1. ScaMPI – Design and Implementation

L.P. Huse, K. Omang, H. Bugge, H. Ry, A.T. Haugsdal, and E. Rustad

Scali AS, Norway
Email: {lph,knuto,hob,hwr,ath,eir}@scali.no

1.1 Introduction

MPI (Message Passing Interface) [8] is an established and de-facto standard for information exchange based on the message-passing paradigm. MPI was standardized by the MPI Forum in 1996. Academia, industry, and vendors of high-performance computers drove the effort.

The MPI has a rapidly growing community as a standard user API (application programming interface) for parallel programming. Today, MPI is the preferred API for portable, parallel programs, and the success of the standard can be illustrated by applications running on both shared and distributed memory systems. Examples of the former are systems from SGI and Sun, whereas IBM, Cray (now SGI), and Scali deliver systems adhering to the latter category. Applications written using MPI are deemed very portable, and they can easily be ported between shared and distributed memory systems. Thus, seen from an ISV (independent software vendor), a message passing application is more portable than an application using the shared memory paradigm, since the message passing application can run on either shared or distributed memory systems.

In the rest of the chapter the design and implementation of ScaMPI, Scali's high performance MPI implementation is presented. Key technical achievements of ScaMPI are low latency, high bandwidth and flexibility of transport medium as well as options for speeding up application performance within SMPs by allowing the use of threads. The programming environment for ScaMPI provides various built-in options for debugging and tuning. In addition ScaMPI is integrated with powerful third party software. Performance of important ScaMPI primitives are discussed in light of recent performance measurements. These measurements also document excellent scalability of ScaMPI for up to 96 inexpensive dual CPU nodes.

1.2 Scali Systems

Scali systems use SCI as interconnect. To make affordable systems, Scali has chosen to use standard I/O (input/output) buses, such as PCI [14] and Sbus [6], as attachment point to the SCI interconnect fabric. SCI has specified cache coherency as an option, but since the I/O buses for most workstations

are decoupled from the main memory bus by an I/O bridge (see Figure 1.1), the I/O bus cannot intercept a processor accessing the local memory. This

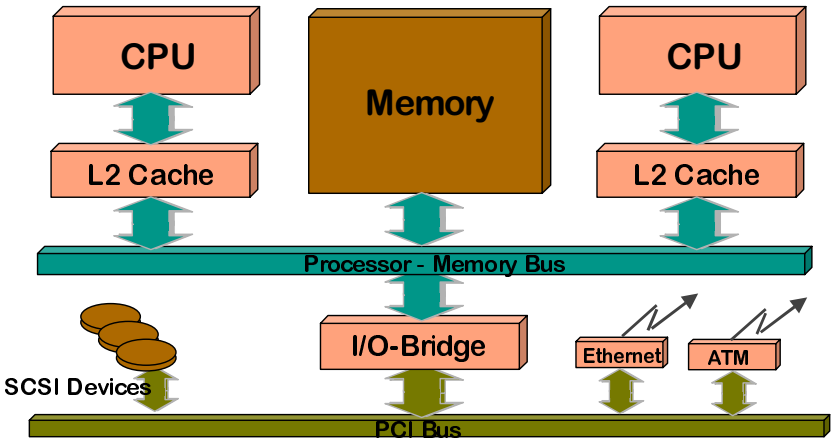

Fig. 1.1. Block diagram of a node

inhibits a global cache coherent memory model to be implemented in hardware. A clear benefit of using the I/O bus as attachment point is that the I/O buses are standardized. For the IHVs (independent hardware vendors) this means having a larger market for their products, and larger volumes again imply lower prices. Also, the evolution of new generations of I/O buses, e.g., faster clock frequencies, wider buses etc., is less frequent compared to the evolution of processors and their accompanying buses. As a consequence, the cost for peripheral boards tends to be low compared to special, proprietary hardware components.

1.3 The SCI Memory Model

The cache coherency in SCI is documented in detail. The memory consistency model on the other hand, is left to the implementers. Thus, any memory consistency model, such as the sequential, processor, weak, or release consistency model might be implemented. This applies to systems using either the I/O or the cache coherent processor bus as the attachment point. Software running on SCI based systems, must therefore either be explicitly or implicitly aware of the memory consistency model provided by the system.

Systems using the I/O bus as the attachment point are in most cases distributed memory systems, where each node runs its own instance of the operating system. Such systems, e.g., SCI based, are able to provide a shared address space programming model, where portions of the virtual address

space of one process can be made visible to a process running on another node. The conceptual simplicity of this model compared to the traditionally layered ISO/OSI model can easily be illustrated. For two user level processes running on different nodes to communicate, both need to map the same SCI shared memory segment into their virtual address space. The SCI shared memory is for performance reasons always physically located on the receiver side. For two processes to communicate, the sender process writes data to the remote memory segment and the receiver reads the data from local memory, without any system calls or expensive protocol processing.

Middleware for this shared address space model must be explicitly aware of the underlying memory consistency model, which is influenced by the characteristics of the processor, the I/O bridge, the SCI adapter, and the SCI interconnect fabric. For example, common hardware techniques for performance enhancements, such as write buffering, write combining, and prefetching might be implemented in multiple of these hardware components. To Scali, this illustrates two problems with the shared address space model: the user must have an intimate knowledge of the hardware components, and very few applications are written for the shared address space model.

1.3.1 Coordinating Use of Shared Locations

There are several ways of using shared locations to exchange data as needed to implement message passing. One well known technique is to use the concept of *critical regions* where only one process at a time has access to a particular shared resource. An example from MPI is how to implement buffer allocation. With mutual exclusion, an implementation can allow multiple sender nodes to allocate receive buffers at a particular remote node from the same buffer pool. This technique has been frequently employed in MPI implementations for SMPs. The approach is simple and makes it easy to implement space efficient resource management policies and can be implemented with efficiency between processors on the same memory bus. However, as the number of communicating processes grows, contention for the involved locks may hurt performance.

Efficient implementation of mutual exclusion locks requires atomic memory operations. Atomic operations such as compare-and-swap and various forms of fetch-and-op are available (or can be built with the available basic atomic primitives) in most modern system bus architectures. The SCI standard specifies a number of such operations, but only a very limited atomic operation support (fetch-and-increment-by-one) is available for the current Dolphin PCI/SCI implementation [3].

Efficiency is also complicated by the fact that remote loads (fetching data over the network) are an order of magnitude slower than local loads. To achieve the atomicity provided by the PCI/SCI hardware in absence of properly implemented locking primitives on the PCI bus, all accesses to the lock

memory must go through the PCI/SCI hardware, i.e., they are remote accesses from a performance perspective [11].

A simple solution where locks are avoided is to make access to shared locations disjoint with respect to stores, that is, only a single process has the right to store to a particular location. For the example on dynamic buffer allocation in the receiver, this means that there must be separate buffer pools for each sender at each receiver. Thus this may look like a speed-at-cost-of-space trade-off. However, with the option of allocating the buffer pool for a particular sender at a particular receiver only when needed, the extra buffer capacity spent can be kept low.

1.3.2 Ensuring Safe Data Transport in SCI - Checkpointing

Usually when transferring data from one node to another over a SCI network, the data arrive at its destination fast and accurate. However, in a multi-node shared memory environment there is always the possibility of nodes being temporary unavailable (e.g., due to high priority OS calls), data alteration in the network due to electronic noise (detected by CRC checks), the I/O bus may be occupied with other high priority traffic etc. All of these events are detected in Scali systems, and those that are related to the SCI network corrected by the SCI driver. To make certain that the data arrive correctly over the network, checkpointing needs to be employed.

Checkpointing is a common programming technique used in systems where dynamic errors may occur, e.g. in shared memory and database systems. Before and after all operations sufficient status information is gathered to check if the operation was completed successfully. If the checkpoint fails the effects of the faulty operation has to be nullified and the operation has to be repeated. For SCI, the checkpoint procedure is initiated by first flushing all data on to the network. The checkpoint state is then derived from the state of the driver and an interrupt counter. If the checkpoint state changes during a data transfer, the data has to be retransmitted. A shadow of all necessary driver information is mapped into user-space for fast checkpointing. An example of using checkpointing in shared memory programming is given in Figure 1.2 Section 1.5.1.

1.3.3 Shared Address Space Programming without the Drawbacks

To enable users to exploit the benefit of the shared address space model without detailed hardware knowledge, standardized APIs are provided to the application programmers. A large existing base of applications can then be used directly (without modification of even old "dusty deck" applications), just by recompiling. The applications are by this able to communicate efficiently, without being burdened by excess copying, system calls, interrupts etc. as would have been necessary if a typical software stack of the traditional

ISO/OSI model had been used. Today, MPI [8], PVM [5], Fast Messages [12], and Split-C [2] are also available on Scali systems. ScaMPI is Scali's own high performance implementation of MPI, the others are available through various academic institutions.

1.4 ScaMPI Design Goals

The following goals were set forth in the design process of ScaMPI:

Scalability: System size ranging from one to hundreds of nodes should be supported.

Low latency: We aimed at message latency around 10 µs for exchanging MPI messages from user level to user level. The latency of collective MPI operations should grow with $O(log(N))$, where N is the size of the system.

High bandwidth: Point-to-point bandwidth should be close to the theoretical maximum for the actual implementations. The bandwidth available to each node performing MPI collective operations should be constant and not be reduced with increased system size.

Fault tolerance: The SCI interconnect fabric might be subject to errors, such as CRC errors, cables being unplugged etc. Such transient errors should be transparent to applications. For example, it should be possible to exchange cables while an application is running, without affecting it except for reduced performance. Another example is that it should be possible to change the routing function in the SCI interconnect fabric, i.e., the communication paths, transparent to the running application.

Flexibility of transport medium: Although SCI shared address space was intended as the primary transport medium, we aimed at leveraging this implementation and support true shared memory as the transport medium for MPI processes communicating on the same SMP node. The selection of the actual transport medium should be automatically and transparent to the user.

User friendliness: To ease application development we aimed at providing different levels of startup procedures to accommodate different requirements, such as debugging, profiling, logging etc.

Thread-safe implementation: ScaMPI must support different mappings of MPI processes to hardware resources. In a one-to-one mapping each MPI process is mapped to its own CPU, while in the one-to-many model each MPI process is mapped to a set of CPUs - typically all the CPUs in a node. In the one-to-many model, explicit multithreading programming or an automatic parallelization tool is used to efficient exploit all system resources. Here different threads constituting a single MPI process might simultaneously request services from the MPI library. Thus, ScaMPI as well as the SCI middleware had to be designed thread-safe in a way that enables a high level of parallelism.

1.5 ScaMPI Implementation

ScaMPI was designed to take advantage of SCI's shared address space architecture. The focus has been on utilizing those features ensuring the best possible performance, both with respect to latency and bandwidth. A write-only protocol [4] was chosen for two reasons. First, performance of remote writes are better than remote reads, as described in [11]. Furthermore, using a write-only protocol ensures cache coherency, even though the attachment point is the I/O bus, as discussed in Section 1.3. Since reading data from local memory is much faster than fetching it over the SCI network, ScaMPI always use a remote-write-local-read policy. This contributes significantly to fulfil one of the design goals of ScaMPI: to be able to scale performance to very large systems.

1.5.1 Fault Tolerance

Three important items are required to securely manipulate data structures on a remote node:

- Atomicity of multi-byte entities must be controlled. This implies that either all or nothing of a multi-byte entity is modified, i.e., that it is never partially modified. Consider for example a simple ring-buffer structure with a write and a read index. The receiver polls the senders write-index, and compares it to the read-index. If the two indexes differ, the ring-buffer contains valid data. If the write-index is represented by a two-byte entity, and if those where updated one at a time, catastrophic errors could be the consequence when the index wrap from the least to the most significant byte!
- Enforcing memory consistency, i.e., ensuring that all previously issued write-requests have been globally performed. For example, if one process writes a block of data to a remote buffer, and then signals the completion of the transfer by writing to a flag in the receiver node memory. If the memory consistency is not enforced between the data transfer phase and the signaling phase the receiver might consume stale data.
- Error checkpointing. As discussed in Section 1.3.2, a transfer might have been corrupted, e.g., the cable has been unplugged. A methodology of checkpointing is needed to ensure correct data transfers.

The natural sequence of operations to securely transfer a data block and set a flag at the receiver is depicted in Figure 1.2.

Scali's SCI driver, ScaSCI, has combined the functionality of enforcing memory consistency and checkpointing, to improve speed. As illustrated in Figure 1.2, memory consistency has to be enforced before the checkpointing routine is called. Otherwise, active outstanding write-operations might be in progress when the `EndCheckPoint()` routine is called, and those might later be exposed to errors. Another important observation from Figure 1.2 is that

```
void SendMsg(long *remoteFlag, void *remoteDst,
             void *localSrc,   int   sizeOfMsg)
{
   CheckPointToken token;
   StartCheckPoint (&token);
   do {
        Memcpy(remoteDst, localSrc, sizeOfMsg);
        MemBarrier();
   } while (EndCheckPoint(&token) != SUCCESS);

   StartCheckPoint (&token);
   do {
        *remoteFlag = SUCCESS;
        MemBarrier();
   } while (EndCheckPoint(&token) != SUCCESS);
}

void RecvMsg(long *localFlag, void *localSCIMem,
             void *localUser, int   sizeOfMsg)
{
   while (*localFlag != SUCCESS) {
      sleep();
   }
   Memcpy(localUser, localSCIMem, sizeOfMsg);
}
```

Fig. 1.2. Pseudo-code for secure one-way data transfer

the side-effect exposed to a remote memory region might be exposed more than once, in case of error indications. The SCI responses might have been subject to errors, and not the requests. If this is the case, the side effect has taken place, but the requestor node can not distinguish between a failure of a request or of the response. Therefore, the requestor node has to re-issue the data transfer. As a consequence, the data structures used in ScaMPI had to be designed idempotent. A data structure being idempotent will be consistent even if an update was carried out more than once.

Another important aspect from Figure 1.2 is that messages actually are securely transferred to the remote node, before the sender is able to signal to the receiver that the messages is ready for consumption. The time spent to enforce memory consistency and to perform the checkpointing will be directly added to the latency of transferring a message. The impact will be more severe the smaller the message is. To avoid this added latency, ScaMPI has a combined message data structure for small payloads, the *MPI message envelope* [8, Section 3.2.3] and a field, *ready*, indicating to the receiver that this structure represents a new, unconsumed MPI message. ScaMPI uses 64 bytes to represent this information, including 32 bytes of MPI data payload. As discussed above, atomicity, or merely lack thereof, must be handled. Since few processor instruction sets have provisions for 64-byte atomic store operations, a mechanism to prevent the consumer from receiving a partly received message had to be found. Even if the sender specifically wrote the *ready* field as the last part of the transfer, the data could appear at the receiving node

in a different order, due to the possibility of reordering of packets in the SCI interconnect fabric. To avoid this pitfall, ScaMPI has included a CRC check value in the structure to protect its integrity. This approach enables ScaMPI to send self-synchronizing messages in a safe way. As a bonus of this the receiver may read the message while the sender complete enforcing memory consistency and perform the checkpointing, thus reducing the latency.

1.5.2 User Friendliness

Scali has put an effort into making ScaMPI and its environment user friendly. The execution of an MPI-program is started and controlled by a monitor program (`mpimon`). The monitor takes two types of parameters on the command line:

Parameters controlling ScaMPI set-up. These parameters include customizing set-up of SCI memory allocation, buffer sizes, barrier fanin/fanout etc. The parameters are checked for validity and, if not correct, the program execution is aborted and appropriate error messages are given. Being tuned for performance, in ScaMPI by default buffers are allocated the first time a communication channel is used. For performance measurements and communication debugging, ScaMPI can be set to initialize all the communication channels at startup time.

MPI program names, their parameters and node specifiers. The parameters are automatically distributed to all processes constituting the parallel program, not only process zero. The node specifiers are checked for legal node names. Each node can occur several times within a node specifier, enabling more than one process per node. For full control over process-to-node mapping, ranks are allocated sequentially from the node specifiers.

ScaMPI has the ability to have multiple MPI programs in one run (MIMD-paradigm). This is specified by adding multiple blocks of MPI programs, parameters and node specifiers on the command line to the monitor.

Input from the user (`stdin`) can be distributed to all or some of the processes. Output from the processes (`stdout and stderr`) is displayed in the window where the monitor was started. All processes inherit the running environment from the monitor shell, e.g., the current directory path,i.e., where the monitor was stared. The user can choose to have none, some or all environment variables copied to the processes.

Scali has added some functionality to ScaMPI and the monitor to ease debugging and to get a better overview of what is happening when running an MPI program. Output from selected nodes can be printed in separate windows or files. MPI programs can selectively be started within a separate window or debugger to allow use of other debug/trace tools.

1.5.3 Third Party Software

From a user perspective, a Scali system interface is a parallel-tools environment. The parallel user environment consists of three basic components, system access control, parallel debugger and parallel performance analysis. The purpose of a parallel-tools environment is to create a single system image of a parallel computer. It is however not possible to shield the user completely from the added parallel complexity needed to get more computational power. But a high quality parallel-programming environment contributes significantly to reduce the time spent in developing and debugging code.

Any standard MPI parallel tools [16, 9] can be used with ScaMPI. Since a Scali system is built from COTS (commercial off-the-shelf) technology by using standard hardware and software components, third party parallel tools by any independent cluster vendor can be used on a Scali system running ScaMPI. To provide the ScaMPI user with basic state-of-the-art parallel tools, ScaMPI is available with the TotalView [17] parallel debugger and the Vampir [13, 15] parallel performance analysis tools.

The TotalView [17] graphical parallel debugger has support for the most important parallel programming models: threads, MPI, PVM and HPF. Supported platforms come from the major supercomputing vendors including Compaq, Digital, SGI, IBM and Sun. The main parallel feature is the single point of control for debugging ScaMPI programs. From a single window it is possible to control individual groups of processes, hide unnecessary and display essential information. On startup, TotalView give the user an option to stop in `MPI_Init()`, the starting point of any MPI program. After this initial stop, the user can set appropriate action points before continuing the parallel debug session. TotalView has a fast and intuitive GUI, with the possibility of data visualization, a useful aid in debugging numerical programs. TotalView is designed for multiprocessing and offer the debugger features a programmer expects.

Vampir [13] is a tool for performance analysis of MPI programs. In the NHSE (National HPC Community Software Exchange) Parallel Tools review [10] Vampir was rated as the best parallel performance tool. It is available on all major supercomputer platforms. To collect performance data, the ScaMPI program is linked with the VampirTrace library, and run. The performance data is logged to a file for post-processing. Vampir ScaMPI performance analysis helps the user to organize the performance data, understand application and communication behavior, evaluate load balancing and identify communication hotspots. A very useful feature is the extensive space-time filtering of data to extract relevant information only. A timeline window display application and message passing activities and shows parallelism as the sum of active non-communicating processes. Communication statistics can be displayed for selected intervals of time and message length.

1.6 Performance Results

The tests were run on a 96 node system with dual Pentium 450 MHz processors PCs interconnected with PCI-SCI cards from Dolphin ICS [3]. The SCI network was organized as an 8 x 12 2D mesh/torus. ScaMPI delivered a 9.4 µs ping latency and up to 76 MByte/s between two MPI processes on separate nodes over the SCI network (4.5 µs and 130 MByte/s between two processes on the same PC). This fulfills two of the ScaMPI design goals (Section 1.4); low latency and high bandwidth.

As stated in the design goals for ScaMPI, latency of collective MPI operations should grow with $O(log(N))$, where N is the size of the system, while bandwidth per node should be near constant for all system sizes. Restrictions of the bandwidth in multi dimensional toruses is analytically calculated in [1] and indicate the feasibility of this goal, and in the next two subsections the achievement of these goals will be shown through practical measurements. To get comparative results the test programs were run on a far more expensive state-of-the-art 128 processor (MIPS R10k) Cray Origin 2000 equipped with 192 MByte main memory.

1.6.1 Barrier

Barrier is a collective operation that carries no data, but synchronizes all processes. Since barrier does not carry any data, it is a good measure for the collective latency. ScaMPI's barrier implementation uses a fixed fanin/fanout tree and operates directly on SCI shared memory.

Nodes	2	4	8	16	32	48	64	80	96
Timing	8.1	8.2	9.3	20.6	24.4	26.1	29.9	30.8	33.1

Table 1.1. Barrier performance in µs

Table 1.1 shows absolute timing of barrier over the SCI network. As can be seen, this is a very fast implementation with sub-latency performance up to 8 nodes! A barrier involving two processes on the same PC use only 1.4 µs. The Origin 2000 used 25.9 µs to synchronize two processes (739 µs for 64 processes) [7]. Figure 1.3 shows graphically how the performance results of the barrier compare to a $const * log(N)$ trend. As can be seen, ScaMPI over SCI shows good match between the timing of barrier, i.e., collective latency, and a $const * log(N)$ trend.

1.6.2 All-to-All Communication

The most demanding communication situation for a machine is when all nodes communicate with all the other nodes. Performance of `MPI_Alltoall()` is therefore a good measure of the aggregate bandwidth of a system.

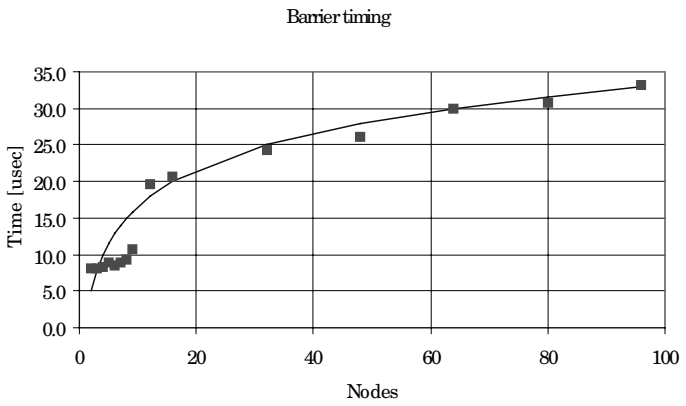

Fig. 1.3. Barrier performance compared to $const * log(N)$

Nodes	2	4	8	16	32	64	96
Throughput	30.9	35.2	36.5	31.1	32.3	31.7	26.2

Table 1.2. All-to-all communication performance per node in MByte/s.

Table 1.2 shows measured communication performance per node of all-to-all communication for long messages over the SCI network. For two processes on the same PC throughput was measured to 54.6 MByte/s. The SCI based system shows far better scaling than on the Origin 2000, which delivered a per node performance of 42.1 MByte/s between 2 processes, 26.9 MByte/s between 16 processes and ends up with a mere 7.3 MByte/s for 32 processes [7]. The very poor performance scaling on the Origin 2000 may have to do with interference from other user programs due to lack of resource reservation.

The all-to-all performance is calculated on the basis of the network traffic. Since `MPI_Alltoall()` use two buffers, an N-th part of the send buffer is copied internally to the receive buffer and is therefore not part of the network data volume. For all-to-all communication between two nodes, half of the data are transferred and the other half is copied. This is the reason for the apparently low performance between two nodes. The shift in performance between 8 and 16 nodes is caused by a change of the algorithm. If the algorithm for larger configurations had been used for all configurations no performance shift would have appeared, but the performance for small configurations would have suffered. For up to 8 nodes, the SCI network delivers sufficient throughput for all nodes to communicate at full speed. This is compliant with the constant per node performance of up to $64 (= 8 * 8)$ nodes and a small decrease for 96 nodes. By going from a 2D to a 3D torus network, the interconnect performance should scale "perfectly" to $8 * 8 * 8 = 512$ nodes. This is conformant with the conclusions in [1].

1.7 Conclusions

Our initial ambitions were make a thread-safe, scalable, low latency, high bandwidth, fault-tolerant, user friendly and flexible (with respect of transport medium) implementation of the MPI standard. ScaMPI are meeting all of these design goals.

The 96 node (192 processor) Scali system at PC2 in Paderborn is currently the world's largest system using SCI as the interconnect technology. Since Scali are using standard workstations as nodes in our systems, technology advances should ensure a continued and increasingly good price-performance ratio. By using COTS components the performance growth does not only apply to new machines buyers, but the existing nodes of an upgraded machine can be passed on in the organization as personal desktop workstations. This adds an important option for cost reduction.

By using a standard programming interface, MPI-conforming third party applications will run on Scali systems without additional work, although optimizing for the architecture may give additional speedup. ScaMPI supports a variety of options and tools to ease the programming effort to get to correct and efficient applications. As shown by the performance measurements in section 1.6.2, Scali systems scales well, thus enabling us to deliver very powerful systems to a low price. In this picture ScaMPI and its support modules plays an important role.

1.8 Acknowledgements

Thanks to Thierry Matthey at Parallab, for making performance numbers on the Origin 2000 and to the service team at PC2, Paderborn, for excellent support in bringing up their 192 processor system.

1. Håkon Bugge. Affordable Scalability using Multicubes. In *Proceedings of SCI Europe '98 at European Multimedia, Microprocessor Systems and Electronic Commerce*, September 1998.
2. David E. Culler, Andrea Dusseau, Seth Copen GoldStein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing'93, Portland, Oregon*, November 1993.
3. Dolphin Interconnect Solutions. *PCI-SCI Bridge Functional Specification*, version 3.01 edition, November 1996.
4. Manolis G.H. Katevenis Evangelos P. Markatos and Penny Vatsolak. The Remote Enqueue Operation on Networks of Workstations. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing Las Vegas, USA*, Lecture Notes in Computer Science. Springer-Verlag, February 1998.
5. G.A. Geist and V.S. Sunderam. The Evolution of the PVM Concurrent Computing System. In *Proceedings of COMPCON spring'93*, pages 549–557, February 1993.
6. James D. Lyle. *Sbus: Information, Applications, and Experience.* Springer-Verlag, 1992. ISBN 0-387-97862-3.
7. Thierry Matthey. Personal communication, 1999.
8. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 1995. Version 1.1.
9. National HPCC Software Exchange - Parallel Tools Library. At http://www.nhse.org/ptlib.
10. Review of Performance Analysis Tools for MPI Parallel Programs. At http://www.cs.utk.edu/ browne/perftools-review/.
11. Knut Omang. Synchronization Support in I/O Adapter Based SCI Clusters. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, San Antonio, Texas*, volume 1199 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, February 1997.
12. Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95, San Diego*, 1995. Avaliable at http://www-csag.cs.uiuc.edu/papers/index.html#communication.
13. Pallas GmbH. *VAMPIRtrace for Solaris x86*, 1998. Release 1.0 for VAMPIRtrace version 1.5. At http://www.pallas.de.
14. PCI Local Bus Specification, Revision 2.1.
15. Scali AS. *ScaMPI Installation and User's Guide version 1.6*, 1999.
16. IEEE CS Task Force of Cluster Computing. At http://www.dgs.monash.edu.au/rajkumar/tfcc/, 1998.

17. *TotalView Multiprocessor Debugger User's Guide*, 1998. Version 3.0. At http://www.etnus.com/tw.