Cooperative Data Sharing: An Architecture-Independent Interface for Implementing Parallel CFD Applications

David C. DiNucci
MRJ, Inc. / NAS Systems Division
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA 94035-1000
(415)604-4430
dinucci@nas.nasa.gov http://www.nas.nasa.gov/~dinucci

1.0 Introduction and Problem Description

More and more CAS applications are being converted to run on parallel architectures. For the most part, each application-architecture pair represents a separate porting effort. In order to amortize the cost of developing and porting these codes, their lifetimes must be extended by making a single source program portable among a large class of architectures, including both shared-bus architectures (sometimes called shared-memory architectures) and distributed-memory architectures consisting of separate nodes with local memory connected with networks with much higher latencies than memory busses (sometimes called message-passing architectures). Portable languages such as HPF can theoretically help to remedy this situation, but HPF is restricted in the types of algorithms which it supports, and has not demonstrated sufficient performance to date.

Currently, perhaps the most popular choices are to code these applications using PVM, a message-passing package developed to utilize heterogeneous clusters, or MPI[4], the message-passing standard still under development by an international forum. Message-passing, by definition, requires copying data from a sending process to a receiving process, and requires the sender to know the identity of the receiver. Message-passing is well-suited to distributed-memory architectures, since the speed of an application can often be greatly improved by copying data across the slow interconnect to bring it into the local memory of the processor which will be accessing it the most. The latency of this transfer is best hidden by moving large chunks at one time, and having the sender initiate the transfer towards the receiver before the receiver requires the data. Message-passing is not well suited to shared-bus architectures, because the overhead of the copy is often superfluous, and the constraint that the sender must know the identity of the receiver serves no purpose.

Lately, hybrid architectures, such as the SGI cluster (“DaVinci”) and Cray J90 cluster (“Newton”) at NAS, have become very common. These architectures consist of several nodes which communicate over relatively high-latency networks, with each node consisting of several processors which share local memory over a shared bus. Optimal processing on these architectures requires altering the communication methods used throughout the program, depending upon whether the processes which communicate are on the same nodes or different nodes. In this case, simply moving a process from one processor to another can radically alter the coding of the program.

2.0 Background and Other Approaches

Over the years, several approaches have been proposed to address the problems which arise from the differences between shared-bus and distributed-memory architectures. Three are especially noteworthy in this context: Linda[2], Reactive Kernel(RK)[1], and Distributed Shared Memory(DSM)[3].

Linda is described as a language, but it actually consists of a few new statements imbedded in standard C and/or Fortran: `in`, `read`, `out`, `readp`, and `eval`. Instead of communicating directly between processes, these statements support I/O to an abstract ubiquitous globally shared construct called a tuple space. One contribution of Linda is the ability for a producing process to make data available (by `out`ing it to the tuple space) without knowledge of which process will need it (i.e. `read` or `in` it) next, even on distributed-memory architectures. Some implementations of Linda also allow a process to effectively find data matching certain attributes in the tuple space. Although Linda is billed as an architecture-independent approach, it still depends on copying the same way that message-passing does.

The Reactive Kernel (RK) contains a subroutine library which supports a method of communication which can be efficiently implemented on both shared-memory and message-passing architectures. To communicate, a process allocates dynamic memory from a special communication area (using a function similar to the standard `malloc`), fills that memory with the data to be communicated, then uses RK to pass a pointer to that memory to another process, where it is queued. The act of passing the pointer relinquishes access to (i.e. effectively frees) the memory from the originating process. When the other process obtains the pointer from its incoming queue (via RK), it has full access to the memory until it frees it or passes the pointer to another process through RK. If a pointer must be passed across a high-latency interconnect to reach its destination, RK transfers the data along with the pointer, without user intervention. Although this communication technique avoids the extra copy required by message-passing on shared-bus architectures, it still requires the sender to know the identity of the receiver. It also requires the process initiating the communication to relinquish access to the communicated data as part of the communication, unlike message-passing, and requires each reading process to have its own copy of the data, unlike shared-memory.

Distributed Shared Memory (DSM) is a generic term describing an approach where standard shared-memory semantics are altered or restricted slightly in order to make them more efficiently implementable on architectures with high-latency interconnects. One such semantic restriction is called “release consistency”[5], which requires that accesses to shared regions take place only while the accessing process holds a lock to that region. This approach can overcome some of the drawbacks of RK -- e.g. multiple readers are allowed in some implementations (using “read” locks), and knowledge of the next process to access a region is not required when a lock is released. Unfortunately, this latter attribute, together with the lack of queues in the model, remove much of the capability for hiding latency in distributed-memory architectures, since data cannot be forwarded to the processor where it will be needed next. Instead, data is transferred over the interconnect only when a process attempts to acquire a lock to the data, so a process must often sit idle while the request travels the interconnect and the data returns over the interconnect. (Linda avoids this problem in some cases through careful compile-time analysis to determine where data

will be needed next when it is added to the tuple space.) Some forms of DSM also require some hardware support, though this can often be accommodated by fairly-standard virtual memory hardware.

3.0 The CDS Approach to Communication

The Cooperative Data Sharing (CDS) library implements communication in a way that achieves portability while retaining the advantages of the above approaches. At its core, communication in CDS is similar to RK. A user process allocates memory from a special area (called the "comm heap") using a CDS routine, moves the data to be communicated to that region using a sequential programming language (e.g. C or Fortran), then uses CDS to move the region pointer to a queue, making it available to another process. Also, as in RK, CDS will automatically make a copy of the data when the pointer is enqueued if the new process is across a high-latency interconnect. However, unlike RK, a pointer can be placed into and/or removed from any queue in any process. Thus, if a process knows the process which will use the data next, it will benefit by putting the pointer into a queue in that process, thereby preemptively moving the data across the interconnect between the processes (if there is one). If a process does not know the process which will need the data next, then it can place the data into any queue (e.g. a queue in itself) where the data will wait until some process later removes it. This latter case provides for the flexibility in Linda or DSM, when it is needed.

Unlike RK, a user process in CDS has a choice over whether to relinquish its pointer to a region when it puts the pointer into a queue. If it does relinquish the pointer, then the semantics of the transfer are similar to RK. If it does not relinquish the pointer, multiple processes (e.g. the process which originally allocated the region and any other process which takes the pointer from a queue) can end up accessing the region concurrently. This is beneficial if all sharing processes are reading the region, since on shared-bus architectures, overhead can be decreased significantly over message-passing or RK, which require a separate copy to be made for each reading process. However, in CDS, if some processes try to modify the region while others are reading, serious harm can result. For this reason, a user process is required to call a special CDS routine (`cds1_mod`) before making any modifications to a region. If no other process has a pointer to the region, `cds1_mod` returns immediately, with virtually no overhead. If at least one other process does have a pointer to the region when `cds1_mod` is called, `cds1_mod` will create a new copy of the region for the calling process. The end result is the implementation of a "copy-on-write" policy in software, similar to that used in some virtual memory systems.

CDS queues, called "comm cells" (or just "cells"), are more flexible than message queues in most other systems. A process can perform five different operations to a comm cell:

`deq`: Acquire and remove first pointer from queue
`enq`: Add pointer to end of queue, and optionally relinquish (i.e. effectively free) the pointer
`read`: Acquire first pointer from queue, but do not remove pointer from queue
`write`: Remove all pointers from queue, then perform `enq`
`zap`: Remove all pointers from a queue

As already described, any CDS process can perform any of these operations to any cell in any process. The cell is specified with the combination of a process ID and an integer cell ID. In general, the cell ID serves a very similar role to a tag in traditional message passing systems.

4.0 Heterogeneity, Copying, and Data Conversion

The routines described above work very well for those cases where all of the processes are running on processors with similar data formats, and the data structures being communicated are naturally created and modified in dynamic memory. For those other cases, where the data to be communicated must begin or end up in a process's private memory, and/or those cases where some processes use different internal data formats, CDS provides `copy` routines to help in copying and/or translating data before or after the mechanisms above have been used to communicate the bytes efficiently between processes.

Along with the source and destination addresses, the `copy` routines take a copy descriptor and a conversion ID. The copy descriptor describes the number, type, and displacement of each field and/or data structure to be copied. Unlike MPI type descriptors, a copy descriptor takes the form of a standard integer array, and the user constructs it without any help from CDS. The conversion ID is a simple integer code which describes how each type is supposed to be converted as it is copied. The CDS routine `transtab` takes any two process IDs and provides the proper conversion ID to convert between them.

Since it is relatively common to require a `copy` together with a communication, CDS provides two routines, called `send` and `recv`, which combine them. A `send` is equivalent to allocating a memory region on the comm heap, calling `copy` to move data into it (converting if necessary), calling `enq` to add the region pointer to a queue, and then `free`ing the pointer. The overall effect is very similar to an `MPI_BSEND`, and the same optimizations are possible. The `recv` function is defined similarly in terms of `deq`, `copy`, and `free`. Because of the semantics of the component functions, the `send` and `recv` functions are somewhat different from their MPI counterparts in that a `send` can move data to a queue in the same process, and `recv` can take data from a queue in another process.

5.0 Handlers (Active Messages)

CDS allows the user to specify a subroutine (i.e. "handler") for each cell, along with a high- or low-water mark. The specified subroutine will be called automatically whenever the number of pointers in the cell falls below the low-water mark, or grows above the high-water mark. These handlers can aid with load balancing by allowing the program to process messages whenever they happen to arrive, or to fill up outgoing queues which are becoming too empty. A very simple priority scheme and thread scheduler is provided to allow the user to specify which handlers should preempt others.

6.0 Dynamic Process Creation

CDS programs can dynamically grow and shrink during execution. A very simple and efficient model of process creation is supported, where there is no connection between the new child process and its parent until that connection is initiated by the child. This allows a parent to spawn several children in parallel, without separately waiting for each to start, and allows each child to perform any necessary initialization of its communication system before any other process knows that it exists.

7.0 Performance

Measurements were made on the NAS Davinci cluster, an SGI Challenge Array consisting of several nodes connected with HiPPI, FDDI, and Ethernet networks, and each node containing between two and eight fast SGI processors. The test program simply passes a region between two processes. One process `enq`s the region into a cell in the other process, the other process `deq`s it and repeats the cycle.

The current implementation of CDS takes approximately 25μsec (best case) to pass a region between two processes on the same node, so about 40,000 regions (i.e. pointers) per second can be passed regardless of region size. SGI's optimized version of MPI can send a 1-byte message in less time, but because of the additional overhead of copying, MPI takes longer than CDS to send messages of more than 250 bytes or so in this case.

In those cases where CDS performs copying and/or off-processor communication, the performance of the current implementation is not exceptional. Latency is approximately 750 to 1100μsec over all networks (Ethernet, FDDI, HiPPI). CDS achieves about 1MByte/sec over Ethernet when passing regions of 10KBytes, 9.3MBytes/sec over FDDI when passing regions of 100KBytes, and 27.5MBytes/sec over HiPPI when passing regions of 1MBytes.

It is expected that these times and latencies can be improved with further optimization, especially by utilizing the same HiPPI bypass technology and cache/memory management techniques used by SGI's version of MPI on the same hardware.

8.0 Summary and Future Plans

Altogether, there are 29 routines in the CDS library described above, called CDS1. It is inherently more efficient than message-passing for shared-bus architectures in many cases, and is as efficient as message-passing in other cases. CDS1 already provides much of the functionality envisioned for MPI1's follow-on, MPI2, even though CDS1 has fewer than one-quarter as many functions as MPI1.

From the outset, CDS1 was envisioned as the basis for other tools. One other tool scheduled to be built atop CDS1 is CDS2, a higher-level subroutine library to provide additional MPI-like constructs (e.g. communicators and more collective functions) to ease the porting effort for those already using MPI. Some of this design of CDS2 has already been completed.

Further funding will foster additional important development. A more robust implementation will be necessary. Some CFD applications (e.g. NAS parallel benchmarks) should be coded in CDS to demonstrate its utility to the CFD community. Additional optimization, such as that currently present in MPI implementations, should find its way into CDS. CDS should be targeted and optimized for more architectures. CDS2 should be fully designed and implemented, possibly as a separate concurrent project with CDS1. Ultimately, CDS should be demonstrated as a viable architecture-independent interface, and it should be promoted as an open standard.

9.0 Further Information

For more information on CDS, please consult the web page at

`http://www.nas.nasa.gov/NAS/Tools/Projects/CDS/`

10.0 References

[1] W. C. Athas and C. Seitz, “Multicomputers: Message-Passing Concurrent Computers,” Computer 21(8), pp 9-24 (August 1988)

[2] D. Gelernter, “Generative Communication in Linda”, ACM ToPLaS, 1 (1985), pp 80-112

[3] K. Li and P. Hudak, “Memory Coherence in Shared Virtual Memory Systems”, ACM Transactions on Computer Systems, 7(4), pp 341-359, November 1989.

[4] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, “MPI: The Complete Reference”, MIT Press, 1994, ISBN 0-262-57104-8

[5] P. Keleher, A. Cox, S. Dwarkadas, W. Zwaenepoel, “An Evaluation of Software-Based Release Consistent Protocols”, to appear in Journal of Parallel and Distributed Computing.