change the order in which mesh edges are traversed in loops. We also investigated methods that renumber mesh vertices and reorder data associated with the mesh. Reordering edges and renumbering vertices is expected to result in better locality and therefore should improve cache utilization and reduce virtual memory management overheads. We also investigated the effects of restructuring edge loops to use a compressed sparse row (CSR) type representation to improve register utilization and to reduce the number of memory operations.

Reordering Edges

We reorder the list of edges so that all of the edges incident on a node are listed consecutively. The edges incident on node 1 are listed first, followed by edges incident on node 2 and so on. We avoid listing an edge (i, j) twice by associating it with node i, if $i < j$, or with node j if $j < i$. The advantage of this reordering is that, for all edges associated with a node i, the data for node i remains in the cache, giving a better cache utilization.

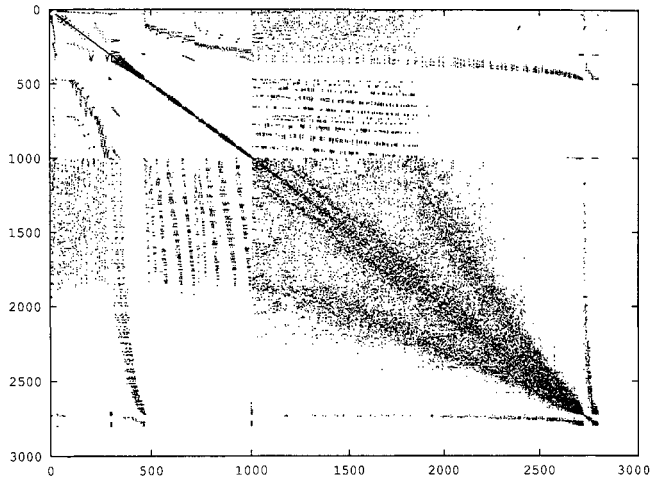

Fig. 2 Initial data access pattern of 2800-node mesh.

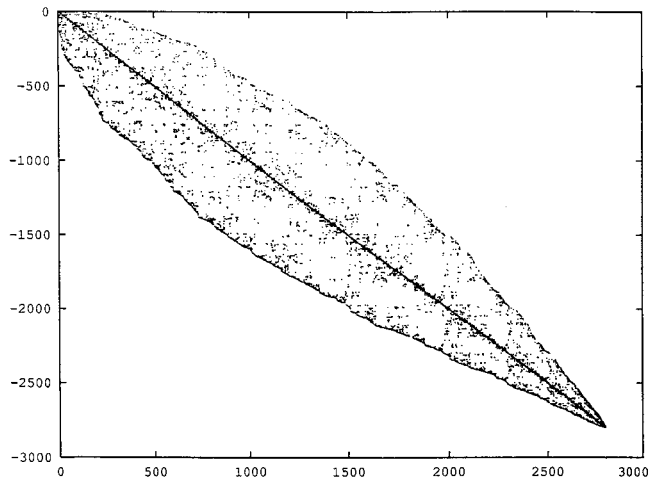

Fig. 3 Data access pattern of 2800-node mesh after RCM reordering.

Loop Restructuring

We can restructure edge loops to use a CSR-type format.[13] When we reorder edges, we list all edges incident on a node consecutively. We can take this a step further by compressing the data structure that represents the list of edges. An edge represents an association between two vertices (i, j). Once we have ordered the edges in the manner described above, we can simply generate an array IA that lists the consecutive values of j for each vertex i of the entire mesh. We also construct a separate pointer array JA, which indicates the beginning and ending location of the list associated with vertex i. Thus $IA[JA(i)],\ldots, IA[JA(i+1)-1]$ represent the edges associated with node i.

Node Reordering

The numbering of mesh vertices can have an important effect on the pattern of data access. We seek to number mesh vertices so that data associated with vertices linked by mesh edges tend to be stored in nearby memory locations. There is no reason to expect that mesh orderings produced by mesh generators should have this property. For instance, Fig. 2 depicts the data access pattern for the 2800-node mesh shown in Fig. 4. The X axis and the Y axis in the figure give node numbers, and each point represents an edge between the corresponding vertices on the X and Y axes.

We expect that the highly nonuniform data accesses shown in Fig. 2 will cause poor cache utilization and high virtual memory management overheads. To reduce these overheads, we reorder the vertices using the reverse Cuthill McKee (RCM) method.[14] The RCM reordering method is frequently used by researchers in the area of sparse matrix computation. It is a graph-based technique to reorder columns of a sparse matrix to reduce its bandwidth. The resulting data access pattern after RCM reordering is shown in Fig. 3.

In comparison to Fig. 2, the data access pattern of Fig. 3 shows much less irregularity and therefore should improve cache utilization and reduce virtual memory management overheads.

Single Processor Results

The aforementioned reordering methods were applied to several meshes to study single node performance on the iPSC/860. The results of our experimentation are summarized in Table 1. Meshes M1 and M2 are "regular" meshes that are represented using our unstructured mesh data structures (i.e., tetrahedral meshes derived from the subdivision of a structured hexahedral mesh), and mesh M3 is the smallest of the unstructured meshes used by the multi-

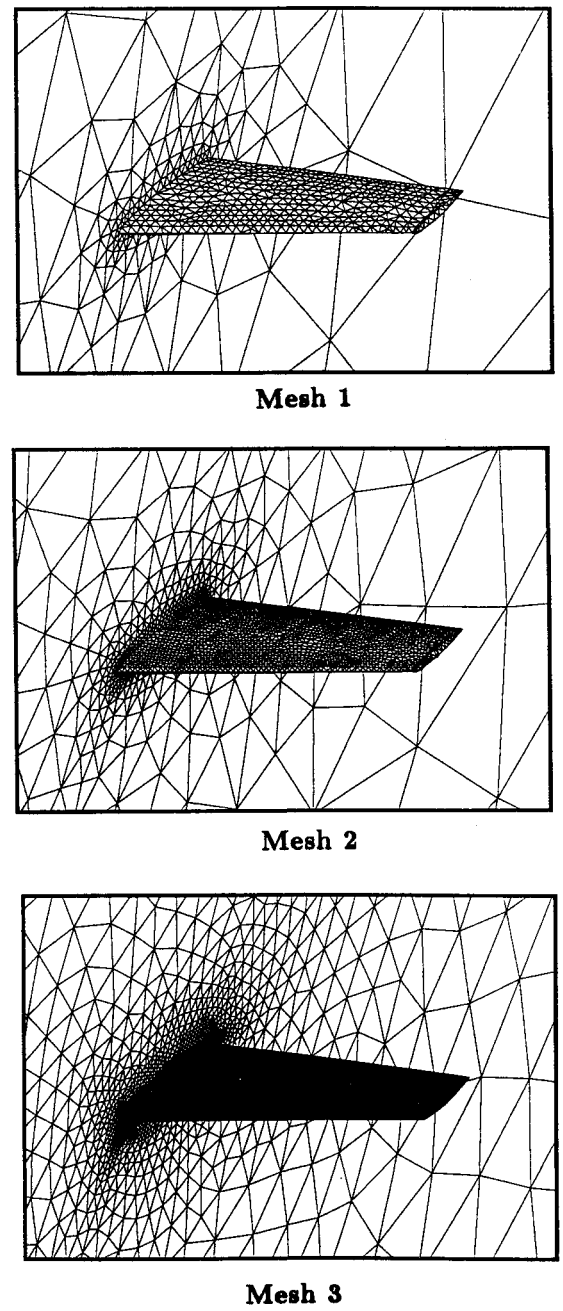

Fig. 4 Sequence of global coarse and fine meshes employed for computing inviscid transonic flow over the ONERA M6 wing; mesh 1: 2,800 nodes, 13,576 tetrahedra, 2,004 boundary faces; mesh 2: 9,428 nodes, 47,504 tetrahedra, 5,864 boundary faces; mesh 3: 53,961 nodes, 287,962 tetrahedra, 23,108 boundary faces; and mesh 4: 357,900 nodes, 2,000,034 tetrahedra, 91,882 boundary faces (not shown).

grid algorithm to solve the transonic test case over the ONERA M6 wing.

Column one in Table 1 gives the number of vertices and identification of the mesh used. The Mflops obtained without any reordering by the unstructured Euler solver are shown in column 2. The next four columns show the improvement in Mflops due to various combinations of three reordering schemes described earlier. The results indicate that edge reordering alone gives significant improvement in performance for all of the meshes. Node reordering in conjunction with edge reordering has more impact on mesh M3 in comparison with the two other meshes. This is probably due to the differences in the ways in which the meshes were generated. The loop transformation does not have much impact on performance (column 3 vs column 5 and column 4 vs column 6) because use of a cache instead of registers is not very costly. From these experimental results we conclude that, by reordering, the performance on a single node of the iPSC/860 can be improved almost by a factor of 2 for the types of meshes used in unstructured applications. Since loop restructuring was found to yield only marginal benefits, in the interest of preserving the original structure of the sequential code, only edge reordering and node reordering were employed in all subsequent implementations.

VI. Results

We describe the results of a number of experiments we have carried out to evaluate the performance impact of our optimizations. These experiments were carried out on an Intel iPSC/860 hypercube and the Intel Touchstone Delta. For purposes of comparison, we cite performance numbers obtained from an optimized Cray Y-MP version of this code.[1]

A standard transonic test case is chosen for this comparison, namely, a Mach 0.84 flow over an ONERA M6 wing at 3.06-deg incidence. The sequence of meshes employed for the multigrid algorithm in this case is depicted in Fig. 4. The coarsest grid contains merely 2800 vertices, whereas the finest grid contains a total of 357,900 vertices and just over 2 million tetrahedra. The intermediate grids have 9428 and 53,961 vertices, respectively. The computed Mach contours of the obtained solution are depicted in Fig. 5, where the familiar double-shock pattern is observed.

Table 1 Performance of reordering on single node of iPSC/860 in Mflops

		Edge reordering		Loop restructuring	
Mesh size	Original	Original node ordering	Node reordering	Original node ordering	Node reordering
545 (M1)	4.03	5.90	6.15	6.53	6.74
3681 (M2)	4.09	6.13	6.27	6.42	6.74
2800 (M3)	2.76	4.28	5.42	4.35	5.76

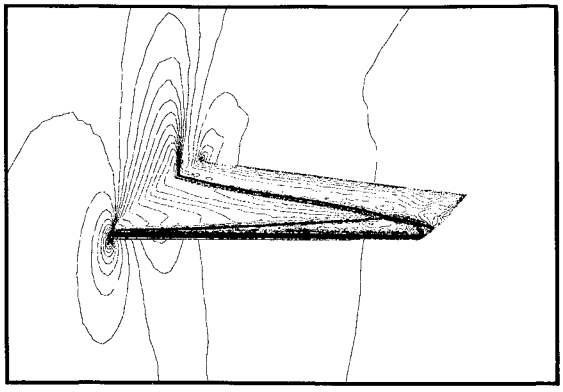

Fig. 5 Computed Mach contours on the finest mesh of the multigrid sequence.

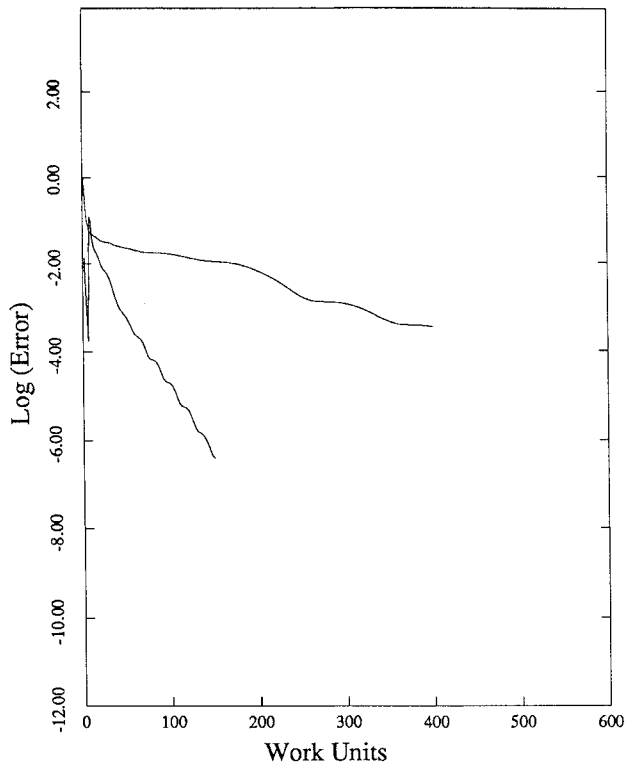

Fig. 6 Comparison of the multigrid convergence rate and the single-grid converge rate on the finest grid of the sequence about the ONERA M6 wing as measured by the average density residuals vs the number of work units.

We employ a single-grid algorithm along with two versions of multigrid algorithms. The two versions of multigrid are W- and V-cycle algorithms. The V-cycle multigrid algorithm visits all meshes an equal number of times within a single cycle, whereas the W-cycle visits coarse meshes more frequently than fine meshes. Details of these algorithms can be found in Ref. 1. W-cycle multigrid strategies require slightly more work per cycle (on the order of 15–25% depending on mesh size) but often converge slightly more rapidly and are thus more efficient overall. However, the relative merits of W- vs V-cycle strategies can be very case dependent. On the other hand, both strategies always offer large increases in efficiency over single-grid explicit methods. Figure 6 provides a performance comparison between the single grid and W-cycle multigrid code by plotting convergence histories in terms of work units for the solution of the flow over the ONERA M6 wing at the previously prescribed conditions on the 357K mesh. In this plot, a work unit is defined as the time required for a single- grid explicit cycle. As can be seen in this figure, the W-cycle multigrid algorithm converges over 6 orders of magnitude in roughly 150 work units, which corresponds to 100 multigrid cycles, whereas the single-grid calculation is seen to require almost an order of magnitude more work to reach the same level of convergence.

On the Cray Y-MP, the single-grid code for this case required a total of 33 MW of memory and ran at a speed of 19 s/cycle on a single processor. The W-cycle multigrid code required a total of 42 MW of memory and ran at a speed of 34 s/multigrid cycle. For both cases, the computational rates achieved were about 100 Mflops. For the multigrid run, engineering solutions (3 to 4 orders of convergence) for this case could thus be obtained in roughly 30 min of Cray Y-MP single processor CPU time.

We employed the recursive spectral partitioning algorithm to carry out partitioning.[15,16] Williams[17] compared this algorithm with binary dissection[18] and simulated annealing methods for partitioning two-dimensional unstructured mesh calculations. He found that recursive spectral partitioning produced better partitions than binary dissection. Simulated annealing in some cases pro-

Table 2 Performance in Mflops of unstructured mesh code on 128 processor iPSC/860

Method	Explicit 53K mesh	Explicit 357K mesh	Multigrid 357K mesh V cycle	Multigrid 357K mesh W cycle
No communication opt., original mesh ordering	85	127	105	92
No communication opt., reordered mesh	88	149	120	100
Communication opt., original mesh ordering	172	217	181	153
Communication opt., reordered mesh	216	356	298	244

Table 3 Communication, total time per cycle(s); unstructure mesh code 128 processor iPSC/860 357K mesh

Method	Explicit Comm	Explicit Total	Multigrid W cycle Comm Intramesh	Multigrid W cycle Comm Intermesh	Multigrid W cycle Total
No communication opt., original mesh ordering	7.6	14.1	30.1	1.7	40.1
No communication opt., reordered mesh	7.6	12.0	29.4	1.6	35.8
Communication opt., original mesh ordering	1.8	8.2	9.6	0.5	22.7
Communication opt., reordered mesh	1.8	5.0	9.4	0.4	14.6

duced better partitions, but the overhead for simulated annealing proved to be prohibitive even for the relatively small meshes employed (the largest had 5772 elements). Venkatakrishnan et al.[7] and Simon[16] also reported favorable results with this partitioner. We carried out preliminary performance comparisons between binary dissection and the recursive spectral partitioning and found that recursive spectral partitioning gave superior results on the iPSC/860 on our three-dimensional meshes. The results we report all have been obtained using recursive spectral partitioning to partition all meshes. Partitioning was performed on a sequential machine as a preprocessing operation. In all of the experimental studies reported in this paper, we use the same optimized version of the communications kernels that employed forced message types, nonblocking receives (irecv), and that employ the heuristic of Ref. 7 to determine the order in which messages are sent.

Table 2 examines the effects of the communication optimizations and the reordering optimizations. The table depicts the Mflops obtained with the single grid, and the W- and V-cycle multigrid algorithms on a 357K node fine mesh. The figure also depicts the performance of the single-grid method on the second finest grid (53,961 vertices). Measurements were performed on a 128 processor Intel iPSC/860. We achieved roughly a factor of 2.5 to 3 improvement in the computational rate from the use of our optimizations. For instance, consider the single mesh code on the 357K mesh. When we employed both optimizations, we saw a computational rate of 356 Mflops. Communications optimizations without reordering yielded 217 Mflops, and reordering without communications optimizations yielded 149 Mflops. When we employed neither optimization, we achieved 127 Mflops. Analogous improvements are seen in the multigrid codes.

The multigrid V- and W-cycle algorithms achieved 298 and 244 Mflops, respectively, when we employed both communication optimizations and reordering. The frequent visits to coarse meshes, interpolation, and prolongation in the W-cycle multigrid might be expected to lead to a significant degradation in computational rate. A degradation of roughly 32% compared with the single-grid code was, in fact, observed, but substantially larger degradations are seen when we leave out either our communication optimizations or reordering.

Table 3 gives information on the effects of the different optimizations on communication and computation time. For the single-mesh code, use of the communications optimizations led to a four-fold reduction in time spent on communication. For the W-cycle multigrid, we have broken down communication time into 1) communication time required to carry out calculations on one of the four individual meshes in the multigrid calculation (intramesh communication in Table 3), and 2) communication time required for transferring data between meshes (intermesh communication Table 3). We note that the cost of carrying out intramesh communication is over an order of magnitude higher than the cost of carrying out intermesh communication. This indicates that we do not appear to be making a significant performance compromise by independently partitioning the meshes in the multigrid algorithms.

The final test case involves the computation of a highly resolved flow over a three-dimensional aircraft configuration. The mesh contains a total of 804,056 points and approximately 4.5 million tetrahedra. This is believed to be the largest unstructured grid problem attempted to date. In Fig. 7, we depict the second mesh of the proposed multigrid sequence (we do not show the 804K mesh due to printing and resolution limitations). For this case, the freestream Mach number is 0.768 and the incidence is 1.16 deg. The computed Mach contours are shown in Fig. 8, where good resolution of the shock on the wing is observed. The single-grid computational rate for this case achieved 778 Mflops on 256 processors of the Delta machine, and 1496 Mflops on the full 512 processor configuration of the Delta. The same case was run on the Cray Y-MP/8 machine, using all 8 processors in dedicated mode. The Cray autotasking software was used to parallelize the code for this architecture. Both the single-grid and multigrid codes achieved a computational rate of 750 Mflops on all 8 processors, which corresponds to a speedup of roughly 7.5 over the single-pro-

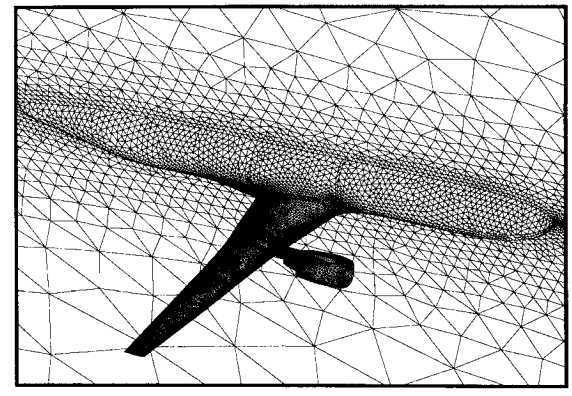

Fig. 7 Coarse unstructured mesh about an aircraft configuration with single nacelle; number of points = 106,064, number of tetrahedra = 575,986 (finesh mesh not shown).

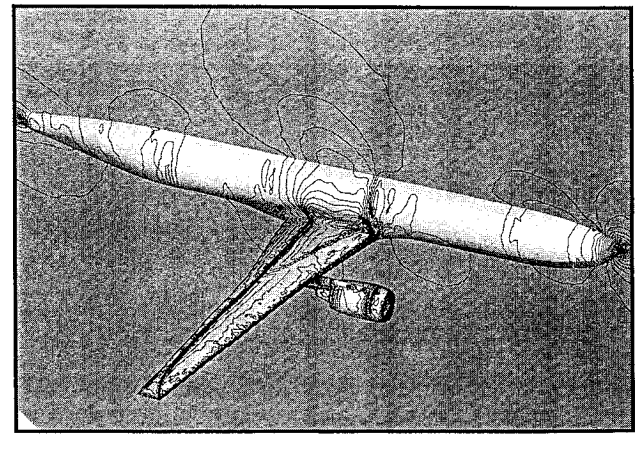

Fig. 8 Mach contours for flow over Aircraft configuration computed on fine mesh of 804,056 vertices and 4,500,000 tetrahedra (Mach = 0.768, incidence = 1.116 deg).

Table 4 Computational rates (Mflops) unstructured mesh code iPSC/860 and Delta; incremental scheduling, blocked and reordered mesh

Method	Explicit 357K mesh	Multigrid 357K mesh	
		V cycle	W cycle
128 processor iPSC/860	356	298	244
128 processor Delta	408	320	267
256 processor Delta	646	516	412
1 processor Y/MP	103	100	100

cessor performance. A residual reduction of 6 orders of magnitude was obtained in 100 multigrid W cycles that required 16 min on the full Cray Y-MP/8. On the 512 processor Delta, the W-cycle multigrid algorithm resulted in a degradation of the computational rate by about 30%, achieving 1.03 Gflops. However, this degradation is far outweighed by the increase in convergence speed. Thus, a solution involving 100 multigrid W cycles could be obtained on the full Delta machine in just under 10 min.

In Table 4 we depict the computational rates achieved on different architectures for the single-mesh solution procedure, along with V- and W-cycle multigrid solution procedures, for the two cases previously described. We employed all of our optimizations in these tests. As expected, for the Cray Y-MP, single-processor rates are relatively insensitive to the problem size and the solution strategy. On the iPSC/860 and the Delta, the computational rates are calculated by counting the number of floating point operations performed. On the Cray Y-MP, the system facility called hpm was utilized to measure the rates. If we scale the computational rate given by the Cray Y-MP facility, to get the rate for the Intel machine, the scaling factor being the ratio of the time per cycle on the Cray Y-MP to the corresponding value on the Intel machine, we find the rate is about an average of 40 Mflops more than that presented for the iPSC/860 and the Delta. On the iPSC/860 and the Delta architectures, maximum computational efficiency is achieved for large meshes using the single-grid solution strategy. Thus the single-grid run of the 804K grid on 512 Delta processors achieves twice the computational rate of the Cray Y-MP/8, or 15 times the rate of a single Cray Y-MP processor. Similarly, the 357K mesh achieved about 7 times the performance of a single Cray Y-MP processor for a single-grid run (roughly equivalent to the full Cray Y-MP /8 performance) and 4 times the single processor Cray Y-MP performance for a W-cycle multigrid run (or about 60% of the Cray Y-MP/8 performance). Since the overall solution efficiency of the multigrid strategy is much higher than that of the single-grid explicit scheme, this emphasizes the need to use overall solution time as a measure of solution efficiency rather than simply computational rates.

VII. Conclusions

A number of earlier reports have noted that two-dimensional, explicit unstructured mesh solvers appear to be well suited for distributed memory multiprocessors. Although explicit schemes may be easily parallelizable, they are not numerically efficient. We have developed a distributed memory version of an efficient three-dimensional unstructured multigrid code. We have shown that competitive computational rates can be achieved for this problem on massively parallel distributed memory architectures. An approximately threefold performance improvement was obtained by optimizations that reduced communication overhead and decreased the computation time required by the processors.

The encapsulation of the communications optimizations into a set of software primitives eases the implementation of a wide range of similar problems and the porting to different architectures. For example, in Ref. 19, Das and Saltz report on our use of the PARTI run-time support in porting the molecular dynamics code CHARMM to the iPSC/860 and Delta multiprocessors. At the time of writing, PARTI has been ported to the CM-5 and to networks of workstations. PARTI is also designed to be used as part of the run-time support for distributed memory compilers, and PARTI has already been used in the implementation of several different Fortran-based prototype distributed memory compilers. The simultaneous use of an efficient multigrid algorithm and massive parallelism results in rapid solution times for large problems.

Our approach is designed to facilitate the development of parallelized adaptive meshing strategies. We have expanded the set of PARTI software to include primitives that support problem remapping (this superset of PARTI is called DYBBUK).[4,20] Although we have not yet implemented a self-unstructured adaptive Euler or Navier-Stokes solver using PARTI/DYBBUK, we have demonstrated a capability in this area by using DYBBUK to develop an adaptive iPSC/860 and Delta implementation of the CHARMM molecular dynamics code. Once this functionality is developed, we will be able to implement adaptive three-dimensional multigrid codes on distributed architectures.

Acknowledgment

This research was supported by NASA under Contract NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering at NASA Langley Research Center. Support from ARPA NAG-1-1485 is also acknowledged. We would like to thank Horst Simon for providing us with his recursive spectral partitioner and Rob Vermeland and Cray Research, Inc., for providing dedicated time on the Cray Y-MP/8 machine. We would also like to acknowledge the National Institutes of Health and NAS at NASA Ames for allowing us access to their 128 processor iPSC/860. This research was performed in part using the Intel Touchstone Delta system operated by Caltech on behalf of the Concurrent Supercomputer Consortium. Access to this facility was provided by NASA. We would like to thank Bob Voigt, Yousuff Hussaini, and Manuel Salas for their encouragement and support over the course of this project.

References

[1]Mavriplis, D. J., "Three Dimensional Multigrid for the Euler Equations," *AIAA Journal*, Vol. 30, No. 7, 1992, pp. 1753–1761.

[2]Brezany, P., Gerndt, M., Sipkova, V., and Zima, H. P., "(SUPERB) Support for Irregular Scientific Computations," *Proceedings of the Scalable High Performance Computing Conference '92*, IEEE Press, Los Alamitos, CA, April 1992, pp. 314–321.

[3]Saltz, J., Berryman, H., and Wu, J., "Multiprocessors and Run-Time Compilation," *Concurrency: Practice and Experience*, Vol. 3, No. 6, 1991, pp. 573–592.

[4]Ponnusamy, R., Saltz, J., and Choudhary, A., "Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse," *Supercomputing 93*, Nov. 1993, pp. 361–370.

[5]Berryman, H., Saltz, J., and Scroggs, J., "Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Architectures," *Concurrency: Practice and Experience*, Vol. 3, No. 3, 1991, pp. 159–178.

[6]Hammond, S., and Barth, T., "An Optimal Massively Parallel Euler Solver for Unstructured Grids," AIAA Paper 91-0441, Jan. 1991; also *AIAA Journal*, Vol. 30, No. 4, 1992, pp. 947–952.

[7]Venkatakrishnan, V., Simon, H. D., and Barth, T. J., "A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids," NAS Systems Division, NASA Ames Research Center, Rept. RNR-91-024, Sept. 1991; also *Journal of Supercomputing*, Vol. 6, No. 2, 1992, pp. 117–127.

[8]Williams, R., "DIME—Distributed Irregular Mesh Environment: User's Manual," California Inst. of Technology, Rept. C3P 861, Cal Tech, Pasadena, CA, Feb. 1990.

[9]De Keyser, J., and Roose, D., "Adaptive Irregular Multiple Grids on a Distributed Memory Multiprocessor," *Proceedings of the 2nd European Distributed Memory Computing Conference*, edited by A. Bode, Springer-Verlag, Berlin, 1991, pp. 153–162.

[10]Jameson, A., Baker, T. J., and Weatherhill, N. P., "Calculation of Inviscid Transonic Flow over a Complete Aircraft," AIAA Paper 86-0103, Jan. 1986.

[11]Saltz, J., Berryman, H., and Wu, J., "Runtime Compilation for Multiprocessors," *Concurrency: Practice and Experience*, Vol. 3, No. 6, 1991, pp. 573–592.

[12]Margulis, N., *i860 Microprocessor Architecture*, McGraw-Hill, New York, 1990, pp. 56–59.

[13]Saad, Y., "Sparsekit: A Basic Tool Kit for Sparse Matrix Computations," *Research Inst. for Advanced Computer Science*, Rept. 90-20, NASA Ames Research Center, MS T045, Moffett Field, CA, 1990.

[14]Cuthill, E. H., and Mckee, J., "Reducing Bandwidth of Sparse Symmetric Matrices," *Proceedings of the ACM 24th National Conference*, Associa-

tion for Computing Machinery, New York, 1969, pp. 157–172.
[15]Pothen, A., Simon, H. D., and Liou., K. P., "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM Journal of Mathematical Analytical Applications*, Vol. 11, 1990, pp. 430–452.
[16]Simon, H., "Partitioning of Unstructured Mesh Problems for Parallel Processing," *Computing Systems in Engineering*, Vol. 2, No. 2/3, 1991, pp. 135–148.
[17]Williams, R., "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations," *Concurrency, Practice and Experience*, Vol. 3, No. 5, 1991, pp. 457–482.
[18]Berger, M. J., and Bokhari, S. H., "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Transactions on Computers*, Vol. C-36, No. 5, 1987, pp. 570–580.
[19]Das, R., and Saltz. J., "Parallelizing Molecular Dynamics Codes Using the (Parti) Software," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Vol. 1, SIAM, Norfolk, VA, 1993, pp. 187–192.
[20]Ponnusamy, R., Das, R., Saltz, J., and Mavriplis, D., "The Dybbuk Runtime System," *IEEE COMPCON* (San Francisco, CA), IEEE Press, Los Alamitos, CA, 1993, pp. 205–214.