Parallelizing a Real Application Code: Is It Worth It?

Ed Felten, Paul Messina and Roy Williams

Concurrent Supercomputing Facility

California Institute of Technology

Pasadena, CA 91125

Abstract

We have studied issues involved in porting a real application to a distributed memory machine while attempting to keep the code efficient and portable. MACH2 is a magnetohydrodynamic simulation code written with about 20000 lines of FORTRAN. Moderate parallelism may be extracted by domain-decomposing the algorithm, but the existing algorithm is not suitable for a massively parallel machine. This is partly because the code is actually used with relatively coarse meshes; and partly because the code requires a large equation of state database, and we discuss parallel implementations of this. We also discuss algorithmic changes to exploit parallelism, such as parallel multigrid and parallel operator splitting techniques. We have made a small version of MACH2, which runs on distributed memory machines; with an automatic load-balancer, and a mesh-generator and optimizer. We present timings of this small version.

Introduction

Programming a supercomputer is not as easy as it used to be. To get maximum performance the programmer must pay careful attention to vectorizing, memory contentions, communication overhead, cache management and other such issues. In addition the fastest parallel algorithm may be different at the highest conceptual level from the fastest sequential one[1,2]. It seems more and more difficult to imagine writing code which is both efficient and portable across a wide range of machines, and has clean readable code. It is even harder to achieve these objectives by making only small changes to an existing code.

The simplest, easiest and most popular way to speed up a code is vectorization: if the existing code happens to use long vectors which are always accessed consecutively, then a vectorizing compiler gives excellent speedup with little human effort. Optimizing and vectorizing compilers are improving, and we can expect some further increases in the ability of these ‘‘automatic parallelizers’’.

Looking further into the future, we see that vectorization cannot continue to provide increasing performance. The programming model is still restricted to a sequential flow of control with coprocessor slaves. An army would be most inefficient with a single general and 2000 ordinary soldiers, unless it were doing an unusually parallel task, such as digging 200 holes in a large field. After vectorization, or if vectorization is not possible, increased performance can come from algorithm development, keeping a task and its data together, and by discovering more inherent parallelism in the existing algorithm.

We should mention in passing that the number of processors in a machine must be commensurate with the size of the task. It would not be worth employing 200 soldiers to dig a single small hole. Furthermore the task to be done must be capable of division into many subtasks or the parallel machine is useless.

The approaches mentioned above are best done by a human, which is expensive and introduces bugs. Before embarking on a complete rewrite, it may be well to ask what kind of performance one might expect and how much of the original code can be salvaged. We have investigated a large scientific application written for the Air Force Weapons Laboratory, to try to discover how much parallelism could be usefully exploited.

MACH2 is a two-dimensional magnetohydrodynamic simulation code for complex experimental configurations[3]. The first problem with porting this code is that it uses CRAY FORTRAN extensions extensively, though some of these may be incorporated into ANSI 8X FORTRAN when it arrives. The code uses pointers, dynamic memory allocation and complex EQUIVALENCE statements in order to recycle temporary memory space. Of course it would be desirable to keep the code in FORTRAN because salvaging the original code would be easier, and because the user community is accustomed to FORTRAN.

In our view it would be more difficult to change the existing code for portability and parallelism than to completely rewrite large sections in a higher-level language with parallelism in mind. In that case the pointers and dynamic memory are built into the language, and constructs such as data-structures allow clear user-friendly code.

We shall discuss the structure of the algorithm as far as it impacts parallelism. We shall discuss domain-decomposition as a way to utilize moderate parallelism (10-20 processors), and some parallel multigrid methods and parallel operator splitting methods which may afford a further factor of about 4. We shall also discuss a preprocessor for load-balancing the computation and some possible difficulties with a large equation of state database which the program needs. We present some efficiency results for Machpar, which is a small version of MACH2 which we hope captures the essence of MACH2 at least as far as parallelization goes.

The program MACH2

MACH2 is a two-dimensional time-accurate solver of a set of conservation laws representing ideal MHD, together with resistive diffusion of magnetic field and consequent joule heating, thermal diffusion, and radiation cooling. The computational mesh moves as the computation progresses, both to follow the moving fluid as a Lagrangian mesh, and also to improve the resolution of detail in places of the domain which need it.

Although the mesh vertices move, the logical or topological structure does not change. This

Figure 1:
Top, a physical domain broken into blocks suitable for meshing. Bottom, the domain filled with equilibrated mesh. Each mesh point is at the average position of its four nearest neighbors.

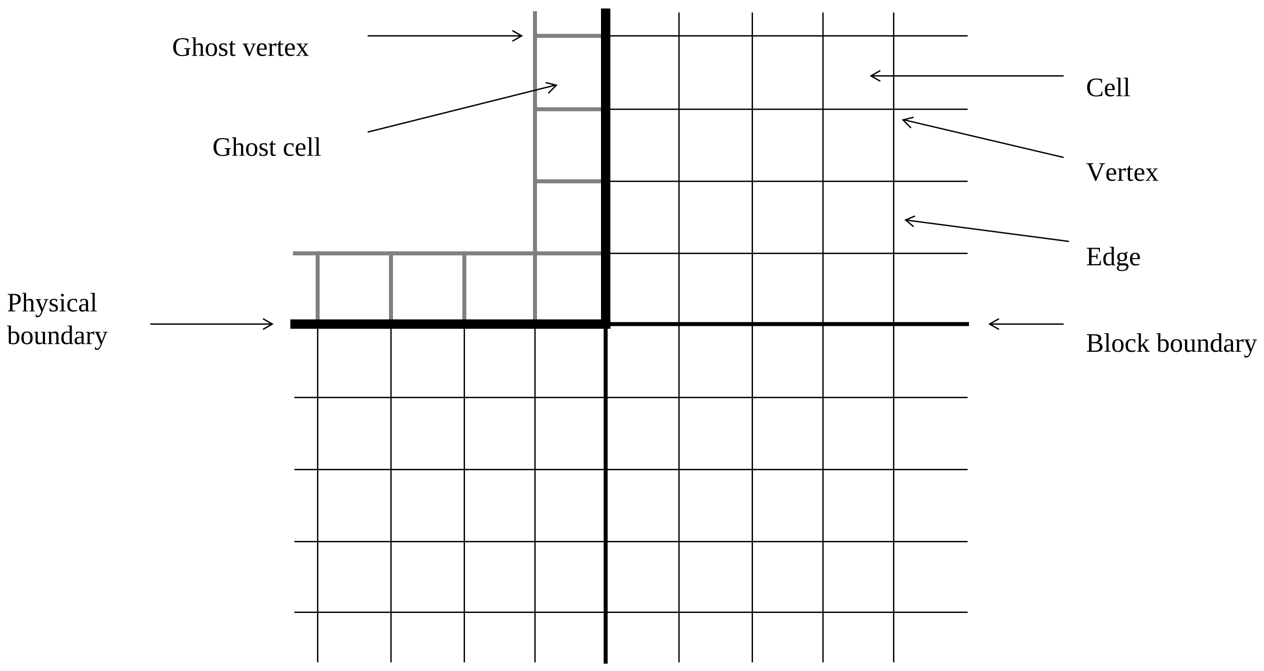

Figure 2
Structure of current MACH2 code, with three blocks meeting at a corner. The vertices need not lie on a grid like this, but may be distorted to fit the problem geometry.

structure is that of a connected set of blocks, each filled with rectangular mesh. An edge of a block may represent either physical boundary such as the wall of the vessel containing the plasma, or it may meet with the edge of another block. When two blocks meet like this they must meet completely, so the edges must be the same length, both physically and in terms of the number of mesh points. Figure 1 shows the physical boundary of a problem domain split into blocks, and a mesh filling the domain. Each block is logically a rectangular mesh consisting of vertices connected by edges, with cells in the interstices. Data may be stored at vertices, at edges, or in the cells.

Boundary conditions are implemented by means of ‘‘ghost vertices’’, ‘‘ghost edges’’ and ‘‘ghost cells’’, which lie outside the physical domain. Figure 2 shows three blocks meeting and the associated ghost vertices and cells. The purpose of ghost data is to set boundary conditions: for example if a Neumann boundary condition is required with zero slope, then the ghost vertex value of the field is set to be the same as the interior value. In this way each vertex (or cell or edge), whether in the interior or at the physical boundary, has a full complement of neighbors and can be treated as interior data. It turns out that even the complex boundary conditions required by a code such as MACH2 can be farmed off in this way into ghost data manipulation, leaving the actual boundary data to be computed in exactly the same way as the interior data.

Domain Decomposition

Often an appropriate way to parallelize a physical simulation is domain decomposition[1], where the physical domain of the problem is partitioned into roughly equal sized sub-domains, and each processor of the machine takes charge of a sub-domain. This works well because the laws of nature can be expressed locally, meaning that each processor need only know about conditions near the boundary of its sub-domain to compute conditions in the interior of the sub-domain. In MACH2 these sub-domains can be constructed from one or more of the blocks which fortunately already exist in the architecture of the sequential code.

Suppose each processor manages a collection of the blocks which constitute the problem domain. The locality of the physical laws means that the processor only needs to know field values within and at the boundary of its blocks, and also a band one grid point away from the boundary. This one-gridpoint-distant value corresponds to an internal grid point of another processor. There seem to be three possibilities here.

1. Shared memory environment: the processor could simply read the field value from the unique global data address at which the value is stored. However, one or more other processors may be simultaneously reading or writing to that data address. Reads are generally not atomic: if the read operation consists of more than a few bytes, the first bytes may be read, followed by a write to that address, followed by the next part of the read., and as a result the read data may be nonsense. We would need a lock to avoid read/write conflicts. In addition, the order in which the computation occurs depends critically on the slight differences in clock speeds of the processors, which can lead to unpredictable and grotesque bugs.

The second and third possibilities for the architecture of a parallel MACH2 take place in a distributed memory environment. An advantage of programming for a distributed memory machine *ab initio* is it is trivial to port to a shared memory (or sequential) machine simply by partitioning the global memory. Also there can be no memory hot spots because the distributed-memory programmer is forced to avoid them.

2. Asynchronous distributed memory: If each processor keeps the data only for its own blocks, it could send a message whenever it needs off-processor data. This message would interrupt another processor which owns the requested data, and the other processor would send back the requested data then resume. As in the shared memory case, locks are required. The execution of the code depends critically on processor clock speeds, and contention problems arise. Also, just as with humans working together, it is best to avoid continual asynchronous interruptions as this can be very inefficient.

3. Loosely synchronous distributed memory: The third method of parallelizing MACH2 would be where each processor keeps all the data it needs in its own memory, including the border of grid points one unit away from its boundary. Some data is stored redundantly; there may be two or more processors keeping what should be the same data value. This feature is useful for debugging: if two values representing the same data point in different processors are in fact different, it is relatively easy to find out when the difference first occurs.

This redundancy fits naturally with the idea of ghost vertices. Instead of only having ghost structures at the physical boundaries of the computational domain, we put them at the boundaries

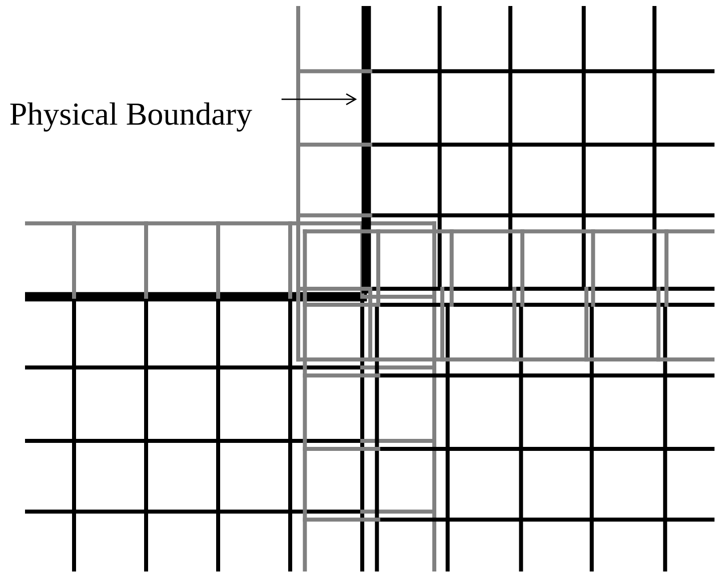

Figure 3
Addition of ghost cells to the each block instead of only at physical boundaries naturally facilitates loosely synchronous parallel operation.

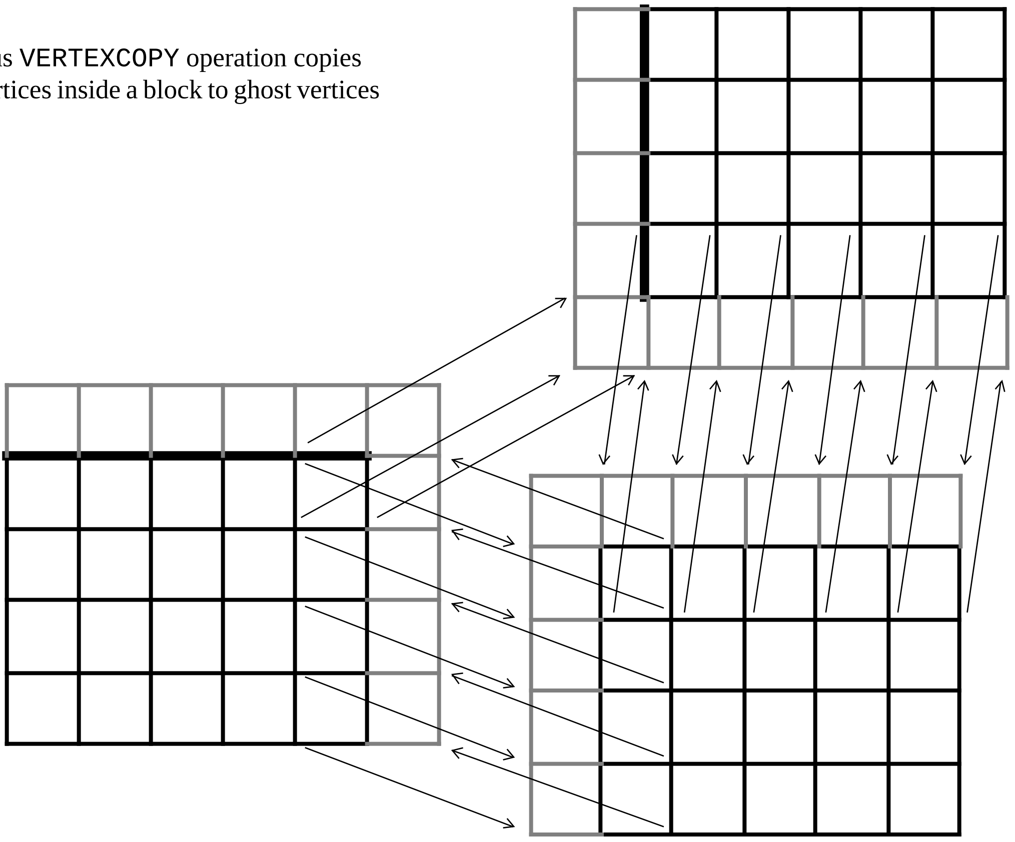

Figure 4
Loosely synchronous `VERTEXCOPY` operation copies data from master vertices inside a block to ghost vertices of other blocks.

of each block, as shown in Figure 3.

We propose to communicate vertex data between blocks by a subroutine call `VERTEXCOPY`, illustrated in Figure 4. Data from the internal vertices of a block is passed to the processor which owns the corresponding ghost vertices, and that ghost data is overwritten with the ‘‘master’’ data. If a processor sends before its neighbor is ready to receive, the sender waits, and conversely if the receiver is ready before the sender, the receiver waits. In this way a natural synchrony is imposed on the calculation.

A simple example would be the parallel solution of Laplaces’ equation with a nine-point stencil, by the Jacobi relaxation method. Each field-value is replaced by a weighted average of the field-values of the eight surrounding vertices, and this is done for all internal and boundary vertices. The ghost vertex values are not changed until the `VERTEXCOPY` operation at the end of each sweep. If one processor has many more vertices to update than the others, the blocking read/writes will force all the others to wait for it, so that the calculation proceeds at the rate of the slowest processor.

Notice that `VERTEXCOPY` does not exchange the vertices at the block boundary. With the Laplace example, all the data surrounding a boundary vertex is the same for both representations of the vertex, so each processor will compute the same result for the updated value, and communication is not required.

MACH2 uses cells and edges as well as vertices, so similar loosely synchronous subroutines can be made to communicate these data across block boundaries. Figure 5 shows the operation of the subroutine `CELLCOPY` for this purpose. An `EDGECOPY` subroutine could be constructed in a similar fashion.

Besides this local communication from the edge of a block to the edge of its neighbor, one more protocol is necessary for global communication, for example if a relaxation is carried out until a global tolerance is reached. We have used the subroutine `combine`, whereby each processor provides one or several data values, and the global sum (or maximum or minimum etc) is put back in place of the data value(s). With the Laplace solver discussed above, the stopping criterion would be evaluated for each block, and a call to `combine` would replace this with the global maximum criterion, and each processor would make the same decision about whether convergence has been attained. As an example:

```
    combine(&tolerance, max, ...);
```

Before the call, each processor has its own local value of the variable `tolerance`, and after the `combine`, each processor has the same global maximum of the local tolerances. The function pointer `max` could be replaced by any other commutative associative binary operation such as `sum` or `product`.

Load Balancing

The workload is divided between processors by assigning different blocks of grid points to different processors. MACH2 already stores the grid as a collection of (logically) rectangular blocks; if there are more processors than blocks, the blocks can be split into several pieces.

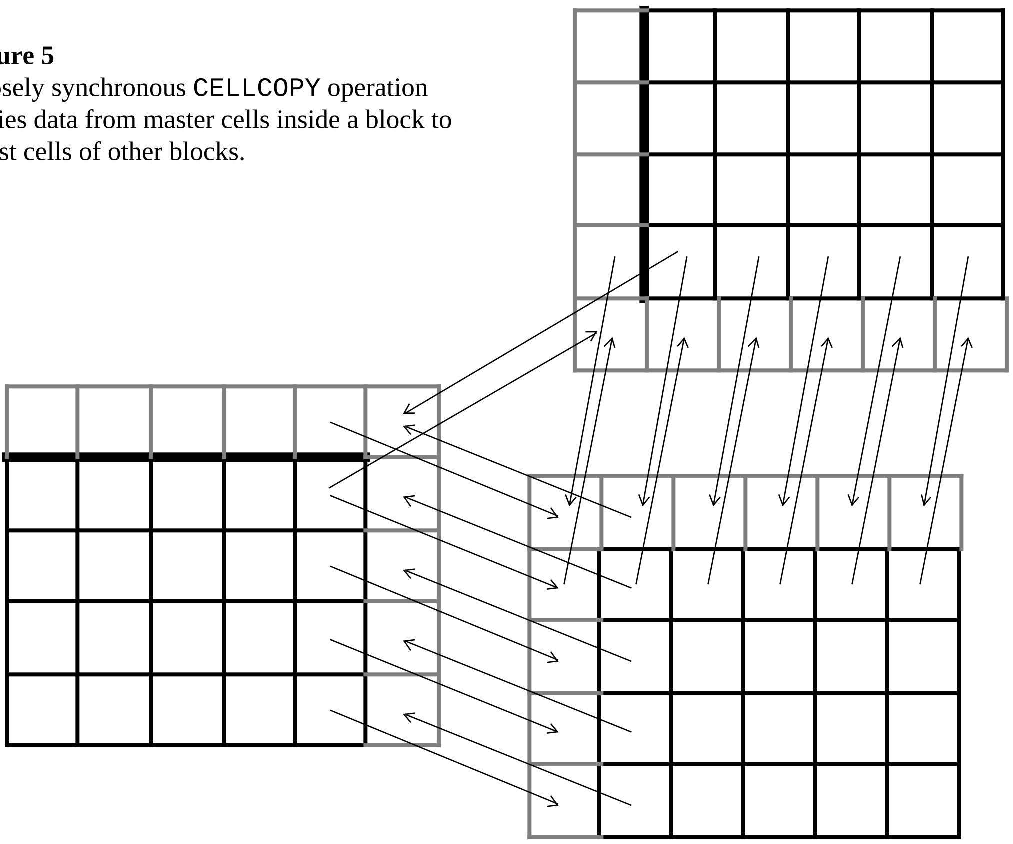

Figure 5
Loosely synchronous `CELLCOPY` operation copies data from master cells inside a block to ghost cells of other blocks.

The load balancer seeks to reduce the total cost of the computation. There are two factors which contribute to the cost: load imbalance and communication overhead. Load imbalance arises when one processor has more work than the others; all must wait while the loaded processor finishes. The load imbalance cost is taken to be the variance of the processor workloads divided by their mean. Communication overhead is the cost of passing edge values from one processor to another. Every boundary between two blocks has a communication cost which is the product of the length *l* of the boundary, and the communication ‘‘distance’’ *d* separating the two processors. This distance depends only on machine architecture. The communication cost is the average communication load per processor.

$$\text{Cost} \;=\; \frac{\langle \text{load}_i^2 \rangle - \langle \text{load}_i \rangle^2}{\langle \text{load}_i \rangle} + C \sum_{\text{edge}} l_{\text{edge}}\, d(\text{proc}_{\text{block1}}, \text{proc}_{\text{block2}})$$

Load balancing is now reduced to a combinatorial optimization problem: assign blocks to processors so as to minimize the cost. This problem is NP-complete and not worth trying to solve exactly. However, excellent solutions can be found by heuristic techniques.

We use an algorithm based on iterative improvement. Iterative improvement consists of searching for a way to move one block from its current location to another processor, lowering the total cost; this process is repeated until there are no more single-block moves which will lower the cost. We start with a random assignment of blocks to processors, then perform iterative improvement. We then carry out a sequence of ‘‘global steps.’’ One global step consists of the following procedures:

1. Move a special, randomly chosen block to a new location.

2. Perform iterative improvement, except that the special block is not allowed to move, until converged.

3. Perform iterative improvement, allowing all blocks to move, until converged.

4. Compare the resulting cost with the cost before step 1, and revert to the original, pre-step-1 cost if it is more favorable.

We continue trying global steps until a certain number (say, 50) of consecutive steps have taken place without changing the configuration.

Our program performs load-balancing as a preprocessing step. The load balancer reads in a file containing a description of block sizes and connections, and outputs another file containing the same information, plus assignments of blocks to processors. It uses the algorithm described above to perform the assignment; there is also an ‘‘outer loop’’ which considers splitting blocks into pieces in order to allow more processors to come into play.

Efficiency of Domain Decomposition

A certain amount of communication overhead is introduced when a task is broken into pieces and done in parallel the efficiency is somewhat less than 1. In the case of domain decomposition, this inefficiency occurs when data is communicated across the boundaries, and it is straightforward to show that when N grid points are divided up amongst n processors the efficiency will be about $1 - C(n/N)^{1/2}$, where C is a constant which is the ratio of communication rate to calculation rate for the particular machine environment. The constant C also depends on the particular problem we are solving. Clearly we would like the number of grid points to be much larger than the number of processors. The main difficulty with parallelizing MACH2 is that there is a great deal of calculation to update each grid point for a time step. This is due to the large number of physical laws controlling the fluid, due to the somewhat complicated mesh-movement algorithm, and due to the interaction between these. It turns out that even a 16x16 grid can run for an hour or more on a CRAY. A grid of this size could reasonably be decomposed into 16 4x4 pieces, and one might hope for a speedup of 8. If one wishes to run the same relatively small problem more quickly, then a massively parallel machine may not be useful, but if the objective is to run larger problems in the same time, say with a 64x64 grid, then a parallel machine would produce a substantial speedup.

We have written a code Machpar based on the principles of domain decomposition as discussed above. Machpar has a preprocessing load-balancer, and solves for the steady flow of a Stokes fluid within a cavity, using the Finite Volume approach as MACH2 does. The code is written in C with the EXPRESS operating system[4], which runs on a variety of distributed memory machines. We have chosen the C language rather than Fortran for this proof-of-principle code because many advanced architecture machines support it better than Fortran and because many features such as dynamic memory allocation, pointers, macros and structures are more naturally expressed in C. Figure 6 shows the mesh of Figure 1 with ghost vertices, and streamlines for flow through this domain. The fluid comes in at the left and out at the right. The times are shown below when

Figure 6:
Top, the domain of Figure 1 with ghost vertices shown. Bottom, streamlines for Stokes flow coming in at the left and out bottom right, using the Finite Volume method.

Machpar is run on an NCUBE/7:

Processors	Solve Time (seconds)
4	23
8	15
16	10

More processors with this small mesh would result in poor efficiency, and there is not enough memory per processor to run this mesh with smaller numbers of processors.

Further Parallelism

The physics which MACH2 simulates is complex, involving magnetic field, current density, fluid velocity, energy density, and fluid density. The evolution of these fields involves several different physical laws, each generating a term in the equations of motion; magnetic and thermal diffusion, magnetic force on the plasma, magnetic field generation, Joule heating, radiation cooling, and convective and viscous effects. Furthermore MACH2 evolves this fluid with a moving mesh, which has its own equations of motion. It would be most desirable to introduce functional parallelism, so that the simulation is not only broken into sub-domains spatially, but also the evolution equation is broken into parts corresponding with a particular physical law. Some tentative steps have been taken towards this Parallel Operator Splitting[5], but these algorithms have hardly been proven even for toy problems. One difficulty is immediately apparent, which is that processor-processor communication must now occur over a whole sub-domain instead of just at boundaries as with domain decomposition, so that for N gridpoints per processor the communication time is order N instead of order $N^{1/2}$ as with domain decomposition. However if the computation time is longer than order N, such as for the solution of a set of linear equations, then the efficiency will still tend to 1 as N becomes large. Even if this loss of efficiency is not catastrophic, deep algorithmic changes would be required to make MACH2 functionally as well as spatially parallel.

MACH2 uses multigrid for the solution of linear equations, which is certainly the most efficient solver known for sequential machines, though at the expense of a more complicated data structure. In parallel the problem is the loss of locality; that we no longer have grid points communicating only with neighboring grid points. In addition the processors are not all used during the coarse-mesh solves, with a corresponding loss of efficiency. Parallel multigrid[2] uses all the processors all the time, but implementing this for MACH2 would involve code changes which are not as straightforward as those discussed above for domain decomposition.

Parallel Database Access

MACH2 simulates real fluids at high pressures and temperatures, when the molecules and atoms of the fluid are being torn apart, and a complex physical processes are competing. All this complexity is wrapped up in the Equation of State of the fluid, and MACH2 uses a table-lookup procedure for this, relating pressure, density and temperature. Unfortunately this database is several megabytes in size and likely to exceed the memory capacity of an individual node in a distributed-memory computer, so we must consider the question of how to store the database and how to manage accesses efficiently.

It is clearly necessary to share one copy of the database between several processors, with each processor containing only part of the database. Several copies of the database may be scattered about the machine; each processor would be assigned to a 'team' which jointly contains an entire copy. When the time comes to look into the database, each processor would look at each of its gridpoints and determine which database entry that gridpoint required and which member of the processor's team contained the desired entry. It would then send requests to all processors from which it desired to receive data; when a request was received it would trigger a database lookup, the results of which would be sent back to the requesting processor. On a hypercube, this can be done loosely synchronously, using the communication system known as the crystal router[1]. All requests and all responses could be lumped together in two crystal router cycles.

Let us assume that a parallel Mach2 is running on a hypercube, and the equation of state database consists of N entries, shared between a team of p processors. Each processor contains g gridpoints. In one ‘‘equation of state cycle’’ each processor will look up g entries in the database (each entry could consist of many bytes).

In the worst case, each gridpoint requires a randomly chosen database entry. The crystal router takes $lg(p)$ steps per cycle (lg is the base-2 logarithm), so the time for a cycle would appear to scale as $g\ lg(p)$. However, the load will be imbalanced since message traffic will not be uniform within the communication system -- some channels will happen to have more traffic than others. These fluctuations will add another factor of $g^{1/2}$, so the total time required will scale as $g^{3/2}lg(p)$. (Note that this is still cheap compared to an explicit solve.)

In practice, there will be locality in the access pattern and this can be used to reduce the lookup time. Locality in time means that a processor will tend to look up the same entry, or nearby entries, in successive cycles. If time locality exists, caching can be used to reduce the number of remote accesses required. Locality in space means that nearby regions of the grid tend to look up the same entry (or nearby entries). In this case, requests can be combined while en route to reduce the total communication traffic. Since we cannot predict in advance how much locality will be available in a real run, we cannot make quantitative predictions except to note that the time per cycle must grow at least as $lg(p)$.

Conclusions

MACH2 is a large, well-maintained, working code, and any step which upsets this equilibrium should be taken with great caution. It would be costly to make significant changes to the code, and only justified if a great benefit accrues. As we have shown above, it is relatively straightforward to domain-decompose the simulation, though of course a great deal of tedious programming and debugging would be required to actually accomplish this. Furthermore it appears that domain-decomposition is not a sufficiently fruitful source of parallelism to justify the effort, in view of the coarse meshes used. Advanced algorithms such as parallel operator splitting and parallel multigrid offer extra parallelism, but at the expense of changing MACH2 at a very deep level. If these efforts could offer a code running say 50 times faster than a CRAY, it would seem worthwhile investing the effort.

The current generation of parallel machines approach the speed of the fastest sequential machines[6], though on particularly well suited problems they can be much faster. In the near future we cannot see this situation changing, though of course it must in the far future. If a completely new version of MACH were to be developed, for example to treat three-dimensional geometry, little extra effort would be required to make it run on distributed or shared memory machines; and this will be an extremely worthwhile investment when parallel machines come of age.

References

1. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Computers*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

2. P. Frederickson and O. A. McBryan, *Intrinsically Parallel Multiscale Algorithms for Hypercubes*, Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications, Pasadena, California, ed. G. C. Fox, January 1988.

3. M. H. Frese, *MACH2: A Two-Dimensional Magnetohydrodynamic Simulation Code for Complex Experimental Configurations*, Air Force Weapons Laboratory, Kirtland AFB, AFWL/PA 87-626.

4. *EXPRESS: An Operating System for Parallel Computers*, ParaSoft Corp., Pasadena, California.

5. R. Glowinski, Private communication.

6. P. Messina, C. Baillie, E. Felten, P. Hipes, W. Pfeiffer, D. W. Walker and R. D. Williams, *Benchmarking Advanced Architecture Computers*, Caltech Concurrent Computation Project report C3P-712