PORTABLE PETAFLOP/S PROGRAMMING: *APPLYING DISTRIBUTED COMPUTING METHODOLOGY TO THE GRID WITHIN A SINGLE MACHINE ROOM*

Paul R. Woodward and S. E. Anderson
Laboratory of Computational Science & Engineering
University of Minnesota

Introduction. The Petaflop/s Programming Challenge

According to today's best projections, petaflop/s computing platforms will combine deep memory hierarchies in both latency and bandwidth with a need for many-thousand-fold parallelism. Initial users of these systems will most likely be asked to meet these challenges to efficient parallel program design armed only with minimal system software: Fortran, C, MPI, and support for POSIX threads or, hopefully, OpenMP on a network node. Unless effective parallel programs are prepared in advance, much of the promise of the first year or two of operation for these systems may be lost. Here we introduce a candidate for a portable petaflop/s programming model that can enable these important early application programs to be developed while at the same time permitting these same applications to run efficiently on the most capable computing systems now available.

Present ASCI Blue platforms as well as collections of machines networked together in NSF supercomputer center machine rooms offer the system software support expected on early petaflop/s systems and they also present the principal challenges of petaflop/s programming. Researchers willing to make the necessary extra efforts may therefore practice petaflop/s programming on these platforms today. These systems can all be viewed as DSM clusters, and hence can, in principle, share the same, portable programming model. An MPI-based model is portable, but its programming paradigm ignores the potential benefits, both for performance and for simplicity of code expression, of hardware support for shared memory within each network node. A threads-based model cannot directly cope with the distributed nature of the memory over the network. Combining both models to get the best of each where appropriate places a large programming burden on even an expert code developer. Therefore a new, portable programming model is needed.

Hierarchical Shared Memory

For a small set of targeted, representative applications and computing platforms, we are implementing a shared memory model, with supporting system software which is in turn built upon an MPI message passing layer. Because the programming model is based on shared memory concepts, our message passing layer must include "put" and "get" primitives. Because of the relatively huge message latencies, and because striping of communications channels introduces a dependence of bandwidth on message size, our "put" and "get" primitives must be tuned for multi-megabyte messages.

As we discuss in more detail below, we have chosen the shared memory programming model because it dramatically simplifies the expression of dynamic load balancing strategies for irregular algorithms. The principal strategy thus enabled is a transparent self-scheduled list of tasks performed in parallel so long as specified data-dependent conditions are met. The DSM cluster architecture forces this shared memory multitasking to be hierarchical in nature, with a self-scheduled list of DSM tasks each of which is in turn decomposed into a self-scheduled list of CPU tasks. Programming techniques for the CPU tasks in a DSM machine and for the DSM tasks in a DSM cluster can be the same, except for memory access granularity and, in the cluster case, for explicit prefetching and writing back of shared data.

To avoid confusion, we note that we use the term "shared memory" programming to mean that all processes communicate only by reading and writing data structures in memory and never directly with other processes. Some might call this "one-sided message passing," but in this view every Fortran assignment statement would have to be regarded as message passing code. One can write shared memory programs using the MPI-2 library, but, given the 40-year tradition of Fortran programming, this could hardly be called natural. The reason that message passing concepts enter this domain at all is the all-important fact that on modern equipment these operations can no longer be viewed as instantaneous, nor can their latencies be hidden by vector pipelining. For many programmers, message passing was their first experience with program operations that require both an explicit beginning and an explicit end ("split" operations). However, for others this concept was already familiar from disk I/O. Thus function calls to read and write data, and subsequent spin waits upon completion of such operations are not new. The

principle difference between asynchronous I/O and asynchronous one-sided message passing is the data access granularity that each can support while still maintaining a reasonable fraction of its peak performance (or bandwidth).

Benefits of the Shared Memory Programming Model

We believe that shared memory programs, in the sense that we have defined them above, are necessary if dynamic load balancing is to be practiced without heroic programming efforts. Shared memory allows the program to be specified as a self-scheduled sequence of tasks, each of which may be executed by any of the available computational resources. So long as there are several times as many such tasks between global synchronization points as there are processors or machines, then even if these tasks take varying amounts of time to complete, the computational resources will all be kept busy, and the whole job will be performed efficiently.

It is essential to understand that in a good shared memory program for one of today's multiprocessor systems very little memory sharing actually takes place. Computing entities make private copies of shared data contexts (if the data is not already locally available), they then work on these data copies in seclusion, and finally they write revised versions of the data back to the shared space. In this way the processors keep out of each other's way, and the vexing problem of "false sharing" is avoided. At the same time they achieve the maximum possible effective memory bandwidth, since their private workspace, implemented on their subroutine stacks, is always placed by the compiler in the best possible place (usually the on-board memory, if not the L-2 cache). By setting semaphores in shared memory indicating that they have read or revised the shared data segments, computers (CPUs or whole machines) can let other computers know what data is up to date. Thus a computer can learn when the data context for its next task is ready, so that the task may begin.

The kind of shared memory program we recommend consists of a carefully ordered list of tasks, each with its own, well defined and carefully packaged data context. These tasks are launched in program order, but they need not complete in this order. The program cannot be efficient unless many of the tasks at any point of the sequence can be executing in parallel. If this number of parallel executable tasks always exceeds by a reasonable factor (such as 2) the number of available computing resources, and if there are no constraints on which resource must execute which task, then all these resources should be kept busy. It is the hallmark of shared memory programs that any computing resource can execute any task. This permits the task sequence to be optimized by the programmer for minimum time to solution. It also places requirements on the design of the computing system, as will be discussed later.

The computational labor within individual tasks or in individual task groups need not be carefully balanced, as is usually attempted in distributed memory programs. Whenever a computer completes a task, it simply begins the next task on the global list. It is irrelevant whether or not other computers have completed their tasks at this same instant. The conditions under which the tasks may safely be launched may be specified in the program as tests on semaphore variables in shared memory. Access to semaphores for this purpose does not require low latency communication, so long as the tasks are very carefully ordered and waiting on a semaphore is extremely unlikely. The semaphores that determine whether a task may execute can be prefetched along with the task data context in order to hide remote memory latency and to accommodate low remote memory bandwidth.

Relation to Other Work

Many investigators have been concerned in recent years with enabling shared memory programs to execute on cluster systems. Some of the early work in this area led directly to the development of distributed shared memory (DSM) machines, particularly the work of the Stanford team led by John Hennessy (cf. Kuskin et al. 1994). More recent contributions (see reference list) have focused on software systems that create from a cluster of machines the practical effect of a DSM machine, rather than simply a research prototype. Much of this work, although not all, has focused on clusters of single-processor machines and therefore features only a single level of the two-level shared memory discussed in the present work. Also, much of the work has involved very fine granularity of software shared memory access, often machine memory page based. In this respect this work stands in contrast to the approach advocated here.

Some of the recent work on out-of-core computation algorithms, particularly that of Salmon and Warren (1997) and Nieplocha and Foster (1996) involves concepts relevant to the work presented here. A feature that this out-of-core work, particularly that of Salmon and Warren, shares with our own is the emphasis on restructuring the numerical algorithm to function well in the new mode. The work

presented here is based on ideas set out in Woodward (1996). Continually updated lists of references to work on SMP cluster computing can be found on the "clumps home page," at http://now.cs.berkeley.edu/-clumps, and through links maintained on that page.

A Virtual Petaflop/s Machine

The model for a petaflop/s system, upon which our experimental run-time system is based, is a cluster of multiprocessor DSM machines with network-attached disks. Our experimental run-time system allows the programmer to view this computing platform as a single machine with a 4-stage memory hierarchy, consisting of (coherent) processor cache, (non-coherent) local shared memory, global shared memory, plus a global disk file system. Large jumps in latency and bandwidth at each stage of this 4-stage hierarchy are assumed, along with corresponding large jumps in computational power. Actual systems are characterized by ratios in latency and bandwidth from stage to stage, and by ratios of each, of latency and of bandwidth, to sustained computational power available at each stage. This virtual petaflop/s machine, or VPM, may be parameterized as follows:

N_{CPU} = number of CPUs per DSM cluster member.

N_{DSM} = number of DSMs in the cluster.

c_{CPU} = Mflop/s per CPU on an application code task.

c_{DSM} = Gflop/s per DSM on an application code task.

b_{CPU} = B/flop = (DSM shared memory bandwidth to this CPU) / c_{CPU}

b_{DSM} = B/flop = (cluster shared memory bandwidth to this DSM) / c_{DSM}

$l_{CPU} = c_{CPU}$ / (latency to DSM shared memory) = flops / (DSM access)

$l_{DSM} = c_{DSM}$ / (latency to cluster shared memory) = flops / (cluster access)

Note that in this list of parameters memory latency and bandwidth appear only in ratios. It is these ratios, rather than the raw latency and bandwidth, that, together with the system size, N_{DSM} and N_{CPU}, reflect the degree of programmability. DSM clusters of similar size and with similar values of these ratios can be expected to run a given program at roughly the same fraction of peak parallel performance. Here by peak parallel performance we mean $N_{DSM} \times N_{CPU} \times c_{CPU}$. Of course, for today's microprocessors we can expect single processor application performance, c_{CPU}, to fall well below the manufacturer's stated peak performance. Since the fraction of peak processor performance actually realized, even after heroic efforts, is usually so low and varies so greatly from processor to processor, it is best here to ignore the manufacturer's claims entirely and to rely instead upon benchmark tests of c_{CPU} for the applications in question. Such tests are easily performed, since only a single processor need be involved, and today's high performance microprocessors are cheap and widely available. In order to get a feel for values of the above system parameters for actual systems today, we compare two DSM cluster systems available to the scientific community at NCSA below using c_{CPU} values for our PPM codes:

A small Origin-2000 cluster with network-attached disks implementing cluster memory:

$N_{CPU} = 128$
$N_{DSM} = 2$
c_{CPU} = 150 Mflop/s
c_{DSM} = 18 Gflop/s
b_{CPU} = 0.7 B/flop
b_{DSM} = 0.015 B/flop
l_{CPU} = 300 flops / access
l_{DSM} = 360 Mflops / access, with cluster memory on disk.
$b_{CPU} / b_{DSM} = 50$
l_{DSM} / l_{CPU} = 1,200,000

A small Intel-based cluster with a Myrinet network:

$N_{CPU} = 2$
$N_{DSM} = 96$
c_{CPU} = 50 Mflop/s
c_{DSM} = 100 Mflop/s
b_{CPU} = 1.0 B/flop
b_{DSM} = 0.1 B/flop
l_{CPU} = 100 flops / access
l_{DSM} = 2 Kflops / access
$b_{CPU} / b_{DSM} = 10$
$l_{DSM} / l_{CPU} = 20$

Note that, contrary to our expectation, the less expensive and less powerful system is the more programmable one, even though the system size is about the same. We will see below that it is the parameter l that determines the granularity of shared memory access that a program must use in order to achieve high parallel performance. This data access granularity is generally related to the task size. The amount of data reuse required for the task to run efficiently is determined by the parameter b. It is commonly the case that larger tasks can achieve greater data reuse. Therefore b tends to determine

task size in conjunction with l. If the task data context is well packaged, so that it can be read or written in at most a few separate sequential data transfers, then l tends to have no further relevance, and it is b that determines the task size according to the following principle. The system must be able to deliver the data context for a task from shared memory and to write back the data resulting from that task to shared memory in less time than is required for the computing entity in question to execute the task once the data has arrived. Clearly, only the ratio b is important; the raw memory bandwidth is irrelevant except insofar as it determines this ratio. With DSM cluster hardware available today, this data bandwidth requirement is not easy to satisfy, but it is not difficult to satisfy either.

Program control that should be exposed to the shared memory programmer

In order to write one of our recommended sorts of shared memory programs, the programmer needs control over the following aspects of the program. This control can come from assertions, directives, hints, or reasonably well founded expectations of the results of the compilation process (the way programmers now control what data is or is not in cache memory).

- Programmers must be permitted to designate the tasks; the program flow then naturally designates the order of task launch.
- Programmers must be able to stipulate task synchronization or lack of same (for example, which previous tasks must be completed before this task may begin).
- Programmers must be able to identify critical and/or atomic operations.
- For DSM tasks, the program must explicitly prefetch task data contexts and asynchronously write back results. Programmers can do this by designating these shared memory operations as separate tasks on the task list.

The shared memory programmer's needs, as listed above, are few. Nevertheless, some system software packages today attempt on the one hand to hide these "details" from the programmer without on the other hand providing any reasonable basis for him or her to build an expectation of the actions that will actually be taken. In so doing, these system software packages obstruct efficient program specification even while they may aid rapid program implementation. Although it is certainly conceivable that a good compiler could assist with task data context prefetching and writing back, it is unlikely that such a compiler would be able to determine the proper division of the labor into separate tasks any time soon without explicit assistance from the programmer.

Today's compilers are highly successful at fine-grained program optimizations, where only a limited context at a time within the program needs careful examination. However, the typical values of the system parameters l and b defined above force efficient programs to have enormous task granularity. It is very difficult, if not impossible, to construct such huge tasks without a deep understanding, at a very high level, of what the program is doing. This understanding is by definition possessed by the programmer; without it he or she could not write the program. It is therefore not a burden for the programmer to express this understanding by designating the separate tasks in some concise way within the program. This programmer expression relieves the compiler of the difficult job of discovering a good task decomposition and allows it to concentrate instead on making the execution of each individual task as efficient as possible.

Determining simple, concise, and effective means of communicating to parallelizing compilers the information necessary for them to automatically decompose the program into parallel tasks is a subject of active compiler research today. While this research is underway, our recommended strategy is for the programmer to state the hierarchical task decomposition explicitly. This is the strategy of the OpenMP standard which is emerging as a very useful tool for writing shared memory programs on SMP and DSM machines today. An extension of this standard to hierarchical task specification seems a natural direction to follow, however, any such extension would have to account for the need for asynchronous data context prefetching and writing back for DSM tasks. One way to do this, which effectively leaves this task in the hands of the programmer, is to designate the data context prefetching and writing back as separate tasks on the DSM task list. It certainly would be preferable if this part of the program implementation could be automated in the compiler.

Assumptions that Permit Efficient DSM Cluster Programs

Our recommendation that DSM cluster programs be written in the shared memory style described above is based upon a small number of assumptions, which we list below. We assume that a job can be decomposed into a set of tasks that can be executed

independently, so long as certain previous tasks are completed at task launch. We further assume that each task can be made to conform to a model, or template, in which:

1) possibly remote data is copied into local memory,
2) this data is operated upon mightily,
3) a few results are written back to possibly remote storage.

We assume that the tasks can be constructed so that, in general, the larger the data context for the task, the larger the amount of potential data reuse. This assumption is necessary to accommodate low cluster bandwidths. To accommodate large cluster latencies we must also assume that the task data contexts can be constructed so that they may be read or written back in only a small number of sequential data transfers. Once in a fast local memory, these data contexts can be efficiently reorganized if necessary.

We assume that global barriers (which give us Amdahl's Law) can be avoided by providing greater system resources and/or by minor modifications of the numerical algorithm. Examples of this principle abound. For example, a program may require all processing to stop so that an image of the problem can be written to a restart file on disk. However, if additional system memory is provided, this restart dump can be written asynchronously without impeding the program flow. Another program might require that a global reduction operation be performed after a time step is completed in order to determine the value of the next time step. However, if enough memory is provided to store the previous problem state, we may guess the time step value (the minimum of the previous 25 time steps might be a good guess) and proceed speculatively. In the rare event that we guess badly, the saved system state will permit us to recover. As a final example, we may be performing an implicit calculation that appears to require global information to be assembled in order to update the value of a variable at a single spatial location. By revising the numerical algorithm slightly, we could require up-to-date information only for the local region and use information from the previous time step or iteration for the more distant data. Once again, this would require a commitment of additional system memory to the job and perhaps an increase in the amount of computation involved.

Services provided by the run-time system

We provide access to global shared memory over the cluster or storage area network through separate memory manager daemons which run on each SMP or DSM machine to serve requests to read and store task data contexts. These global memory servers manage access only to data structures that have been registered with them as global. The knowledge of the global memory layout for these data structures is encapsulated within the memory server processes. Because the granularity of access must be huge, it is generally possible to define these data structures as arrays of records or blocks that need to be accessed individually by DSM tasks. Requests to read or write global shared memory are passed to the memory servers through local DSM shared memory. When a data record or block is brought into local DSM shared memory by a DSM task, it can then be rearranged, or unpacked, for efficient local access. Only the DSM task program needs to know the detailed layout of the record contents. It exploits the low latency and high bandwidth of DSM memory to unpack and repack this record to enable serial data transfers to and from remote cluster memory.

To coordinate the computation, the services of a global task manager are required. If necessary, in a very large system the task manager can be constructed hierarchically. The task manager needs essentially no knowledge of the internal details of tasks or their data contexts, but only the dependency relations between them. Tasks must be able to write back data not only to their own data contexts, but also to others. These other contexts need only be specified in relative terms, and those terms can be interpreted by the memory servers, which have global data layout knowledge (but no knowledge of details).

Feasibility requirements

There are increased bandwidth requirements for this programming model over the usual distributed memory programming model. However, there are essentially no meaningful latency requirements. The bandwidth requirements are determined by the demand that any computing resource at a given level of the hierarchy should be able to execute any task at that same level, regardless of the location of its data context. New requirements of this model at the DSM cluster level for data prefetching and asynchronous writing back are absolutely essential. Despite the demand that any resource be able to perform any task, it is clear that, especially among groups of tasks where launch order is less important, the task manager should have limited intelligence to avoid stupid data movement. It should dynamically reorder the task list, permuting elements that are equally or nearly equally qualified candidates for the next task

to be launched, taking data location over the network into account. A final requirement is that local memory for various computing resources must be sufficient to accommodate data contexts offering sufficient data reuse, but this is not a new requirement.

A possible research agenda

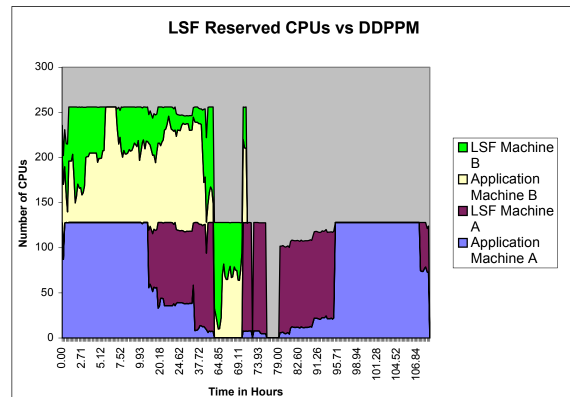
Before hierarchical shared memory can be chosen by the community as a general programming model, we need to see what kinds of problems it can and cannot solve efficiently. To solve problems in a particular application domain, special (new or old) algorithms may have to be constructed or adapted. Limitations of this model for dynamic load balancing in various application domains need to be determined. Finally, the actual extents of increased or decreased system demands need to be established for various application types. We can make significant progress on these fronts by choosing a small number of representative application codes, restructuring them to conform to this hierarchical shared memory programming model, and then determining their performance as well as the network bandwidth and latency ratios that can maximize this performance for systems of given size and power.

Proof of Concept

At the LCSE, we have restructured the PPM gas dynamics code according to the hierarchical shared memory principles outlined above. We wrote the run time system described above, with memory servers reading and writing a fast Fibre Channel network-attached disk system designed by the LCSE's Tom Ruwart. Two 128-processor Silicon Graphics Origin-2000 machines at NCSA, interconnected by a single fast Ethernet, shared a common file system on 48 Seagate Fibre Channel disks supplied by LCSE partner MTI. Each machine was connected to all 48 disks via 4 Fibre Channel loops. Each machine was connected to the disks through its own set of ports (the disks were dual ported). Read/write bandwidth from the PPM application from each machine was in excess of 270 MB/s, sustained, even when both machines accessed the disks simultaneously. Control information, such as DSM task completion semaphores, was passed via MPI over the fast Ethernet link.

This restructured PPM code was used to simulate Mach 2 homogeneous, compressible turbulence on a billion-cell (1024^3) uniform grid. A typical task for a single CPU was to update for one 1-D pass a grid pencil of $4 \times 4 \times 256$ cells. A typical task for a single 128-processor Origin-2000 machine was to update for six 1-D passes, or 2 time steps, a $256 \times 256 \times 512$ brick of grid cells. 32 old and 32 new grid brick records were stored on the shared Fibre Channel disk system. Each grid brick record of 954 MB consisted of 27 separate records: the brick interior (640 MB), 6 brick face records (27.5 MB each), 12 brick edge records (2.4 MB each), and 8 brick corner records (200 KB each).

During each grid brick update, the Origin-2000 was asynchronously prefetching the next grid brick record and writing back the results of the previous grid brick update to 27 different grid brick records on disk. The grid brick record, after being read into DSM memory from disk (in 3.5 sec), was unpacked to form a single, augmented grid brick of $300 \times 300 \times 556$ cells. This brick was then updated in six 1-D passes, with each consisting of 8192 single-CPU tasks (requiring 2.5 sec with 128 CPUs). In this demonstration run, there were barrier synchronization points at the ends of the 6 passes, but these can be eliminated at the cost of further code complexity. After the 6 passes, the new data was written into a new grid brick record in DSM memory, and this was transferred back to disk (in 3.5 sec).



The figure above documents about 4 days of continuous PPM computation at NCSA. Two 128-CPU Origin-2000 systems were used. Processors obtained by PPM on the first system are represented by the cream colored area in the figure. This system was not always available to us, due to scheduling of dedicated access for other jobs. Processors obtained by PPM on the second 128-processor system are represented by the blue area in the figure. PPM adjusted its number of processors on each machine at the beginning of each 1-D sweep for each grid brick. When 128 CPUs were in use, this adjustment interval was about 2.5 seconds. Both machines were shared dynamically with other users, and PPM benefited by inserting requests for CPUs in several batch queues,

grabbing the CPUs as they became available for this single, large computation. The small departures from full resource utilization that are shown reflect system functions performed by the operators, not any failure of PPM to exploit these opportunities.

Reevaluating this Demonstration Code for Parameters of the IBM SP System

Our sPPM ASCI benchmark code indicates that, within a couple of per cent, the delivered performance of a PowerPC 604e microprocessor running at 333 MHz is equivalent to that of a MIPS R10K processor running at 195 MHz. Therefore, one IBM SP "silver" SMP of 4 CPUs gives a value for c_{CPU} of about 600 Mflop/s. This is 32 times less than c_{CPU} for a 128-CPU Origin-2000. Hence to run well on this IBM system we should reduce the computational labor involved in a single DSM task of the PPM code by about a factor of 32. We can do this by reducing the brick volume by a factor of 8, resulting in a brick of 128×128×256 cells, and by performing only a single 1-D sweep rather than 6 per task. This DSM task should now take 15×32/48 = 10 sec. Because we are performing only a single 1-D sweep, the data reuse in this task is 6 times less than on the Origin-2000. However, the "cluster" bandwidth per CPU Mflop/s, b_{CPU}, is now 4×25/600 = 1/6. This is 12 times greater than for the Origin-2000 at NCSA operating from shared memory on the Fibre Channel disks. As a result, the PPM code should run on this system even more efficiently, unless there is contention on the SP's communication network. To do so, of course, the job would have to reside completely in the relatively expensive DRAM system memory rather than on an inexpensive disk subsystem. A form of overhead for the parallel code is redundant computation performed in "ghost" cells surrounding each grid brick. The fraction of the computation time devoted to this redundant work would have remained the same as in the NCSA run if we had performed three sweeps rather than just one sweep per DSM task. We have thus reduced the redundant computation overhead by a factor of $(278^2\times534 - 256^2\times512)/(8\times(130^2\times263 - 128^2\times256)) = 3.9$, and job execution should reflect this lower overhead in greater efficiency.

A Plan to Apply this Approach to Representative Applications

Under the auspices of the National Computational Science Alliance (NCSA), our LCSE team plans to work with two other application code teams to explore the usefulness of the hierarchical shared memory programming model. First, we will work with our own PPM gas dynamics code, but with cell-by-cell adaptive mesh refinement at multifluid interfaces, shocks, contact discontinuities, and slip surfaces. Second, we plan to work with the Princeton cosmology code team to restructure its particle following techniques and its calculation of the gravitational potential using FFTs. Third, we plan to work with the Illinois-NCAR mesoscale meteorology code team to restructure their multiple time stepping to handle the disparate signal speeds of sound and of the wind.

PPM with AMR gives dynamically irregular computational loads. Our demonstration at NCSA, discussed above, shows that we can easily deal with these irregular loads without compromising aggregate performance. Our approach requires a blocked, compressed data structure for all the refined grid data. We may need to subdivide domains that are heavily refined into, say, 8 separate tasks in order to roughly control the variation in processing time required for our tasks. Ideally, we want to have a large number of tasks all of which take approximately the same amount of time to complete. Then we can be assured that the order of task completion will not be too different from the order of task launch. This relationship between order of launch and order of completion is assumed in optimizing the launch order to minimize the chances that any task will have to wait on the completion of earlier tasks.

The mesoscale meteorology code treats fast sound waves by subcycling within a time step sized for the wind velocity, which is 10 to 20 times smaller. Each such subcycle calculation is simple, and therefore its difference stencil is compact. As a result, we can do 10 subcycles within a single DSM task without requiring extreme numbers of ghost cells surrounding a problem subdomain corresponding to this task. If the number of ghost cells required to update a grid block grows too large, we can exercise a fall back strategy by letting sound travel at only, say, 350 mph, which would still be much faster than the wind and should therefore produce nearly the same weather. Meteorology problems can have up to 20 constituents of the air which must be advected along with the wind. Frequently used advection algorithms involve very little computational labor per updated data value. Use of these algorithms could cause the bandwidth requirements for the calculation to rise dramatically. In such problems we can employ instead much more accurate and more computationally intensive algorithms. We intend to experiment with such algorithms and evaluate their additional costs and additional benefits on DSM cluster systems. We also intend to experiment with implicit methods for

sound waves based on domain decomposition and to compare those methods with the subcycling approach.

The Princeton cosmology code computes the gravitational potential at each point with an FFT, which requires global information. The FFT runs from disk, transposing data which is blocked with large granularity. This out-of-core technique requires about 100 MB/s sustained data transfer rates for each 32 MIPS R10K CPUs. In a single time step, only the high-order multipole terms of the potential due to distant galaxy clusters change appreciably, but the effect of these terms on the local potential is entirely negligible. Therefore, in this potential calculation we could use data about distant masses that is one time step out of date. The error introduced in this manner could be quantitatively assessed and kept small. This procedure should still converge to truth upon "mesh" refinement, and it would no longer require tight synchronization between potential calculations for widely separated domains of the problem. Such a technique would remove the necessity for certain task synchronizations, but only at the cost of increased data storage and increased computational labor. For petaflop/s computation, it might prove useful to adopt such modified algorithms, and the Princeton cosmology code provides a context in which such options can be quantitatively evaluated.

Conclusions

We believe that portable petaflop/s programming is possible based upon an extension of shared memory multitasking techniques to address the presumed hierarchical structure of the memories of petaflop/s machines. A hierarchical memory structure, with perhaps even greater challenges than we are likely to find in petaflop/s machines of the future, characterizes DSM cluster systems today. We therefore believe that these systems can be used to develop practical and portable petaflop/s programming techniques today.

The decomposition of a job into a self-scheduled list of tasks for DSM machines, each of which is itself a self-scheduled list of tasks for individual CPUs, is a natural way to address the need for dynamic load balancing in irregular computations or in dynamically varying computing environments. Performing the computation from a persistent problem image in a non-volatile memory, such as a shared disk system, is a means of tolerating the equipment failures that are likely in increasingly complex computing systems. The programming techniques discussed here can guide exploratory efforts. However, we believe that success is most likely if the code and algorithm developers work closely with developers of the special system software that is required. This is necessary to permit global optimization of the overall approach. Success may demand that cherished numerical algorithms be abandoned or significantly modified. It may also require that the functionality and/or performance of the system software be significantly modified, extended, or enhanced. Only through close collaboration can these important changes be made and the lessons learned that perhaps only these changes can impart.

Acknowledgements

We are pleased to acknowledge generous support for the test run presented here from the operations staff of NCSA, working with LCSE disk experts Tom Ruwart and Alex Elder. We also acknowledge useful discussions with Ian Foster and Steve Tuecke of the Globus team at Argonne National Laboratory. The test run of the model presented here is a part of a larger investigation into homogeneous, compressible turbulence, involving David Porter, of the LCSE, and Annick Pouquet, of the Observatory of Nice, as well as Bill Dannevik's ASCI turbulence team at the Lawrence Livermore National Laboratory.

This work was supported by the Department of Energy and the National Science Foundation. Our work on implementing codes efficiently on DSM clusters has been supported by the DoE's ASCI program through a Level-2 project with Los Alamos, subcontract B33700016-3Y, completed in 1998, and by NSF through a MetaCenter Regional Alliance grant, ASC-9523480, and through the PACI program via subcontracts from NCSA. Support for our continued numerical algorithm development and the packaging of our algorithms in library form, PPMLIB, specifically targeted toward SMP cluster systems has come from the DoE's Office of Energy Research through grants DE-FG02-87ER25035 and DE-FG02-94ER25207, respectively. Support for our investigations of compressible turbulence, of which our test run reported here was a part, has come from a NASA Grand Challenge team award, NASA Cooperative Agreement Number (CAN) NCCS-5-151, through a subcontract from the University of Chicago, from DoE grant DE-FG02-87ER25035, and from our DoE ASCI Level-3 project with Livermore, subcontract LLNL/B331627/DOE. We would also like to acknowledge support for computer time from the NSF PACI program through NCSA, and also local support from the University of Minnesota's Minnesota Supercomputing Institute.

References

1. Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W., 1996. "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, Vol. **29**, No. 2, pp. 18-28, Feb. 1996.
(available at http://www.cs.rice.edu/~willy/-TreadMarks/papers.html)

2. Baden, S. B., and Fink, S. J., 1999. "A Programming Methodology for Dual-Tier Multicomputers," to appear in *IEEE Transactions on Software Engineering*.
(available at http://www-cse.ucsd.edu/groups/-hpcl/scg/kelp.html)

3. Bilas, A., Iftode, L., and Singh, J. P., 1998. "Evaluation of Hardware Support for Shared Virtual Memory Clusters," Proc. of the 12th ACM International Conf. on Supercomputing (ICS'98), Melbourne, Australia. July, 1998.

4. Cox, A. L., Hu, Y. C., Lu, H., and Zwaenepoel, W., 1999. "OpenMP on Networks of SMPs," to appear in Proc. 13th International Parallel Processing Symposium, April, 1999.
(available at http://www.cs.rice.edu/~willy/-TreadMarks/papers.html)

5. Culler, D. E., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C., Wong, F., 1997. "Parallel Computing on the Berkeley NOW." JSPP'97 (9th Joint Symposium on Parallel Processing), Kobe, Japan.
(http://now.cs.berkeley.edu/Papers2/Postscript/jpps.ps)

6. Erlichson, A., Nuckolls, N., Chesson, G., and Hennessy, J., 1996. "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory," Proc. of the 7th Int. Conf. On Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, pp.210-20. (available at http://www-flash.stanford.edu/architecture/papers/paperlinks.html)

7. Fink, S. J., and Baden, S. B., 1997. "Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters," *Lecture Notes in Computer Science*, ed. Y. Ishikawa et al., Vol. **1343**, 1-8. (available at http://www-cse.ucsd.edu/groups/hpcl/scg/-kelp.html)

8. Gropp, W. W., Lusk, E. L., 1995. "A Taxonomy of Programming Models for Symmetric Multi-processors and SMP clusters," in Proceedings of Programming Models for Massively Parallel Computers, October 1995, pp. 2-7.
(HTML Abstract: http://now.CS.Berkeley.EDU/-clumps/gropp.html)

9. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J., 1994. "The Stanford Flash Multiprocessor," Proc. 21st International Symp. On Computer Architecture, pp 302-313.
(available at http://www-flash.stanford.edu/-architecture/papers)

10. Lumetta, S. S., Mainwaring, A. M., Culler , D. E., 1997. "Multi-Protocol Active Messages on a Cluster of SMPs," Proc. Supercomputing '97.
(available at http://now.cs.berkeley.edu/clumps/-sc97)

11. Nieplocha, J., and Foster, I., 1996. "Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations," Proc. IEEE Conference on Frontiers of Massively Parallel Computing, Frontiers '96, pp 196-204.

12. Nieplocha, J., Harrison, R., and Littlefield, R., 1994. "Global Arrays: A Portable 'Shared-Memory' Programming Model for Distributed Memory Computers," Proc. Supercomputing '94, pp 340-349.

13. Nieplocha, J., Harrison, R., and Ian Foster, 1996. "Explicit Management of Memory Hierarchy," *Advances in High Performance Computing*, Ed. L. Grandinetti, J. Kowalik, M. Vajtersic, NATO ASI 3/30: pp 185-198.

14. Salmon, J., and Warren, M. S., 1997. "Parallel Out-of-Core Methods for N-Body Simulation," Proc. 8th SIAM Conf. On Parallel Processing for Scientific Computing.

15. Scales, D., and Lam, M., 1994. "The Design and Evaluation of a Shared Object System for Distributed Memory Machines," Proc. First Symposium on Operating Systems Design and Implementation, Nov. 1994.

16. Woodward, P. R., 1996. "Perspectives on Supercomputing: Three Decades of Change," *IEEE Computer*, Vol. **29**, Oct. 1996, pp. 99-111. (available at http://www.lcse.umn.edu/-computer)

Appendix: Tuning the PPM Code for Performance on a Single DSM Machine

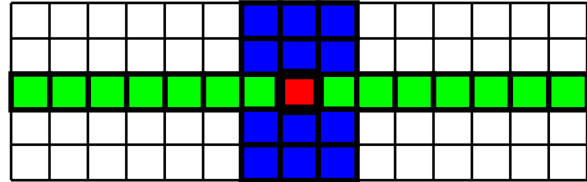
The PPM algorithm for compressible gas dynamics was initially developed with vector supercomputers in mind. As a result, it is formulated as a sequence of 1-D sweeps over a regular grid, with each sweep written as a long series of unit stride vectorizable loops enclosed in outer loops over strips of grid cells. In this original vector code, the data representing the fluid state of a given strip of grid cells was read into the vector CPU from main memory many times over in order to advance the calculation for only a single 1-D sweep. On vector machines of the 1980's this reading and rereading of data from main memory came at no cost in performance. This vector PPM code achieved about half the peak performance of every vector computer on which it was implemented. However, this algorithmic structure does not fit the supercomputing platforms of today, for which this profligate expenditure of memory bandwidth is entirely inappropriate. To make the algorithm efficient on clusters of DSM machines like those at NCSA, the way the computation proceeded had to be entirely rethought and restructured.

Transforming the PPM code kernel for performance on a single CPU

Transformations for L-1 Cache Utilization:

To make the PPM computation run well on a single O2K CPU, it was necessary to change the fundamental unit of work, the task that is assigned to a single CPU only. In the original code for Cray vector multiprocessor machines, this CPU task involved the updating for a 1-D PPM sweep of a single strip of grid cells. In principle, each cell of the grid may be updated independently, but in this update process intermediate results must be generated in neighboring grid cells in the same grid row (the cells shown in green in the diagram at the left). Since these intermediate results can be reused in updating these adjacent grid cells, the overall computation becomes increasingly efficient as the length of the grid strip to be updated is increased. Roughly speaking, the work involved in updating a strip of n cells is $(n+7)\times w$, where w is the work per cell for an extremely long grid strip.

When a single grid strip is broken into sub-sections to be updated in parallel by different CPUs, redundant work must be performed in 14-cell regions centered on the locations where the grid strip was subdivided. This redundant work can be considered as an overhead associated with parallel computation. To keep this overhead low, we try, for large problems, to keep n over 128, with values of 256 or 512 commonly used in practice.



The difference stencil of one variant of the 2-D PPM gas dynamics code for a single 1-D sweep in the x-direction. In order to update the red grid cell in the center, data from the 12 nearby blue grid cells must be used, and intermediate results must be computed in the 7 green grid cells on either side. The number of intermediate results computed is greatest for the green grid cells that are closest to the central, red cell. These intermediate results can be used to update those cells.

The decomposition of the problem into CPU tasks involving long strips of grid cells is not done only to save redundant work but also to achieve reuse of cached data. In its original vector form, the grid strip update code module took as input 5 vectors, one for density, pressure, and the three components of velocity, for each of 9 grid strips – the central one, two above, two below, two in front, and two behind. From these 45 input vectors, over 150 intermediate result vectors were computed in the course of generating the output of the new 5 vectors for the central grid strip. These vectors were computed in a series of vector loops of unit stride. Each successive loop used intermediate results from the previous one in such a way that the loops could not be combined without destroying the vector nature of the algorithm.

In a modern microprocessor, these vector loops are favored for the independent arithmetic that their iterations present, but they demand a large amount of data traffic between the processor's registers and the cache memory. With the vector length set greater than 128, as discussed above, the requisite hundreds of temporary vectors will not fit into an on-chip L-1 cache memory. Performance therefore becomes limited by the bandwidth to the off-chip L-2 cache. To overcome this limitation, we restructured the PPM grid strip update module so that it became a single, monolithic, unvectorizable loop for which software pipelining is nevertheless possible. This was precisely the style in which the precursors to the PPM algorithm were written in the 1970's, before the advent of vector computers. On the MIPS R-10000

processor, using an early compiler, this Fortran-to-Fortran manual code transformation yielded a 50% performance boost for a simplified version of PPM, called sPPM.

We informed Silicon Graphics of this transformation, they included it in their compilers, and now it is no longer necessary to perform this transformation manually. (However, it is necessary to write code that the compiler can recognize as a candidate for this transformation.) Performance boosts from this transformation were similar on the DEC Alpha processors in the Cray T3D and T3E, but they were less on the IBM CPUs. This code transformation yielded a speed-up on every processor, except of course the Cray vector processors, for which we tried it. The very different cache performance before and after this transformation can be seen in the output, below, from the MIPS R-10000 CPU's hardware performance monitor performing a large number of strip updates with n = 400. These tests were performed on July 8, 1997, using 64-bit arithmetic. The R-10000 was installed in a Power Onyx at the LCSE. Tests on an Origin-2000 at NCSA gave essentially the same results at the time.

For the transformed code:

```
agate> perfex -a xprwsppm8 < sod3din-bench
WARNING: Multiplexing events to project totals--inaccuracy
    possible.
1
 card input for this problem was as follows:

 bttpropu   ibot, itop, rho, p, u:
 end        0    0 0.00000E+00 0.00000E+00 0.00000E+00
n= 2984  t= 1.00032E-01  dt= 3.42129E-05  cour= 2.54994E-02
Cycles........................   5187755136
Issued instructions..........................  6592839552
Issued loads.................................  1339296496
Issued stores................................   781376528
Issued store conditionals....................         224
Failed store conditionals....................           0
Decoded branches.............................    77885024
Quadwords written back from scache...........      165760
Correctable scache data array ECC errors................. 0
Primary instruction cache misses..     179648
Secondary instruction cache misses        832
Instruction misprediction from scache
    way prediction table...                         68816
External interventions.......................       15408
External invalidations..... .................       17392
Virtual coherency conditions.................           0
Graduated instructions.......................  5587467936
Cycles.......................................  5187755136
Graduated instructions.......................  5589560912
Graduated loads..............................  1032718016
Graduated stores.............................   781338288
Graduated store conditionals.................          64
Graduated floating point instructions........  1906695216
Quadwords written back from
    primary data cache...                        20124000
TLB misses....................            432
Mispredicted branches........................     1000496
Primary data cache misses.......     15940544
Secondary data cache misses........       160
Data misprediction from scache
    way prediction table....                73152
External intervention hits in scache.........       15408
External invalidation hits in scache.........       17392
Store/prefetch exclusive to
    clean block in scache...                 ..        16
Store/prefetch exclusive to
    shared block in scache....              ...         0
26.637u 0.101s 0:27.29 97.9% 0+0k 22+7io 13pf+0w
agate>
agate> date
Tue Jul  8 18:56:23 CDT 1997
agate>
```

For the vector code:

```
agate>
agate> date
Tue Jul  8 18:59:04 CDT 1997
agate>
agate> perfex -a xprwsppm4 < sod3din-bench
WARNING: Multiplexing events to project totals--inaccuracy
    possible.
1
 card input for this problem was as follows:

 bttpropu   ibot, itop, rho, p, u:
 end        0    0 0.00000E+00 0.00000E+00 0.00000E+00
n= 2984  t= 1.00032E-01  dt= 3.42129E-05  cour= 2.54994E-02
Cycles........................    8316727104
Issued instructions.......................... 12033638400
Issued loads.................................  1744159936
Issued stores................................  1036913200
Issued store conditionals....................         464
Failed store conditionals....................           0
Decoded branches.............................   269331312
Quadwords written back from scache...........     2815360
Correctable scache data array ECC errors......           0
Primary instruction cache misses...   3297168
Secondary instruction cache misses..  199584
Instruction misprediction from scache
    way prediction table...                        691776
External interventions.............       ....      14384
External invalidations...................  ..       20880
Virtual coherency conditions.................           0
Graduated instructions....................... 10560158064
Cycles.......................................  8316727104
Graduated instructions....................... 10567197856
Graduated loads..............................  2706599200
Graduated stores.............................  1035344560
Graduated store conditionals.................         144
Graduated floating point instructions........  2117035840
Quadwords written back from
    primary data cache....          ...........  417686608
TLB misses.................. .           768
Mispredicted branches........................     6666816
Primary data cache misses......    419177232
Secondary data cache misses.......    404160
Data misprediction from scache
    way prediction table..               2168016
External intervention hits in scache.........       14384
External invalidation hits in scache.........       20896
Store/prefetch exclusive to
    clean block in scache...        .........      170656
Store/prefetch exclusive to
    shared block in scache...        ........          16
42.756u 0.150s 0:43.32 99.0% 0+0k 7+7io 4pf+0w
agate>
```

The improvement in the effectiveness of the memory system in this example is dramatic (reduction in primary data cache misses by a factor of 1/26), and the improvement in performance is satisfying. (On an Indy desktop workstation, with the MIPS R-5000 CPU, the performance improvement was 280%, but Indy's were not really designed to run this sort of code.) We communicated this transformation for the sPPM kernel code to Silicon Graphics and IBM, since sPPM was used by the DoE's ASCI program as an official benchmark for its "blue" platform procurements. It is not clear to what extent it has been incorporated into these vendor's most recent compilers, although with these compilers its benefits have been reduced to the range from 1% to 5%. Thus either this transformation or one equally effective has been incorporated.

The extreme version of this loop fusion which we used manually in the test documented above is probably overkill for today's microprocessors. However, as these processors continue to outrun their main memory systems, these transformations may become increasingly important. For the new generation of microprocessors with dual sets of floating point functional units, the monolithic scalar loop of the transformed code may be pipelined, so that two of its iterations are performed in each passage through the transformed loop. This of course involves writing if-tests with four branches in places where only two branches were originally required. If in such cases far more work is done in one branch than the other, in the relatively rare cases where the two iterations go down different branches of the original test the dual functional units will be less well utilized until the control returns to a section of identical code for each original loop iteration.

Transformations for L-2 Cache Utilization:

The updating of a single, long grid strip by now sounds like an ideal task for a single CPU to perform. However, such a task requires that 45 grid strip vectors be read in for each 5 grid strip vectors that are produced as final results. In order to update the entire grid for a single 1-D pass, each strip vector must be read in 9 times for use in updating different grid strips. If all these 9 read operations are performed by the same CPU, then only a single data transfer from main memory is required, and the other 8 are done from the off-chip L-2 cache. Since main memory bandwidth is a problem on most machines today, it is therefore a good idea to bundle several neighboring grid strips together for processing by a single CPU as a single, indivisible task.

We call such a bundle a grid pencil. We therefore update grid pencils of $m \times m \times n$ grid cells in our CPU tasks for the PPM code. If we choose $m = 4$, then we must read each grid strip 3 times from main memory in a single 1-D sweep. If we choose $m = 8$, then we read each strip only 2 times. The larger we set the value of m, the more efficient is our use of the main memory system; however, larger values of m give us fewer CPU tasks for each 1-D sweep. If we have too few CPU tasks, some CPUs may find themselves with no tasks available to do for a short interval while other CPUs catch up. For this reason, it is rarely advisable to choose m greater than 8.

Transformations for Main Memory Performance:

The choice of the transverse width of a grid pencil is affected by another major consideration related to the performance of the main memory system. A CPU must be able to read the data for its grid pencil efficiently regardless of the direction, x, y, or z, of the 1-D sweep. If the 5 physical variables are stored in separate arrays, we must therefore choose m to be 5 times larger to achieve the same effective main memory bandwidth in two of the three 1-D sweeps. Therefore, we interleave these 5 arrays in a single array structure, which now has a fourth dimension, the number of the physical state variable, running from 1 to 5 (the fast-running array index). Still, it is necessary to choose m to be at least 4 in order to get good main memory performance on today's machines. For $m = 4$, we read 40 words at a time, each of 4 (or 8) bytes, regardless of which 1-D sweep we are doing. Hence we do not waste much of the information in the cache lines we access. Even when we use 64-bit arithmetic for the grid update process, we generally store variables in main memory in 32-bit format. There is very little to be gained for the PPM algorithm from 64-bit main memory storage, and much to be lost in the way of main memory bandwidth and main memory problem size.

A final consideration in constructing the single CPU tasks for the PPM code is the layout of the principle 4-D data arrays (one new and one old) in main memory. This data layout interacts with the order in which the CPU tasks are launched. The key principle is that we must prevent situations in which many CPUs try nearly simultaneously to access the local main memory of any single CPU. We can prevent this either by carefully scripting the task launch order or by laying out the data in such a way that to read any grid pencil in any 1-D sweep requires accessing the local main memories of several different CPUs. Either strategy works well, and we find the main memory bandwidth of DSM machines like the Origin-2000 to be thoroughly adequate to our purposes.

Transforming the PPM code for performance on a single DSM:

We have described above the construction of highly efficient tasks for individual CPUs. Each of these tasks follows the design of our template, discussed earlier — (1) a data context, the grid pencil, is read into local, high-bandwidth, low-latency memory, (2) this data is operated upon, performing about 900 flops per updated grid cell (or about 60 flops per word transferred to or from main memory), and (3) results are written back to main memory. The entire task data context fits into any reasonable L-2 cache memory, although this is not a requirement for good performance. If the vast bulk of our computation consists of these grid pencil

update tasks, then we should get very good performance so long as no CPU needs to sit idle while waiting for such a task to be assigned to it. Achieving high parallel performance thus boils down to preventing any CPU from waiting for work.

The most fundamental technique to avoid idle CPUs is to assure that any CPU can perform any unassigned grid pencil update. Because nearly all the data traffic to and from the CPU has been limited to the L-1 and L-2 caches or, if the cache is too small, to the local main memory, the cost of importing the task data context and of exporting the final results is negligible on any reasonable DSM or SMP machine today. For the sPPM kernel code, there were only 16 million L-1 cache misses in over 2 billion load and store instructions, for an *L-1* cache hit rate of over 99%. For this reason, a CPU can execute any grid pencil update at nearly the same efficiency regardless of the location of its data context.

Careful construction of the CPU tasks for the PPM code overcomes the constraints of the "owner computes" rule of data parallel computation, but a CPU can still be idle if we do not order the sequence of task launches very carefully. The goal is that whenever a CPU completes a task, there should be another one that it can begin immediately. Each of our grid pencil updates is entirely independent, so long as we consider a single 1-D sweep. However, to update a grid pencil oriented in the y-direction in the second 1-D sweep of the 3-D algorithm, we must be sure that the appropriate x-oriented grid pencils have been updated. If we launch the x-oriented pencil tasks appropriately, and if we are perhaps willing to update y-oriented pencils of half the maximum possible length, we can be fairly well assured that necessary x-pencil updates will always be completed when the time comes to launch each y-pencil task.

In the PPM demonstration calculation mentioned earlier, each DSM task consisted of six 1-D sweeps over a grid brick with 256×256×512 cells. Each grid pencil was 4×4×256 cells, so that in each 1-D sweep there were 8192 independent CPU tasks for the 128 CPUs to perform. Since each CPU updated 64 grid pencils on the average, we could afford to simplify the code by inserting a barrier synchronization point at the end of each 1-D sweep; this convenience could exact a performance penalty of no more than 1.6%. However, on a smaller problem, updating for example a grid brick of 128×128×256 cells with grid pencils of 4×4×128 cells, we would have only 2048 independent tasks, and this performance penalty ceiling would grow to 6.3%. For a grid brick of 64×64×128 cells and grid pencils of 4×4×64 cells, this performance penalty could grow as high as 25%. Thus it is for small problems that we must work the hardest to order the task list to permit each phase of the computation to begin before the previous one is entirely completed.

Our experience shows that if one is willing to make the programming effort, this task ordering is usually possible. As convenient barrier synchronizations are removed more and more aggressively, the complexity of the code increases due to elaborate tests on completion of previous tasks that must be carried out before each successive task on the list is launched. Debugging of such code is difficult, since race conditions are usually involved. Therefore, there is a practical limit to such multitasking strategies that causes the programmer to accept poor performance on very small problems.

Transforming the PPM code for performance on a DSM cluster:

From the very efficient PPM grid pencil update module, a modular task for an entire DSM machine like the O2K was constructed. This task involves the updating for two time steps, that is, for six 1-D sweeps, of an entire grid brick. The cache-line granularity of access of the DSM main memory is exploited to permit efficient extraction of grid pencils in all 3 grid directions. This allows CPUs to update grid pencils of maximum length in each 1-D pass, and thus to reduce computational labor at the ends of grid pencils that is redundant with that for grid pencils in the next grid brick.

The order in which the pencils are updated can be optimized to allow one 1-D sweep to begin before the previous one has ended. The construction of the pencil update tasks makes so few demands on the shared memory that we need not reorder these pencil updates (suboptimally) in order to have particular CPUs update data that is near to them in any special sense. As a result, for problems in which dynamically changing physical conditions in the fluid flow can cause one grid pencil update to require much more computation than another, all CPU loads are automatically and effortlessly balanced so long as there are many grid pencils to update.

The final stage in the restructuring of the PPM code was to update the entire grid, brick by brick, with dynamic load balancing over all the DSM machines in a cluster. The grid bricks are updated in an order optimized so that a new round of brick updates can begin before the previous round is ended. As soon as a DSM machine finishes updating a brick, it begins on another, assigned to it by a task manager

process.

The brick update is made latency tolerant by structuring it in 3 parts: (1) transferring the grid brick data into local DSM memory, if necessary, (2) updating this data via a sequence of grid pencil updates, and, (3) if necessary, transferring the new brick data back to its proper location. The fine-grained structure of the DSM main memory is exploited to package each grid brick data record into 27 contiguous blocks, the brick interior and its faces, edges, and corners which overlap neighbor bricks. In this format the grid brick record can be transferred at maximum possible bandwidth over network interfaces either to other DSMs in the cluster or, perhaps more interestingly, to a network-attached disk file system. The coordination of brick updates by the DSM machines of the cluster is complicated by the need to tolerate high latency and low bandwidth by prefetching each successive brick data record and by asynchronously writing back each updated brick record while the computation proceeds.