Cluster Programming with Shared Memory on Disk

Sarah E. Anderson, B. Kevin Edgar, David H. Porter & P. R. Woodward
Laboratory for Computational Science & Engineering (LCS&E), University of Minnesota

Abstract

The advent of high-performance workstation clusters with matching high-performance network interconnects offers the opportunity to compute in new ways. Cluster implementations of our hydrodynamics code typically have left domain decomposed independent data contexts in each cluster member's memory, sending only an updated "halo" of domain boundary information to neighboring nodes. On these new network & processor balanced clusters, we have enough separation between data production and reuse to completely overlap the read or write of two complete contexts to remote node disk with the update of a third context. Any node may update any context, resulting in a simple coarse-grained load balancing capability. In addition to essentially continuous check pointing on local node disk, we may stop and restart a computation on any subset of the cluster, or run "out of core", solving problems larger than would fit in even the whole cluster's memory. Our hydrodynamics application is based on a small Fortran or C callable library implementing asynchronous remote I/O, transparently reading and writing named data objects residing anywhere on the cluster. In addition, library routines are supplied to simplify master/worker task queue management. To the application programmer, the library offers the basic interlock-free read/write capabilities of a shared file system without giving up the performance provided by a high-speed network interconnect.

1 Introduction

1.1 Background & Motivation

Our group performs simulations of hydrodynamic flow for astrophysical problems, and more abstractly, for the study of fluid turbulence [16], [17], [24]. The CFD solver we use is called PPM (the Piecewise-Parabolic Method), created by Woodward & Collella. The numerical method is a finite-difference explicit scheme, and features a high degree of data re-use, employing a series of small difference stencils. The data use pattern, good for efficient cache memory use, is also the key to the method's advantages in parallel computing. By performing redundant computation at the ends of a node's domain one can stave off the need for communication for multiple directionally split sweeps or even multiple ***time-steps*** (sets of sweeps advancing the solution in time.) This is efficient on reasonably sized meshes due to the surface (communication) to volume (computational load) ratio. The advantages of high data re-use and cache-friendly coding have become well known, and similar organizations are used in many other numerical codes. Therefore, the method described here applies generally to numeric algorithms other than our own PPM scheme.

Earlier SMP clusters we have used have had as many as 128 processors per communication node (SGI/Cray Origin 2000 systems) and were connected with moderate to low performance networks. Our first task was to efficiently use the CPUs within a node. We found that by considering a nodes domain as a collection of independent tasks, and by CPUs selecting and solving each task in a self scheduled first-come first-served way, we could well use both CPUs and the cached distributed shared memory within the node. For details, see the description of the sPPM benchmark code [22]. Of course, we wanted to extend this load-balancing technique across more than one node. To do this in the same way as within a node requires high-speed communication between nodes, which was not available. However, as an experiment & demonstration, we did create such a well-connected system. In late 1998, we connected a dual-ported fibre-channel disk array to two 128-processor Origin 2000 systems at NCSA. The hardware permitted each system to read or write to this shared disk device in excess of 250 Mbytes/second, well faster than any extant network. At the time, no high performance global file system existed for these systems; therefore, we used "raw" I/O, that is, no file system at all. We arbitrated I/O operations in the application, keeping reads and writes disjoint and properly ordered. This worked well, and proved we could operate efficiently using any combination of active computational threads on each machine, up to and including all 256 processors. Since the data contexts were resident on disk we enjoyed the considerable side effect of having a non-volatile shared memory. This experiment and some computational aspects of PPM are described further in [23].

1.2 Where we are now

Of course we next wanted to extend the preliminary Origin 2000 results to other clusters and programmers. The systems we wanted to address were larger clusters of nodes with fewer processors - as few as 2 processors. Happily, the networks are now much faster, with low cost yet high performance options such as Myrinet or Giganet giving nearly the performance between nodes the large SMP systems had within a node's distributed shared memory. The movement away from TCP/IP [6] based protocols for intra-cluster communications towards small, fast protocols such as Fast Messages [4] was also a promising step. Measured performance delivered to applications, even with an additional software layer implementing MPI messaging exceeds 80% of the bandwidths inherent in the hardware –or at least, the lowest available level of software running on the communication hardware. [12].

Unfortunately, the convenience of high performance shared-memory programming is missing on these clusters, so lacking some sort of meta-language preprocessed to a standard language, all communication must be explicitly coded. The communication code need not be message passing (we have explored various single-sided communication possibilities similar to Cray's shmem library, or the remote memory access get/put libraries within HPVM), but the communication event must be actually coded by the application writer.

We have adopted an abstraction for the cluster communication problem that uses the very familiar single-ended communication concept of the ***file***, a named collection of bytes. These 'files' need not be actually disk resident; we intend to permit memory resident objects, application specific virtual file systems, using the idea of the file read or write only to express the needed communication. This abstraction has the right semantic hint for application programmers, in that the granularity for efficiency is implied to be large. The library implementing these operations would have value in portably packaging these I/O operations, even if they only mapped to an actual global shared file system. Practically speaking, such a globally shared file system is unlikely to be currently available in large computational clusters because such high performance is expensive, implying SAN "fabrics" (switches and host adapters) prohibitively expensive to install on all nodes of a cluster.

1.3 Related Work in Communications and I/O

The Linda project had the idea of named, persistent data objects, in their case also including the program (or reference to a program), which was to operate on the data. [3]. Linda did not, however, include concepts of disk residence, virtual or otherwise, or of asynchronous operation. The Linda "Piranha" scheduler is also similar to the master/worker-pool task assignment logic at which we arrived. The master/worker structure for dynamic process

management and task assignment is a natural fit for others as well as us, and has been previously suggested and supported [8].

We are beginning to consider addressing remote memory to be too low a level of abstraction for scientific application programming. It is, however, a wonderful basis for implementing a library such as the one this paper describes, the “Shared Memory on Disk”, or SHMOD library. (A close relative might be SHMON, “On Network”, or, perhaps, SHMOE “On Everything”.) Cray Research offered a simple remote memory put & get library, “shmem” for the T3D/T3E systems. An IBM offering is the LAPI [9] library for single-sided communication and active messaging for SP systems. Single-sided communication operations are present in the MPI-2 [11] standard. Unfortunately, MPI-2 is unavailable on the systems we contemplated using.

A few groups, most notably at PNL [13], provide libraries for remote memory access across networks but have not as yet released versions for the high-speed cluster network we need. The ARMCI (Aggregate Remote Memory Copy Interface) library [15] is the basis for the Global Arrays library. This library is aimed toward linear algebra problems, and so simplifies access to distributed global n-dimensional arrays or array sections. While some features of this library would be convenient to our application directly, we wanted a minimal set of simple high-performance functions and felt the packaging of data for communication was not an undue burden for the application programmer. Our approach also applies to irregular data distributions, i.e., not arrays. There was specific attention given to disk resident global arrays in a related project [14]. What most distinguishes that work from ours is the view of I/O operations as collective, rather than independent. This supposes an algorithmic structure that is more fine-grained and synchronous, less truly MIMD than we prefer. This bias to parallel I/O as a collective act is especially prominent in MPI-2 or MPI-IO [5], [19] inspired efforts such as ROMIO [18] and RIO [7]. It is possible to conceive of programming our application using the collective style for cluster I/O, but we would lose flexibility for no performance gain and some loss in simplicity, clarity, and future portability.

Many groups have addressed the more general remote I/O problem. GASS [2] is oriented to wide-area networks, and uses the Globus toolkit to solve the thorny problems of access to non-local and non-heterogeneous systems. Some, such as Condor [10] and UFO [1] viewed the solution on local system networks more as a convenience, giving transparent remote access to typically small control or output files. As with GASS, asynchrony was not supported, with referenced data files being automatically staged to the local file system. However, some scheduling features of Condor are intriguing. Condor began as an idle workstation scavenging scheduler. Condor’s features for dynamic process addition and deletion may provide a solution to the problem of dynamic process scheduling, which we do not currently address.

The Legion project [21] has, among other things, an extensive remote I/O suite, incorporating the idea of a virtual, rooted file system (based on host system file systems) implementing a globally shared file system. The wide-area network emphasis, and the complexity and scope of the Legion system did not match our requirement for a minimal easily ported small set of functions delivering a high fraction of the cluster network bandwidth. Another project based on Legion is "Smart File Objects" [20], which describes object-oriented application specific remote file access methods. These "smart objects" are application defined class methods, such as "get_next_row" (intended for matrix operations) that provide access to remote server functions for data transfer, including adaptively optimized data pre-fetch. We opted instead to identify and supply to Fortran and C the minimum possible set of generic byte transfer operations: "read" and "write". We then concentrated on optimizing bulk I/O across the cluster local area network.

2 Using SHMOD

2.1 The Application and Target System

We wanted to re-visit a study of 3-D homogenous compressible turbulence [24]. This new study will involve a relatively short simulation time run with large to record setting mesh resolutions of 512^3, 1024^3 and 2048^3.

The system we wanted to use was a cluster of IBM Itanium workstations at NCSA. This cluster consists of 32 Myrinet-connected nodes, each node containing dual 733 Mhz processors and a 1 Gbyte of memory. For the problems we had in mind, running "out of core" was a necessity, as we wanted to be able to run less than the full cluster. Even the mid-sized 1024^3 -zone problem requires a minimum of 24 Gbytes to store one copy of the fluid state quantities.

Our group has managed a clean separation between the computational physics code and the parallel structure code. This is helpful, because different people, who apply their own specialties, physics or computer science, write these two components. This paper will describe the parallelism and data movement aspects of the code, which we generically call the ***driver***. This code is responsible for presenting fluid state variables to the computational ***kernel***, and performs all necessary data marshalling, storage, and task assignment.

2.2 Domain Decomposition into named objects: *things*

For a cluster, we must split the global problem into at least as many domains as we expect to have nodes; more are better, so we may have the opportunity to load balance between nodes. We must compromise, however, because as the domains shrink, the ratio of redundant computation on domain edges to domain volume increases. We split our 3-D global domain in two

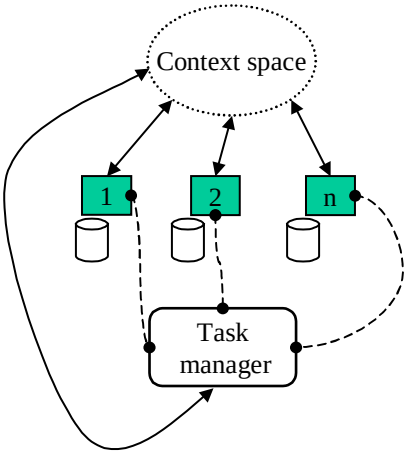
dimensions, creating 'pillars' of the problem which are assigned to nodes for update. For example, in the 1024^3 –zone problem, we split the domain into 32 pillars each 256x128 zones, resulting in 8 256x128x128 computational sub-domains within each pillar. That size is close to the largest we can fit in 1 Gbyte of memory, as at least 4 (we currently use 5) copies of this sub-domain (together with a halo of boundary zones) must exist concurrently in a node's memory to allow a pipeline of pre-fetch, update and write-back to exist.

The pillar of zones assigned to a node for update defines the named objects (hereinafter referred to as ***things)*** we use our library to share. Each pillar with its halo of boundary zones around it comprises a complete, independent work context for the node. We have, then, 4 slabs for each border face of the pillar, 4 slabs on each border corner, and the interior itself, the largest piece. The entire ensemble must be double-buffered, as we want to have a set being read (pre-fetched and updated), and a set being written. This lead us to a simple naming convention for the things of the form "S-xy-[ddd, bxl, bxr, byb, byt, bxlyb, bxlyt, bxryt, bxryb]" where "ddd" refers to the pillar interior, and the others to the boundary regions just described. "S" is either "0" or "1", the double buffer set numbering, and "xy" is the pillar position in the 2-D domain decomposition. A node update proceeds then by pre-fetching the next portion of this context, while updating a fetched portion, and writing back an updated portion. This process repeats in a pipelined fashion to overlap all the communication (I/O) and calculation.

2.3 Cluster Processes – the Task Decomposition

Each "thing" must have a ***home***, a node whose memory and/or disk serves as the things normal repository. In our out-of-core application, this means the node has a disk file with the same name as the thing. All processes globally know the home node of each thing, so when a read or write request is issued it may be relayed by the library as a remote I/O operation to the home node. In this way, the computational nodes double as memory servers, reading and writing their local disk on behalf of non-local requests. This function could be split off into a separate process entirely, but for reasons to be described later, this was a convenient implementation initially.

When cluster processes start, they examine the contents of a specified local disk directory for things. They then share the fact they "own" them with all other cluster processes. When a thing is created or destroyed by a process, that fact is similarly shared. These thing names must be unique across all processes, so that subsequent read or write operations on any node need only name the thing. At present, no facility is provided to relocate things, as that was not needed by our first application.



The nodes expected to update data contexts need some global coordination. We have chosen to do this with a "master/worker" scheme which concentrates all global task assignment logic into a "master" we call the task manager. The ***n*** computational nodes and the manager communicate with a set of task communication routines provided by our library. (The dashed lines in the figure to the left diagram possible task assignment or request channels.)

3 Application Programmer Interface Summary

3.1 Input/Output

The essential idea is to support a uniform, global name space of objects (things) across participating user programs. This is done with simple calls to read or write, which are provided in non-blocking form.

`integer start_thing( rootDirectory, sharedFileSystem )`

> ***rootDirectory****:* The disk objects on this node will be 'rooted' at this location on the local file system. If *!sharedFileSystem*, the names of files (objects) in this directory will be communicated to other routines so they may be used in read or write calls.
> ***sharedFileSystem****:* If a shared file system is used, non-zero, otherwise 0 indicates the supporting file system is local to each node only.
>
> This call is necessary for each participating process to make once, at start-up. The process is assigned an identifying integer (node number), which is 0.., returned by the routine.

`integer create_thing( name, attributes, initialSize )`

> ***attributes***: DISK, MEMORY, etc.
> ***initialSize***: Size in bytes of the object initially. This may be zero.

This routine instantiates an object. The name must not previously exist (globally), and the created object name will be passed to other nodes before this routine returns.
Note in the case of a non-shared file system that the node, which calls create_thing 'owns' it in the sense that the bytes will physically reside on the owning node.

integer read_thing(*name, offset, address, nbytes, status*)

offset: byte offset at which to begin reading
address: local address (array) at which bytes are deposited
nbytes: total number of bytes to transfer
status: status array, dimensioned locally to pass to wait_thing.

integer write_thing(*name, offset, address, nbytes, status*)

Same arguments as read_thing().

integer wait_thing(*status, waitflag*)

status: array previously passed to read_thing or write_thing.
waitflag: -1 (FOREVER), or the number of milliseconds to wait for a completion.

GetIOstatistics(*status, bytesPerSecond, nbytes*)

status: result returned from a completed I/O operation, as tested by wait_thing.
bytesPerSecond: (real) I/O speed from the start of the operation to its completion.
nbytes: Number of bytes transferred.

3.2 Task Assignment and Reporting

Task management routines have a common interpretation for parameters. The 'task' identifier is handed back and forth from requester, assigner or reporter. The content and length of this is user and application defined.

task*:* user defined array (address) providing a way for the worker to specify details about what kind of work it wants.
nbytes*:* number of bytes
timeout*:* FOREVER, or the number of milliseconds to wait.

integer waitForWorkRequest_thing(*task, nbytes, timeout*)

The return value is –1 (no worker requesting), or the node number of the worker making a request.

integer waitForWorkDone_thing(*task, nbytes, timeout*)

The return value is –1 if no worker has a completion message.

integer assignWork_thing(*node, task, nbytes*)

This is typically called by node 0, the task manager, to assign a unit of work to a worker.

integer getwork_thing(*node, task, nbytes*)

A worker, to receive assigned work descriptors, calls this.

integer workDone_thing(*node, task, nbytes*)

This is called by a worker to notify the task manager an assigned piece of work is completed. There is no requirement to call this, but if used, it will be returned as an item for waitForWorkDone_thing().

4 Implementation: Performance and Results

We have implemented the SHMOD library using subsidiary threads within each node, one for I/O and one for an MPI thread to handle all inter-node communication. The choice of MPI was an expedient one, as a high-performance version exists in the cluster environment. The peak measured point-to-point performance using Myrinet's proprietary "GM" library was 144 Mbytes/second. MPI gave us 142 Mbyte/second peak, both with large message sizes, which we always use for bulk data transfer.

We are in the process of running our simulations on the NCSA Itanium cluster using this described library in its out-of-core, that is, disk oriented flavor. The 512^3 simulation is complete, having produced 250 1.5 Gbyte time snapshots. This run was a warm-up, produced on 8 nodes augmented by NCSA with an extra 36 GB SCSI disk apiece for context and output storage. We had planned to be running on disks with performance as poor as 15 Mbyte/second; these additional disks more typically give us 30 Mbyte/second. We were able to begin the 1024^3 simulation, using these 8 nodes and an additional 8 with only the "as delivered" 8GByte system disk; these additional nodes retrieved context elements remotely from the more disk-gifted nodes. NCSA will soon augment enough nodes so we may compute the highest resolution run we have ever done on any hardware, the 2048^3-zone simulation.

At this writing, we are counting floating-point operations (by hand) in the new version of PPM we are using for these simulations. This PPM kernel is not purely vectorizable, performing different numbers of operations depending on the features of the fluid flow. We estimate we are within a factor of two

(downward, unfortunately) from the speed given by the purely vectorizable sPPM benchmark, which was 672 Mflops/processor. With more effort expended on kernel tuning, and as the Fortran compiler matures, we hope to better the simplified PPM time to solution with the new high-accuracy PPM kernel. Nevertheless, we are pleased to be producing the results we are on such early hardware & software.

5 Future Work

The first extension to pursue is application of the named object read or write to cluster communication rather than remote I/O, which we've previously referred to as "SHMON", "Shared Memory On the Network". This extension is actually a small one, which would rely on the host OS memory allocation and de-allocation routines in much the same way SHMOD relies on the hosts local file system. We would also add features for object migration, and for easily writing these objects to backing store (disk) on demand as a periodic check pointing operation. Another improvement to pursue would probably be replacement of the underlying MPI communication substrate with one that would permit freely dynamic process creation and destruction and fault detection.

6 References

[1] Alexandrov, A. D., Ibel, M. I., Schauser K. E., and Scheiman C. J. "Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System". *ACM Transactions on Computer Systems*, Vol. 16, No. 3, August 1998, pages 207-233.

[2] Bester, J., Foster, I., Kesselman, C., Tedesco, J., and Tuecke, S. "GASS: A data movement and access service for wide area computing systems". In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pp 78-88, Atlanta, GA May 1999, ACM Press.

[3] Carriero, N and Gelernter, D. "Linda and Message Passing: What Have We Learned?" Technical Report 984, Yale University Department of Computer Science, Sept. 1993.

[4] Chien, A., Lauria, M., Pennington, R., Showerman, M., Iannello, G., M. Buchanan, M., Connelly, K., Giannini, L., Koenig, G., Krishnamurthy, S., Liu, S., Pakin S. & Sampemane, G. "Design and Evaluation of an HPVM-based Windows NT Supercomputer". *The International Journal of High-Performance Computing Applications*, Vol. 13, No. 3, Fall 1999, pp. 201-219.

[5] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, W., Prost, J.-P., Snir, M., Traversat, B., and Wong, P. "Overview of the MPI-IO parallel I/O interface" In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1-15, April 1995.

[6] Feng, W., and Tinnakornsrisuphap, P. "The Failure of TCP in High-Performance Computational Grids". *Supercomputing 2000*, IEEE Press.

[7] Foster, I., Kohr, D., Krishnaiyer, R. and Mogill, J. "Remote I/O: Fast Access to Distant Storage" In *Proceedings IOPADS '97*, pages 14-25. ACM Press, 1997.

[8] Goux, J.P., Kulkarni S, Linderoth J., and Yoder, M, "An Enabling Framework for Master-Worker Applications on the Computational Grid", In *Proc. 9th IEEE Symp. On High Performance Distributed Computing*, 2000 IEEE Press.

[9] IBM Corporation, LAPI documentation: http://www.research.ibm.com/actc/Opt_Lib/LAPI_Using.htm

[10] Livny, M. "High-Throughput Resource Management", In Foster & Kesselman Eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999 pages 311-337.

[11] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. http://www.mpi-forum.org/docs/docs.html

[12] Myrinet Inc., http://www.myri.com

[13] J. Nieplocha, J., Harrison R. J., and Littlefield.R. J. "Global Arrays: A Non-Uniform-Memory-Access Programming Model For High-Performance Computers". *The Journal of Supercomputing*, 10:169-189, 1996.

[14] Nieplocha, J., and Foster, I. "Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations". *Proceedings of the Frontiers of Massively Parallel Computing Symposium, 1996.*

[15] Nieplocha, J., and Carpenter, B., "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems", *Proc. of 3rd Workshop on Runtime Systems for Parallel Programming of the International Parallel Processing Symposium* IPPS'99.

[16] Porter, D. H., Woodward, P., R., and Pouquet, A. "Inertial Range Structures in Compressible Turbulent Flows". *Physics of Fluids*, Vol. 10, Issue 1, pp. 237-245, January 1998.

[17] Sytine, I. V., Porter, D. H., Woodward, P. R., Hodson, S. H., and Karl-Heinz Winkler. "Convergence Tests for Piecewise Parabolic Method and Navier-Stokes Solutions for Homogeneous Compressible Turbulence". *J. Computational Physics*, 2000,Vol. 158, pp. 225-238.

[18] Thakur, R., Lusk, E., and Gropp, W. "Users Guide for ROMIO: A High-Performance Portable MPI-IO Implementation", Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 234, September 2000.

[19] Thakur, R., Gropp, W., and Lusk, E. "On Implementing MPI-IO Portably and with High Performance". *Proceedings of the workshop on Input/Output in Parallel and Distributed Systems*, 1999, Atlanta GA.

[20] Weissman, J. B, "Smart File Objects: A Remote File Access Paradigm". *6th ACM Workshop on I/O in Parallel & Distributed Systems*, May 1999.

[21] White, B. S., Grimshaw, A. S., and Anh Nguyen-Tuong, "Grid-Based File Access: The Legion I/O Mode", *9th IEEE Int'l Symposium on High Performance and Distributed Computing*, Pittsburgh Penn., August 1-4, 2000.

[22] Woodward, P. R., and Anderson, S. E. The sPPM Benchmark, 1995. http://www.lcse.umn.edu/research/sppm/README.html

[23] Woodward, P. R., and Anderson, S. E. "Portable Petaflop/s Programming: Applying Distributed Computing Methodology to the Grid Within a Single Machine Room". *Proceedings of the 8th International Symposium on High Performance and Distributed Computing*, Pittsburgh, PA, 1999.

[24] Woodward, P. R., Porter, D. H., Sytine, I., Anderson, S. E., Mirin, A. A., Curtis, B. C., Cohen, R. H., Dannevik, W. P., Dimits, A. M., Eliason, D. E., Winkler, K.-H., Hodson, S. W., "Very High Resolution Simulations of Compressible, Turbulent Flows".World Scientific, 2000.