Efficient Communication Mechanisms for Cluster Based Parallel Computing[1]

Al Davis, Mark Swanson, Mike Parker

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA
ald@cs.utah.edu, swanson@cs.utah.edu, map@cs.utah.edu

Keywords: communication latency, protocols, network interface, cluster of workstations, message passing)

Abstract

The key to crafting an effective scalable parallel computing system lies in minimizing the delays imposed by the system. Parallel algorithms must communicate frequently. The communication delay between sender and receiver is a system-imposed *latency*. The existence of relatively inexpensive high performance workstations and the emerging availability of high performance interconnect options provide compelling economic motivation to investigate NOW/COW (network/cluster of workstation) architectures. However, these basic components have been designed for generality. The nodes in NOW/COW workstations are connected by longer physical wire paths than found in special-purpose supercomputer systems. Both effects tend to impose intractable latencies on communication. Another system-imposed delay arises from the overhead of sending and receiving messages. This overhead can come in several forms, including CPU occupancy by protocol and device code as well as interference with CPU access to various levels of the memory hierarchy. Access contention becomes even more onerous when the nodes in the system are themselves symmetric multiprocessors. Additional delays are incurred if the communication mechanism requires processes to run concurrently in order to communicate with acceptable efficiency. This paper presents the systems approach taken by the Utah Avalanche project which spans user level code, operating system support, and network interface hardware. The result minimizes the constraining effects of latency, overhead, and loosely coupled scheduling that are common characteristics in NOW-based architectures.

1 Introduction

Minimizing communication latency in parallel systems is a critical design component since parallel programs communicate frequently. Reduced latency permits increased levels of parallelism between finer grained components. The result is that communication latency is the key to achieving high levels of aggregate system performance and reasonable scalability. MPP (massively parallel processors) supercomputers achieve very low communication latencies and are capable of very high sustained bandwidths. Some MPP systems also achieve enhanced performance by providing direct support for maintaining synchronization between the communicating programs. Even though today's MPP systems are all based on

[1]This work was supported by SPAWAR contract #N00039-94-C-0018 and ARPA order #B990.

commodity microprocessors, the logic and packaging to support the communication and synchronization mechanisms are specialized for the low-volume supercomputer market. The result is that MPP systems are too expensive.

The existence of relatively inexpensive high performance workstations and the emerging availability of high performance interconnect options provide compelling economic motivation to investigate cluster based multiprocessor architectures. If performance scalability can be achieved then a wide range of performance options can be selected to suit the customers needs. However, several aspects of commercial workstations and networks are significant handicaps from the standpoint of low-latency communication. Workstation I/O architectures and networks have been designed for generality rather than performance. These problems span the entire system framework from code path lengths in applications, libraries, operating systems and device drivers to physical delays in the processor, memory system, I/O controllers, network interface, and the interconnect fabric's switches and wires. It is clear that these latency problems must either be drastically reduced or effectively hidden if these clusters are to achieve acceptable levels of performance. The approach described here is an interesting hybrid in that it provides both significant latency reduction as well as a mechanism for hiding some of the remaining delays.

In clusters of workstations (COWs), a large fraction of the latency can be reduced with improved software message passing protocols. Further reduction can be achieved by improved network interfaces that accelerate these protocols and tightly integrate the interconnection fabric with the processor and its memory system. This paper describes the approach taken by the Utah Avalanche project. The approach includes a new set of efficient message passing protocols and the associated applications programming interface, direct hardware interface support for these protocols, improved coupling with memory subsystems of the sender and receiver, and improved cache performance. The Widget is the Utah Avalanche network interface component which couples Hewlett-Packard's multiprocessor workstation technology to Myricom's high speed interconnect fabric to achieve user to user message latencies that are less than 4 microseconds. This includes using a standard Unix operating system with minor modifications and does not sacrifice the traditional security of TCP/IP based methods. An additional benefit of the approach is that the protocol permits asynchronous message passing applications to be self synchronizing which alleviates the need for overhead intensive gang scheduling operations. The paper restricts itself to message passing performance issues even though the Widget provides hardware support to increase the efficiency in distributed shared memory operations as well.

The paper elaborates on the nature of the latency, overhead, and scheduling problems to provide background for certain design choices. The design of the message passing components of the Avalanche system are then presented. Although this is not primarily a measurement paper, simulation results are then provided to quantify the utility of the design decisions. A survey of related work is then presented followed by concluding remarks.

2 Message Latency

Quantifying message latency is a non-standard process which varies with the focus of the particular experiment. The concern here is with total system performance. Therefore message latency is defined to be the time between a sending application process initiating the send to the time the receiving application process can actually issue instructions which use the information contained in the message. Message latency is therefore the combined influence of several system components:

- **Interconnect Performance:** This includes the fall-through delay, delays imposed by connection set-up and tear-down, propagation delay of the wires or fibers, and the transit delay imposed by any logical components in the transmission path.
- **Interconnect Behavior:** This includes overheads associated with maintaining the quality of service for aspects such as reliability, flow control, access control, packet size, and contention delays.
- **Network Interface:** Delays are influenced by the amount of software aid required to transport data in and out of the interface.
- **Memory System Effects:** In NOW architectures, the message data on the send side tends to reside in the cache while on the receive side the data is only useful when it is either in the cache or the processor registers. There is often a considerable delay incurred in moving the data between the cache and the network fabric.
- **Protocol:** The generality of the protocol choice translates proportionately into software overhead at both the sender and receiver. This includes a number of potentially hidden delays such as context switching, memory bus contention, cache miss penalties in addition to the protocol code path lengths.

There are numerous design decisions that can be made in each of these five areas. For NOW/COW systems utilizing commodity fabrics and processors, many of the choices are beyond the control of the system architect. The choice is restricted to choosing the appropriate component set based on cost and integration capability. The challenge for NOW/COW systems is to maximize performance while utilizing commodity components in order to minimize cost. The generality and wiring ease of shared media networks results in intractable latencies under heavy loads. The choice of a high capacity point to point interconnect fabric is currently the best option.

The specific Avalanche approach is based on the use of Hewlett-Packard's symmetric multiprocessor (SMP) workstation and Myricom's high performance interconnect technologies. These are then coupled by a custom network interface called the *Widget.* Hewlett-Packard provides one to four processor workstation platforms based on their HP-7200 or PA-8000 microprocessors [5, 7]. While these processors vary significantly in their internal architecture, they both use the Runway [3] bus as their main memory bus. The Runway bus is a 64-bit, split transaction bus which supports cache coherence in the multiple processor SMP configurations. For HP7200 based systems the clock speed is 120 MHz with a maximum sustainable bandwidth of 768 megabytes/second in 4-way systems. The results presented in this paper are based on the HP7200 and the additional performance possibilities of the PA-8000 platform are ignored.

Myricom's Myrinet-II [2] fabric runs at 160 MHz and uses bidirectional, byte-wide, data paths in each link to provide a source routed wormhole point to point interconnect fabric. Switching is done using 8 or 16 link crossbar switches which have a fall-through time of 110 ns. The Myrinet also provides reliable message delivery, as well as internal error and flow control. The choice of source routing permits arbitrary cluster topologies and hence eases both wiring and incremental expansion costs. While the Runway bandwidth is 4.8 times that of the Myrinet, the system is reasonably well balanced since the Runway must support cache to cache, and cache to memory transactions as well as I/O.

The main defect of the Myrinet technology for NOW/COW systems is the network interface card (called LaNai) which incurs several microseconds of latency at both the send and receive sides. This is exacerbated by additional overheads since the LaNai connects to the node's I/O controllers rather than directly to the memory bus. This precludes coherent fabric transactions and does not efficiently support the improved protocols described in this paper.

The design of the fabric interface can have a dramatic effect on both latency and overhead[12]. The overhead incurred by the CPU interactions to send or receive a message are usually difficult or impossible to eliminate from the basic latency. The Avalanche choice has therefore been to create a custom network interface capable of directly coupling the Myrinet to the Runway bus. The Widget permits improved memory locality by providing cache coherent message handling and direct hardware support for improved protocols. From the perspective of other processors and the memory controller in the SMP node, the Widget looks just like a processor. The downside of this approach is that the node loses a processor slot in order to achieve efficient communications. This does not need to be the case in general, but the current decision is imposed by the existing commercial packaging choices.

3 Communications Overhead

As with latency, overhead results from many factors, including protocol complexity, fabric interface design and capabilities, and protocol implementation. For parallel applications, the largest overhead contributions come from data movement and from polling. Protocol design and implementation dictate the number of copy operations, with a practical minimum of two copies to move the data into and out of the communications fabric. Interface design impacts the copy overhead. For example, DMA-capable interfaces eliminate the instruction execution overhead of these copies. Further integration of the interface into the memory system can reduce the cache, bus, and main memory overheads. Additionally, an interface which requires polling of device registers in order to check for message arrival is more latency intensive than an interface which permits notifications to be posted as a cachable memory object.

4 Scheduling

Scheduling can impact parallel application performance in a number of ways. Delays in scheduling tasks can delay all tasks as they wait for synchronization. In a communications model where receiving tasks are required to extract messages from the fabric, delay in scheduling a receiving task can delay the sender of an otherwise non-synchronized communication. These effects have naturally led to concern that clusters of workstations running independent operating systems will exhibit poor parallel performance. Results from our own simulations [2] and from work done by Paikin, et. al. at the University of Illinois at Urbana show that it may be unnecessary to to modify the workstation kernels to perform coordinated scheduling.

Under high loads and with appropriate communication mechanisms such as those described here, distributed parallel applications show a tendency to *self-schedule*. This improves performance significantly in terms of time-to-completion for single parallel applications and in terms of throughput for multiple competing applications. An appropriate communication mechanism should provide a blocking message reception primitive for cases when the message is not yet available. It should also provide an inexpensive primitive for the case where the message is available. This form of co-scheduling is imprecise since send and receive processes may not be running concurrently. Therefore the communication mechanism must efficiently address the case where the receive process is not running. This will remove the synchronization requirement for the receiving process, and support asynchronous communications. An additional constraint is that this capability must be provided in a fashion that does not incur additional scheduling overheads elsewhere in the system.

5 Message Passing in Avalanche

Message passing in Avalanche takes a complete system approach to minimizing latency and overhead, and to reducing the importance of coordinated scheduling. Lightweight, sender-based protocols[13] are used to reduce the software components of latency and overhead, and to allow simple hardware acceleration of common operations. Operating system involvement is retained in connection establishment and in message transmission. This permits protected use of the fabric by multiple processes. Operating system involvement in message reception is optional, at the discretion of the application. A custom network interface is used to lower overhead and reduces processor involvement to transfer data between the fabric and the memory for both transmission and reception. Additional latency and overhead improvement is achieved by connecting the network interface directly to the memory bus rather than via an I/O adapter.

5.1 Sender-Based Protocols

The key concept of sender-based protocols[11] (SBPs) is that the sender manages a reserved buffer in the receiver's memory and directs placement of messages within that buffer. The sender can rely on an arriving message being placed within the buffer at the location the sender has specified. This means that, given a reliable network, a message that is successfully sent will be received, since neither the network or the receiving processor/interface will drop the message.

The Avalanche implementation of SBPs is called *Direct Deposit*[10] (DD); it provides system call-based connection establishment and message transmission. Much of the security and safety overhead is isolated to connection setup time and is provided by going through the kernel. Since message transmission requires access to the communication interface, use of a system call allows implementation of safe and

[2] Detailed measurements will be provided in the final paper. They have been deleted here due to space restrictions.

atomic shared access to that interface. It also complements the security of OS-mediated connections by allowing the kernel to limit a process' access to only its own connections. While mechanisms to allow "secure" user-level transmission via a shared interface have been proposed, they often rely on added interface complexity and/or virtual memory mapping tricks. The latter approach seems a poor choice at a time when TLBs already appear inadequate to map reasonable portions of application memory.

DD does support user mode message reception. On message arrival, the Widget writes a *notification object* in a connection specified queue. This queue can be in kernel memory or in user memory. Message buffers can likewise be in kernel or user memory. When both are in user memory, a completely user-mode receive is achieved. A system call is provided, however, that allows a process to sleep and wait for the arrival of a message notification on a specified queue.

A key to the performance of DD and the Widget is that receive buffers and notification queues must reside in *wired* memory. Thus, the Widget can write to them when a message arrives. They need not be mapped by the TLB, as the Widget performs memory operations using physical addresses. The use of wired memory may be viewed by some as non-scalable. The cost of the extra DRAM required by this approach is a minor fraction of the cost of the system. It is a very small price to pay for a significant increase in performance.

5.2 The Network Interface

As seen by the CPU the Widget provides a simple interface: both DIO (processor mediated) and DMA transmissions are supported. The CPU enters a transmission request block (TRB) onto a memory queue and then issues a single write to a device register to initiate transmission. The CPU operations on the TRB should all hit in the cache after at most one initial miss. The Widget then reads the entire TRB in one bus transaction. Completion of transmission is signaled via a status bit in the TRB. Given a reasonable number of TRBs, the Widget will be finished with a TRB before the software needs to reuse it; thus the software only incurs a single cache miss to determine that the TRB has been processed.

The Widget and CPU generally do not interact at all on message reception. As described earlier, the Widget writes a notification into memory when a message arrives. In some cases the receiver will block waiting for message arrival notification. A status bit in the notification indicates that the Widget should interrupt the CPU in this case. This is the only time in normal reception that the Widget interacts directly with the CPU.

5.3 Integrating the Network Interface into the Memory Hierarchy

The Avalanche Widget is a system memory bus resident interface. It operates as a peer with the CPU(s) in maintaining memory coherence. Both incoming and outgoing message data are moved into/out of the memory system using coherent bus transactions. The software need not flush or purge cache lines to ensure data consistency. This reduces both overhead and latency in the CPU, its cache, the bus, and the memory system.

Incorporated into the Widget is a 128K SRAM buffer called the Shared Buffer (SB) used to stage outgoing messages and to cache incoming message data. Outgoing messages are actually *pipelined* through the SB in units of 128 bytes (4 cache lines) at a time. For long messages, the transmission of data can be overlapped with obtaining following data from the memory system. Variations in the time to acquire data are *smoothed out* by this mechanism, allowing the actual Myrinet interface to see a steady stream of data.

For incoming messages, data is stored in the SB, and the Widget asserts *ownership* of the affected cache lines. The Widget then serves as a fast source for subsequent CPU accesses to these lines. Another benefit is that buffering in the SB allows data to be moved out of the Myrinet without waiting for the main memory to absorb the data. Once again, bus and memory bandwidth consumption are reduced, as is cache miss latency as seen by the CPU when it accesses the data. Some messages are too long to be stored in the SB. In this case, the SB still serves to smooth out the impacts on the system.

6 Performance Results

The fundamental performance of communication primitives of an Avalanche system based on 120 MHz HP 7200 processors and a 160 MHz Myrinet are examined here. Whole-program results are available, but space does not permit reporting them here[3].

6.1 Simulation Environment and Methodology

All of the results reported in this paper were obtained from execution-driven simulations using the Paint simulator[9]. Paint simulates the HP PA RISC 1.1[6] architecture and includes an instruction set interpreter and detailed cycle-level models of a first level cache, system bus, and memory system similar to those found in HP J-class systems. In addition, it contains a detailed cycle level model of the Widget and a simple Myrinet simulator that models contention only at node inputs. For these particular experiments, a simple processor is modeled at 120 MHz, with non-blocking loads and stores. Most instructions (except floating point) take one instruction. The cache is direct mapped (virtually indexed), 128 KB with 32 byte lines. It provides single-cycle loads and stores, a one-cycle load-use penalty, and data streaming. The Runway bus is also modeled at 120 MHz. The memory system contains 4 banks and a 17 entry write buffer. The minimal cache miss serviced by memory takes 39 cycles. The Myrinet is modeled at 160 MHz (giving a nearly 160 MB/second bandwidth); fallthrough time for the single switch is 13 processor cycles (approximately 108 ns.), propagation delays are 1 cycle on each side of the switch.

The simulation environment includes a small, BSD-based kernel used to provide the device driver and system calls which initialize the Widget, set up connections, and perform message send operations.

6.2 Latency and Overhead

Latency is measured by starting programs on two nodes. One program establishes a communication endpoint and enters a loop polling for incoming messages. The other program connects to the endpoint

[3]They will be included in the final paper.

established by the first and then enters a loop wherein it waits to ensure that the receiver will be polling. It then sends a 32 byte message using programmed I/O. Time is measured from before the sender's system call to send the message until the receiver has discovered the message's arrival and touched the first data word, causing the cache line to be brought into first level cache. The average latency is 463 cycles or 3.85 microseconds; the range is 429 to 569 cycles over 16 messages, due to cache misses and the nature of the polling loop. Corresponding times when the receiving Widget does not cache the data, but rather writes it directly to memory, are 503 cycles (4.19 microseconds) average and range 466 to 605 cycles.

To illustrate receiver overhead, a similar experiment is used, except that the receiver delays long enough for all 16 messages to arrive, thereby allowing it to receive all 16 with no waiting. The time between messages is a close approximation to message reception overhead, ignoring second order effects such as bus/cache contention incurred as the messages arrive at the node. The average receiver overhead is 119 cycles (approximately 1 microsecond); the range over 16 messages is from 95 to 205 cycles, due to cache effects.

To illustrate sender overhead, the sender is changed so that it does not delay between sending messages. In this case, the time between messages is a close approximation to message transmission overhead. The average sender overhead is 170 cycles (1.4 microseconds); the range over 16 messages is from 148 to 253 cycles, due to cache effects. For a 128 byte message the average increases to 204 cycles.

One last performance-critical operation is polling time. A simple poll for a message that has not yet arrived takes 65 cycles if it causes a cache miss on the message arrival notification structure, which is likely on the first poll, and 34 cycles on subsequent polls as long the notification remains in cache.

7 Related Work

Many groups are currently researching efficient messaging for workstations. The Hamlyn[4] effort at Hewlett-Packard Labs is similar in many respects to the Avalanche approach. It utilizes a sender-based protocol, HP Runway-based workstations, and the Myrinet interconnect. As their interface, however, they use a LaNai positioned on an I/O bus rather than the system bus. They report a one-way message latency of 12.7 microseconds with a large portion of the time spent in the interface processor and in high-latency bus translation hardware.

Illinois Fast Messages[8] is another I/O-bus resident system, in this case using SPARCstations and Myrinet. Their protocol uses a more traditional buffering approach, and this, together with their use of the Lanai interface and position on an I/O bus result in a small message latency of 32 microseconds.

U-Net[1] attempts to achieve many of the same goals as Avalanche, though with a constraint of using completely off-the-shelf hardware. They arrive at many of the same conclusions, such as the retention of system call interface for message transmission. They also report interesting performance effects resulting from fabric interface design. As with the other projects surveyed, positioning the network interface on an I/O bus and the use of a processor in the fabric interface limit the latency to approximately 32 microseconds for a one cell ATM message.

None of the surveyed projects addressed memory hierarchy effects in any depth. This is likely due to their positioning the network interface on an I/O bus, which limits their options. I/O bus performance will likely continue to improve more slowly that CPU and memory bus performance. The memory hierarchy effects of messaging will become even more pronounced. The performance impact of positioning network interfaces solely on I/O busses, which is already significant, will become even more compelling.

8 Conclusions

This paper has specifically described a set of design choices that have been made by the Utah Avalanche project which result in very low-latency message passing that is appropriate for NOW and COW style multiprocessing architectures. The approach, while specific to a particular commercial processor and fabric choice in the Avalanche implementation, is not a vendor specific architecture choice. It takes advantage of the fact that modern microprocessors are being designed to be used in both uniprocessor and small-way SMP product platforms. The primary feature that supports this capability is the provision of a system or memory bus that supports cache coherent transactions between the processors and the memory system. By providing an interconnect network interface that attaches directly to the coherent system bus, it is possible to significantly reduce the software overhead associated with message communication. The paper has also described a new efficient communication protocol (DD) which further reduces message overheads and which provides the additional benefit of permitting the system to be insensitive to task synchronization delays. These delays are imposed by systems which only support synchronous message transactions. The Avalanche network interface has also been designed to provide direct hardware support for these new protocols which significantly reduces software overheads on both the send and receive side while maintaining the level of security that is expected from existing TCP/IP based systems. The result is that message latency is reduced to approximately 4 microseconds and the need for overhead intensive gang scheduling mechanisms is unnecessary. This represents an improvement of 2 orders of magnitude over a choice which uses the same processor configuration and conventional TCP/IP protocols on an *unloaded* ethernet fabric.

References

[1] Basu, A., Buch, V., Vogels, W., and von Eicken, T. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[2] Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J., and Su, W.-K. Myrinet – A Gigabit-per-second Local-Area Network. *IEEE MICRO 15*, February (February 1995), 29–36.

[3] Bryg, W., Chan, K., and Fiduccia, N. A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers. *Hewlett-Packard Journal 47*, 1 (February 1996), 18–24.

[4] Buzzard, G., D.Jacobson, Mackey, M., Marovich, S., and Wilkes, J. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation* (October 1996).

[5] Chan, K., Hay, C., Keller, J., Kurpanek, G., Schumacher, F., and J.Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal 47*, 1 (February 1996), 25–33.

[6] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[7] Hunt, D. Advanced Performance Features of the 64-bit PA-8000. In *COMPCON '95* (1995), pp. 123–128.

[8] Paikin, S., Lauria, and Chien, A. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '88* (1995).

[9] Stoller, L., Kuramkote, R., and Swanson, M. PAINT- PA Instruction Set Interpreter. Tech. rep., University of Utah - Computer Science Department, March 1996. Also available via WWW under http://www.cs.utah.edu/projects/avalanche/paint.ps.

[10] Stoller, L., and Swanson, M. Direct Deposit: A Basic User-Level Protocol for Carpet Clusters. Tech. Rep. UUCS-95-003, University of Utah - Computer Science Department, March 1995. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

[11] Swanson, M., and Stoller, L. Low Latency Workstation Cluster Communications Using Sender-Based Protocols - Computer Science Department. Tech. Rep. UUCS-96-001, University of Utah, March 1996. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

[12] Thekkath, A., and Levy, H. Limits to Low-Latency Communications on High-Speed Networks. *acm Transactions on Computer Systems 11*, 2 (May 1993), 179–203.

[13] Wilkes, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, Nov. 1992.