Efficient Implementations of PPM Hydrodynamics on Clustered SMP Computers

Saraah E. Anderson, Paul R. Woodward, amd David H. Porter
Laboratory for Computational Science & Engineering
University of Minnesota

Clusters of shared memory multiprocessor (SMP) systems provide an opportunity to exploit high price-performance RISC processors in quantities sufficient for truly high performance computing. However, awareness of architectural performance bottlenecks must be reflected by algorithm design at the processor, multi-processor, and network communications levels. Efficient implementations, 2-D and 3-D, of the Piecewise-Parabolic Method (PPM) fluid dynamics code are described.

Introduction

This paper describes the implementation of an explicit, regular-mesh finite difference scheme - a version of the PPM hydrodynamics code. The parallelization strategy is a blend of efficient multi-tasking at a shared memory multiprocessor (SMP) node, and message passing between such nodes. Because current network interfaces between SMP nodes are an order of magnitude slower than the communications between processors on a node, the algorithms described here overlap inter-node communication operations with intra-node calculation. This latency tolerant algorithm design is able to overcome the long latencies and low bandwidths found in interstate Internet communication.

Hierarchical Parallelism

Instead of a flat parallelism structure, such as a data parallel, or uniform MPI process structure, we use an explicitly hierarchical structure. Doing so allows us to address inherent features, both advantages and disadvantages, of the system architecture. There are three critical levels in this hierarchy:

1) The single processor (CPU) level. The design of the kernel computation is made as efficient as possible for a single processor to execute. To do this, the computation is organized to reuse cached data.

2) The multi-processor (SMP) level. The decomposition into independent computational tasks is made as efficient as possible,so that multitasking overhead is minimized. Tasks are constructed and processed in an order which reduces the likelihood that execution must wait upon the completion of earlier taks. The separation of the calculation and communications tasks makes the next level easier to implement.

3) The network (cluster) level. Widely separating the dispatch from the receipt of messages allows concurrent calculation and network communication. Redundant computation is also used to reduce the need for frequent network communication.

Optimizations for a single processor.

It would be easy to obtain high parallel efficiency using a poor serial algorithm, because communication and task overhead times would then be less significant, relative to computation time. Of course, since our goal is to solve the problem in less time, it is first necessary to ensure that the compiler is making good use of the instruction level parallelism offered by RISC CPUs such as the MIPS R8000 and R10000, the DEC Alpha and IBM PowerPC. Single processor optimizations for these systems can be categorized as:

- Cache memory related: The performance penalty due to main memory accesses in commodity based (economical) system designs with cache memories. These problems can be reduced by maximizing locality of reference and data reuse. These memory access patterns must be built into the algorithm at a basic level.
- Processor and compiler related: The performance obtained for a code depends on how well the required arithmetical and logical operations can be executed.

Some things cannot be readily optimized; if one must divide or take a square root, we can only hope the compiler and instruction set allow this to be done quickly. Other high level design decisions in the organization of a calculation can radically affect performance.

PPM was developed to run well on Cray vector processors, and so nearly all of its loops are vectorizable, suiting the Cray vector hardware instruction set. These loops are also suitable for the compiler optimization known as ‘software pipelining’, which is a similar technique which interleaves execution of succesive inner-loop iterations so as to maximize the opportunity to use the multiple-issue capability of the processor. While the ‘vectorizable’ characteristic is a requirement for fast codes on the MIPS R8000 processor and others, some processor/compilers are able to efficiently schedule and execute codes with branches and other loop-carried dependencies. With branching within inner loops allowed, the requirement to compute everything for each loop iteration disappears. These so-called scalar codes can have decreased floating-point operation/second performance, but nevertheless shorter elapsed time-to-solution. The MIPS R10000 processor is not a vector optimized architecture; it incorporates features in hardware such as out of order execution and speculative execution, which are intended to run non-vectorized codes well. The DEC Alpha processor also performs well on this sort of scalar code.

Optimizations for SMP computers

Getting multiple processors to cooperatively update their shared slice of the distributed computational domain boils down to efficient task scheduling, emphasising:

- Low overhead task dispatch. There should be a minimum of processor synchronizations, or “barriers”. Since low overhead does not mean zero, the size of a task should be made large enough to overcome the remaining overhead.
- Independent tasks. Memory caches are used in fixed-size pieces, the cache-line size. Overlap on reading memory to cache is probably not important within today’s SMPs, but one should make sure that the conflict known as “false-sharing” on writeback to the cache does not occur. This

is easy to arrange if the shared data locations written by processors are designed to be disjoint. The intermediate results of a task should not be shared but instead placed within the task's private stack.

- Separable communication and calculations tasks. In order to allow the cluster-parallel level to perform well, the calculation task(s) must be separated from the task responsible for handling inter-node communications.

Optimizations for clusters.

This is the stage at which communications becomes an important factor. Once data goes "out of the box", beyond a single SMP, the cost of communication between processors goes up by at least an order of magnitude, given current networking technology. This means that the primary optimization is to minimize communication cost. We can do this by:

- Minimizing the number of messages, and maximizing the size of each, allows the use of relatively long latency, low bandwidth networks. This strategy often involves redundant computation at network nodes to eliminate the need for immediate communication.
- Maximizing the time between messages so as to overlap the communication time with useful computational work.

The structure of directionally split PPM

The particular variants of the PPM hydrocodes described here are directional split, and operate on regular grids. The physical state variables for the fluid are stored in arrays which are distributed across SMP nodes. This domain decomposition must also be regular; the overall problem domain must be split into a **i** x **j** (or **i** x **j** x **k**, for 3-D) array of nodes. The relative sizes of the domains given to SMP nodes should be proportional the number of processors available at that node.

The updating of the physical quantities in a simulation proceeds in discrete time steps. A time step update is accomplished by successive sweeps in the problem dimensions - x & y, or x, y and then z for 3-D. To increase accuracy, these time step updates are done in pairs using the same simulation time step, inverting the order of the directional sweeps.

The fundamental fluid physics is done using 1-D strips of zones. To update a ***pass-direction*** strip of zones typically requires neighboring strips of zones in the other, ***transverse***, direction(s). For example, the PPM algorithm used for the 2-D example requires 2 strips of fluid pressure, density and velocity information adjacent to the strip being updated.

A key feature of the kernel code, which updates a 1-D strip, is the degree to which this work can be done in isolation from other processors. We begin the strip update with known information for ***nbdy*** *fake*, or 'ghost' zones beyond the ends of the current ***n*** zone (*real* zones) strip. By computing decreasing amounts of fake zones in successive kernel loops, we end up with ***n*** updated real zones. This structure allows us to separate the setting of the fake zones, which in a distributed calculation is a communication event, from the computational update.

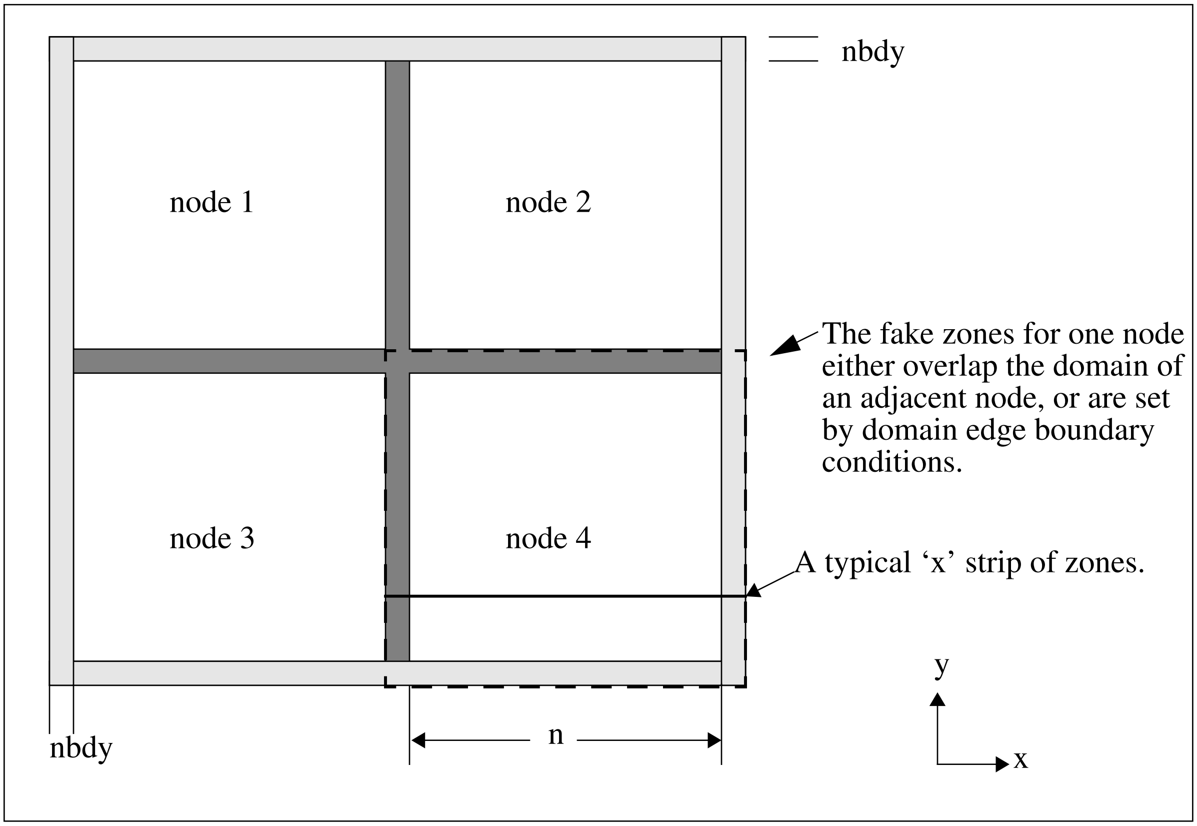


Figure 1: The data distribution for a four node 2-D simulation.

Multiprocessing strategy

In a pass, or directional sweep, each strip update is an independent event. If we imagine that all fake zones surrounding a node's tile are filled appropriately, any strip update task can be assigned to any of the processors on the SMP. Practically, as we shall see below, there are patterns of update which help overlap the communication of fake zones. The loop which updates a domain then simply assigns strips to processors, which update and copy the new state variables to a second array. Since it is unlikely that loads will be perfectly balanced on this fine-grained a view (there are other processes running on the SMP - operating system or communications processes at least), the processors take new work strips as soon as they are available to do so. They are "self-scheduled", rather than being pre-assigned work at the outset of the computation loop.

Calculation and communication cooperation

By ordering the strip updates in certain ways, it is possible to separate the message transmission from the requisite receipt in the next sweep. The complication is the sweep ordering. For 2-D, we pair two time step updates, or 4 sweeps, in this order: X, Y, Y, X.   To keep the sizes of messages large, we have split each sweep into two halves. A message then consists of all the fluid state information for a ***n*** x ***nbdy*** or ***n/2*** x ***nbdy*** edge of the node's domain.

Each half sweep proceeds by assigning strips to processors in the indicated (see figure 2) order. When a communication and calculation dependency exists, the necessary interlock is accomplished with the Unix semaphore facility. This facility encapsulates the atomic increment of a counter, and a process block, depending on the state of the counter. For example, the availability of the top ***nbdy*** strips to send to a neighbor node in part 2 of the X1 sweep is signalled by incrementing a semaphore. Once the semaphore has been incremented ***nbdy*** times, the communication task is free to send the boundary information. Similarly, a calculation task cannot proceed until the *Lt* and *Rt* fake zone boundary information has been received from a node neighbor's previous X2 sweep. The reciept is signalled by decrementing the semaphore on which the calculation tasks may be waiting.

The only remaining complication is the need for valid 'corner' information. Looking at figure 1 again, node 3's upper right-hand fake zone information must come from node 2. Rather than make these diagonal neighbor messages explicit, we have elected to have the communication task effectively forward the information. For example, in the X1 part 1 sweep, as soon as the *Lb* and *Rb* messages have arrived, the communication task *augments* the newly computed bottom edge data with the left & right message information, and sends the bottom edge message to the node "below" it. This defers the sending of the bottom message until the left and right message have been received, eliminating four possible corner messages.

Examining a specific boundary communication in more detail is interesting, and leads to a lower bound on the communications bandwidth. We require (for the worst case, a fully-connected, perhaps periodic processor mesh) that the left and right edge messages be sent, corresponding messages received, the augmented bottom message sent, and corresponding message received. before the updates for sweep X1 part 2 can start.

Assuming a square n x n problem, and further defining:

T_{calc}	The time to update 1/2 of an update sweep,
T_{comm}	The time to send & receive the left and right bottom half edge messages, augment the bottom edge, and then send & receive the bottom/top messages.
C	The aggregate calculation rate in zones/second at a node
R	The communication rate between nodes in bytes/second
b	the value of nbdy, assumed the same for pass and transverse direction
p	The number of processors at an SMP node

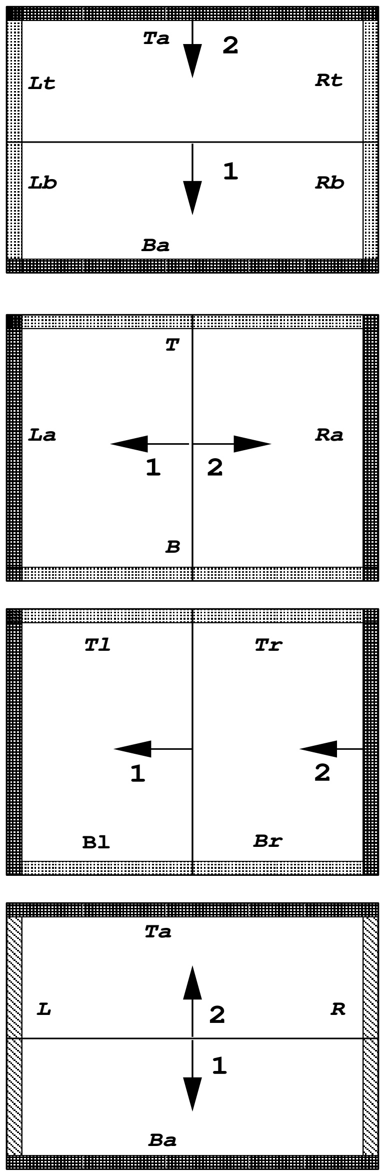
We must have $T_{calc} > T_{comm}$ for overlap. Referring to figure 2 again,

$$T_{Calc} = \frac{\frac{n}{2}n \ \text{ zones}}{(C \ \ \text{zones/second})\,p} \quad \text{and}$$

$$T_{Comm} \geq \left(\left(\frac{n}{2}b \ \ \text{zones}\right)(2\,\text{sides}) + (nb)\right)(20\ \text{bytes/zone})(2\ \text{send\&rcv})\left(\frac{1}{R}\right) = \frac{80nb}{R}$$

$T_{Comm} < T_{Calc}$ means that $R > \frac{160bCp}{n}$.

Plugging in the two tested domain sizes, b=10, and 10^5 zones/second/cpu for 1-D update the computational kernel delivered, we obtain 1Mbyte/second for a 1024x1024 zone problem with 7 processors at each node, or 500 Kbyte/second for 2048x2048 zones. These are upper bound times, but indicate the way the algorithm scales with problem size and node communication bandwidth.



X1

Update 1 begins when Lb/Rb is rcvd from X2. The send of Ba & rcv of Ta is concurrent with the update.

The send of T/B happens as soon as strips are updated.

Y1

Update 1 begins when T/B rcvd from X1. La/Ra send is concurrent.

Send Tl/Bl when update 1 is done.. and Tr/Br when update 2 is finished.

Y2

Update 1 begins when Tl/Bl rcvd. La/Ra send is concurrent with update.

Send L when update 1 finishes, R when nbdy strips of update 2 complete.

X2

Update 1 begins when L/R rcvd, Ta/Ba send is concurrent with update.

Send Lb/Rb when update 1 finishes, Lt/Rt when update 2 finished.

Figure 2: 2-D Domain decomposition overlapping Communication and Calculation.

2-D Performance tests

The performance of this 2-D algorithm was tested on two 8 processor SGI PowerChallenge SMP nodes in the LCS&E laboratory, and on one such node at the National Center for Supercomputing Applications in Illinois. Within the LCS&E laboratory, the network connection was TCP/IP Hippi. The communication between LCS&E and NCSA was standard Internet, in this case a fairly well performing public link.

Figure 3: Parallel Speedup for 2-D PPM on two SGI nodes.

Processors	Speedup	Efficiency	1024 x 1024 problem
1	-	-	
2	1.85	92%	
4	3.44	86%	
7	6.16	88%	
14 across Hippi	11.94	85%	
14 across Internet	6.69	48%	

Processors	Speedup	Efficiency	2048 x 2048 problem
1	-	-	
2	2.26	113%	
4	4.38	109%	
7	7.56	108%	
14 across Hippi	14.9	106%	
14 across Internet	13.0	92%	

While the results of the larger problem are partly obscured by super-linear speedup as a result of cache effects, it can be seen that these two problem sizes bracket problems too small and large enough to run efficiently even on the bandwidth delivered by the Internet connection.

It is interesting to examine the performance of the run over time. Figures 3 and 4, below, show the elapsed time in seconds for a range of time steps. After the starting transient, the elapsed time settles to a constant steady state. In this long time behavior, the communication has been successfully overlapped with computation. Note for the small problem on the internet, the *1024/14* run, the variable traffic on the Internet results in obtaining overlap at only a few time steps. For the large problem, the *2048/14local* and *2048/14* times nearly coincide, which is remarkable for runs done in the same lab with Hippi interconnects, and runs across the public Internet, respectively. These were demonstration runs, and large scale applications using more powerful computational nodes will require correspondingly faster communications links.

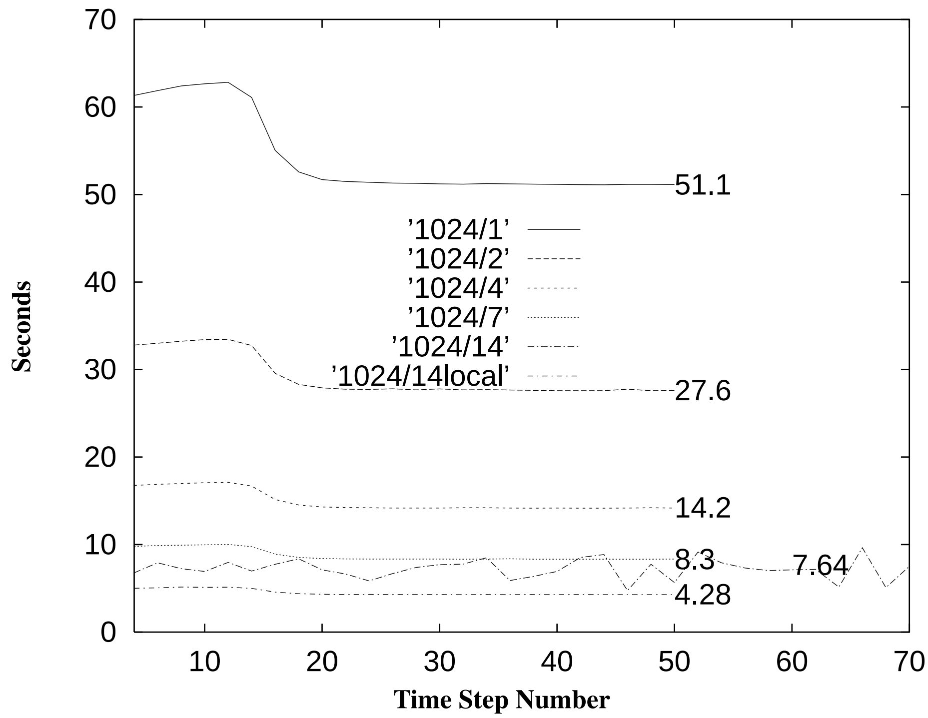


Figure 4: Several 1024x1024 PPM problems, showing elapsed time in seconds for 70 time steps. The suffix /1, /2, etc. is the number of SGI processors at each of two locations. The run 1024/14local was done using a TCP/IP Hippi LAN, not the Internet, for comparison.

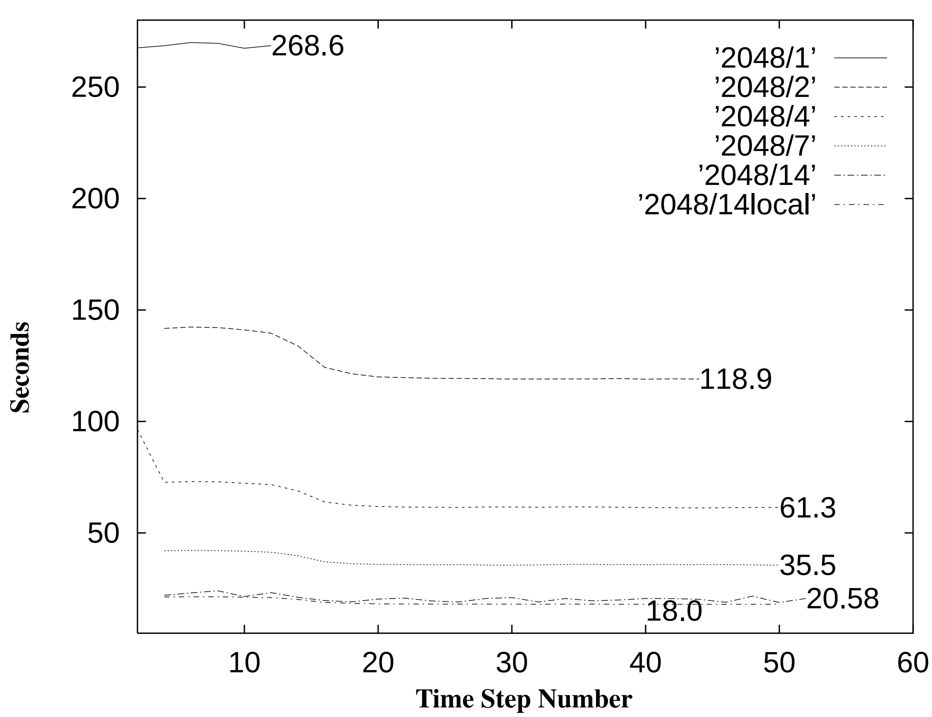


Figure 5: Several 2048x2048 PPM problems, showing elapsed time in seconds for 60 time steps. The suffix /1, /2, etc. is the number of SGI processors at each of two locations. The run 2048/14local was done using a TCP/IP Hippi LAN, not the Internet, for comparison.

3-D Regular mesh PPM

A major performance problem for 3-D mesh applications is the performance loss due to large stride memory access on cached memory systems. As in the 2-D case, we apply PPM in a directionally split way, with successive x, y, and z passes. By defining regions, *pencils* of size (n_{pass} x m x m), as the single-processor units of work, we can extract and replace the data needed for the update in stride 1 memory order. Figure 6 below depicts a y-pass pencil as it is extracted or replaced into the main data array. Note that the pencil is also transposed; the pass direction, y, is stride 1 on the pencil buffer which is used for reading zone strip neighborhoods.

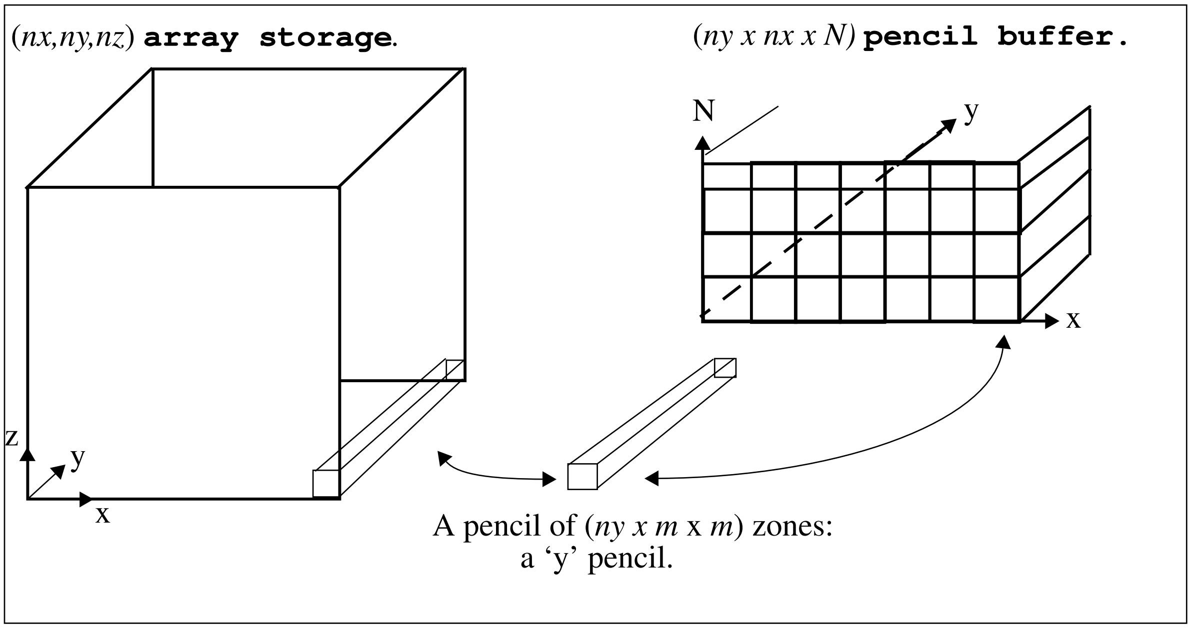


Figure 6. The sPPM buffered update strategy

To unify the tasks of building the pencil buffer and actually updating a pencil, we stagger the load of the pencil buffer with the update. This also allows (at the cost of a very rarely necessary wait) writing the updated data back to the source array. The parallel task loop to perform a y sweep is then:

For all pencils $P_{x,z}$ x=1..$N_{xpencils}$ and y=1..$N_{ypencils}$
if z < $N_{zpencils}$ then copy pencil $P_{x,z+2}$ to the pencil buffer.
while pencils in the neighborhood (+/- 1) of $P_{x,z-2}$ have not been updated, wait.
Update pencil $P_{x,z+2}$ with PPM, and store the result in the array storage.

This is the approach used by the 3-D benchmark code "sPPM". More information, and the code itself can be obtained at the world-wide web site:
http://www.lcse.umn.edu/sppm/README.html

Overlapping Communication

To accomplish some overlap of communication and calculation, it is natural to send messages between nodes as early as possible. In the next version of a 3-D driver to be described, we send boundary zone messages as soon as a CPU has computed them. In actual implementation, each CPU hands any necessary message to a separate process on the SMP, which is responsible for serving all the computational thread's communication needs. This communications thread does the actual network sends and receives. It turned out to be a simplification of the code to divide the work this way, and is in fact required by the versions of MPI libraries we tested. The MPI libraries were unfortunately not designed to be *thread-safe,* that is, they do not allow more than one thread on an SMP to call communications routines simultaneously.

In late October of 1996, time was scheduled on the SGI PowerChallenge array systems at the National Center for Supercomputing Applications (NCSA), in Champaign, Illinois. First, a test problem using 4 SMP nodes was run on these machines:

Node	CPUs		
Name	Available	Used	R8000 equivalents[‡]
vidar	20 r10000	15	19.5
freya	16 r10000	15	19.5
bragi	16 r8000	15	15
odin	16 r8000	15	16
	Total	60	69

[†]On this code, the R10000 systems gave us 1.3 times the performance of the R8000 systems.

The domain for a 512x512x512 zone problem was decomposed by cutting it in *z* and *y,* in such a way as to statically balance the loads on the machines. The PPM kernel used requires about 3000 Flops/zone to perform a time step; on average, a time step pair required 134 seconds. The delivered performance then is about 6 Gflops. Also, since we obtain 100 Mflops/ R8000 CPU on this code:

Speedup = 6000 Mflops / 100*(69 R8000 equivalents) = 60/69
Efficiency = 87%

This code was also used to compute a more challenging non-uniform problem on a less heterogeneous collection of machines. We used 6 machines:

Node	CPUs	Used	R8000 equivalents
odin	16 R8000	15	15
thor	18 R8000	17	17
vidar	20 R10000	16	20.8
magna	16 R10000	15	19.5
freya	16 R10000	15	19.5
bragi	16 R8000	16	16
	Total	94	107.8

The problem was to compute a rotating, convecting solar atmosphere - a model star. More information can be found on the WWW site http://www.lcse.umn.edu/ncsarun. Given the above set of processors, and the fact that there was hydrodynamics to compute only in non-vacuum zones, which comprise about 1/3 (1/3.02 at the initial state) the zones in the cubic mesh of 544^3 zones, it was expedient to cut the domain into 6 z slices. Unfortunately, this simplification limits our maximum efficiency. Communication between nodes is indeed overlapped with computation, but there is a synchronization imposed by requiring the computation sweep to be 'slab-wise', as was done in sPPM. The result is that a directional sweep may not begin until all messages have been sent, and incoming messages received. We obtained on this cluster, a time-step pair in 103 seconds on average. Since there was not enough memory on a single node to run the 544^3 problem on one CPU, we estimate the single CPU time from a 256^3 zone problem. This smaller problem, using a single R8000 CPU, requires 844 seconds for a time step pair. Since there is 9.6 times as much work in a 544^3 zone problem, we estimate the single CPU time for a pair of time steps as 8102 seconds. Therefore, the speedup and parallel efficiency obtained were approximately:

Speedup = 8102 seconds / 103 seconds = 78.7
Efficiency = 78.7 / 107.8 = 73%

Asynchronous Overlapped Communication

While economical with memory, the previous approach does not lend itself to an arbitrary pencil update order. To asynchronously overlap calculation during a sweep with the communication needed for the next sweep, it is necessary to update zones on the edges of distributed domains first, and dispatch the new values immediately to neighboring domains, as was done in the 2-D case. For the next refinement in 3-D driver design, we use a complete second copy of the 3-D arrays, alternating source & destination with successive passes. This vastly simplifies the asynchronous communication of boundary zones, as incoming boundary messages can simply be placed in the fake zones of the 2nd copy array storage, in any order. This is an important advantage. For example, consider the interlocks needed to compose bits of x-pass boundary information for y-pass pencils. The second copy is not as wasteful of memory as it seems. For sPPM running on nodes with many processors, the size of the slab buffer required to keep the buffer write (pre-load) separate from the update approaches the size of a complete second copy of the arrays.

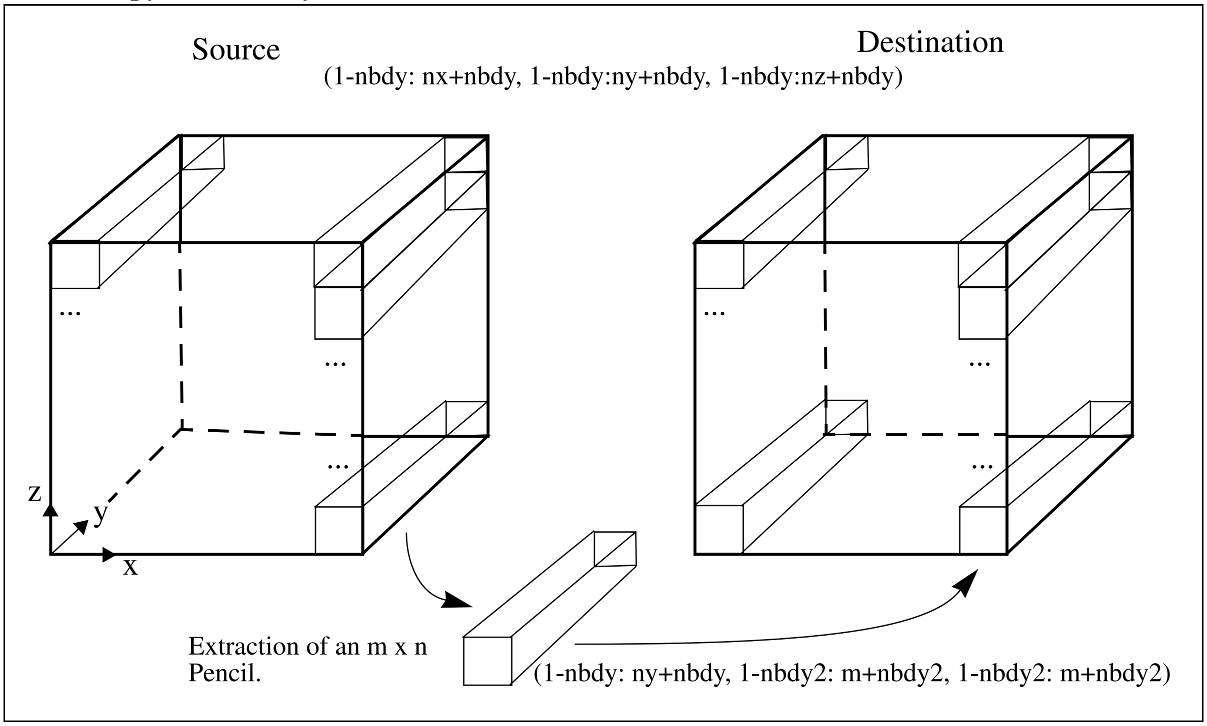


Figure 7: Pencil update with two array copies

To totally encapsulate the pencil sized work unit, it may be necessary (depending on the 3-D stencil for the algorithm in use) to augment the pencil by *nbdy* zones in the pass direction, and *nbdy2* in transverse directions. For PPM, typical values are 7 and 2, respectively. The size of the pencil itself should depend on the cache size. In practice, this size is not critical; the gain made by re-using cached data in the pencil was realized by doing it at all; varying the pencil width *m* from 4 to 16 had very little effect. A typical value is 8. For a specific example, 5 data/zone, 4 bytes/datum, 2 transverse zones, this leads to a <u>minimum</u> contiguous memory access of (5)(4)(8+2+2) = 240 bytes.

Communications strategy

Referring to fig. 7, and recalling that we will follow a y-sweep with a z-sweep, we see that it would be advantageous to update the top & bottom z-caps of the domain first. As each CPU completes its update of a pencil, it sends the new values to neighboring nodes as boundary zones for the next sweep. Once the entire wrapper of y pencils has been updated and appropriate messages dispatched, the CPUs at a node continue updating the interior of the domain.

Boundary messages received are stored in the fake zones bordering the data. So how does the CPU extracting a pencil “know” when the fake zones are appropriately updated? Rather than create a complex system of counters or semaphores to signal this, we simply proceed on the assumption that the fake zones *are* current; but we also compute a checksum. A convenient flag value for a non-current boundary zone is an IEEE “not a number” floating point value. If any of these are found in boundary zones, the CPU attempting the pencil load sleeps -gives up its time slice for a while - and tries again. These flag values are created in the destination data arrays pencil-by pencil, at the same time each updated pencil is written.

Origin2000 Test Results

We had the opportunity to test our code on a 64 processor Origin2000. This test, using a 512x512x128 mesh, gave the following on this single powerful node:

Processors	Pair of time steps Elapsed Time (sec.)	Speedup	Efficiency
1	1622	-	-
4	394	4.11	103%
16	96.4	16.8	105%
32	49.3	32.9	103%
64	27.2	59.4	93%

We need to be careful interpreting parallel efficiencies on this new architecture. It is unlike previous SMP machines in that it has physically distributed memory even within a single "node". Therefore, we see super-linear scaling because the single node timing actually is using memory belonging to the other processors in the machine. Still, this code runs 1.12 times as fast on a single O2000 processor as it does on a single R1000 CPU on an SMP. This tells us that the multiple processor O2000 scales very well from good performance on a single CPU. The performance at the 64 processor level is 6.7 Gflop/second.

Further testing was done on a cluster of four Origin 2000 SGI machines, again at NCSA. These four machines were 32 processor nodes, each node having between 3 and 4.8 Gbytes of memory. The nodes were connected with 10 Mbyte/second 100Base-T fast Ethernet. Unfortunately, the network did not prove to actually provide 10 Mbytes/second between pairs of nodes *simultaneously*. Isolated tests between pairs of nodes did deliver 8 Mbytes/second using 64KByte TCP/IP packets. However, when both node pairs used the network simultaneously, we saw as low as 3.3 Mbytes/second between nodes. This is not sufficient to keep up with the calculation rate of 32 CPUs at a node; we really need all the performance expected from fast ethernet to keep up with the processors. We wanted to continue the 544^3 problem begun on the

PowerChallenge array, and collect comparative performance data, so we proceeded, using as many processors on each node as possible given the network constraints.

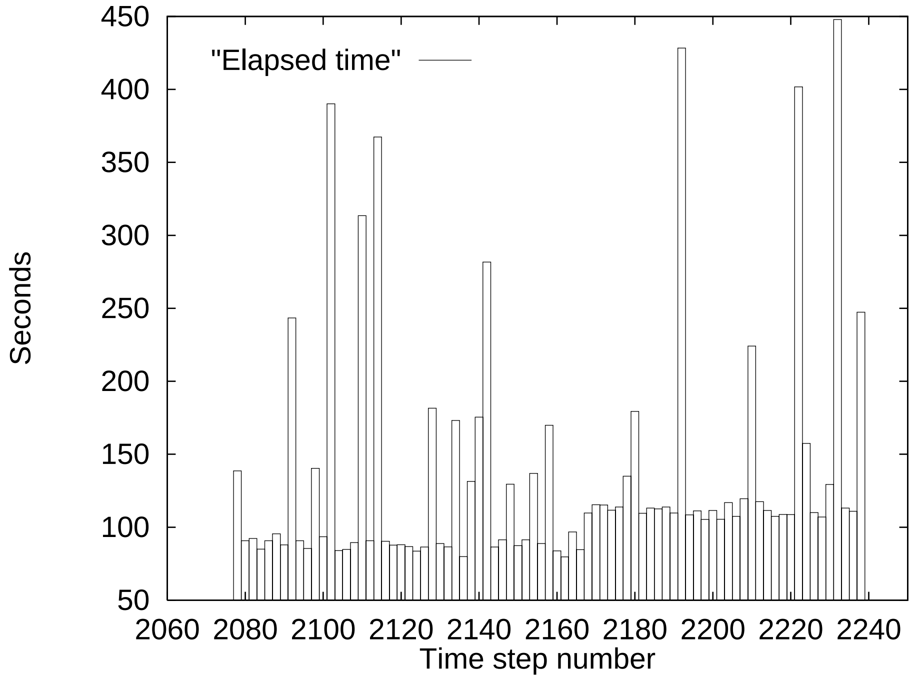


Figure 8: Time in seconds for a pair of time steps, on four O2000 nodes running the 544^3 star simulation.

Time steps 2078-2166 used 16 processors per node, and 2168-2240 used 12 processors per node. Time steps affected by writing output files stand out rather well, taking over 200 seconds. The range of steps from 2126-2160 are affected by sporadic heavy use of the network. Discarding times more than twice the average as presumably anomalous or output related, and taking the average results in:

64 = 16x4 processors: 88.0 seconds per time step pair,
48 = 12x4 processors: 112.0 seconds per time step pair.

Using some approximations, we can compare this result with the PowerChallenge array run. For the array we used 83 PowerChallenge R10000-equivalent processors to compute a time step pair in 103 seconds. Here, we obtain 92% (103/112 seconds) of the performance of the PowerChallenge array, interconnected with Hippi, with 42% fewer processors interconnected with fast Ethernet.

These tests predict high performance can be maintained once a reasonably fast network interconnect is available. As tested, the Origin 2000 cluster more closely approximated a wide-area network than a cluster in a single machine room. Taking the “silver lining” in these troubles suggests contemplating using machines of this class distributed over high-speed WANs such as the vBNS. The vBNS is well capable of speeds required to run our codes on machines as powerful as the 32 processor nodes we tested. However, also learning from this experiment, actually delivered network bandwidth must be demonstrated at the application code layer, whether it be an MPI library, or any other message passing mechanism.