SCI-MPICH: The Second Generation

Joachim Worringen

Abstract- **The first version of SCI-MPICH offered full MPI-1 functionality with very good communication performance. However, it needed improvement in the areas of reliability and stability, compatibility and usability. This paper documents the development SCI-MPICH has undergone in these areas. Next to these improvements on existing functionality, new features have been introduced. A new, asynchronous message transfer protocol using DMA with little CPU utilization is presented. The introduction of a server-less, DSM based MPI-IO implementation which directly uses the high-performance SCI interconnect for communication is an important step towards a complete MPI environment on SCI connected clusters.**

Keywords- **MPI, message passing, cluster, SCI, parallel IO, MPI-IO**

I Introduction

The first release of SCI-MPICH [1] provided the complete MPI-1 functionality. Further improvement of the performance of point-to-point communication was hardly possible. However, our own experiences and the feedback from other sites using SCI-MPICH showed that there was still a lot of work to do in terms of improvement of reliability and stability (startup and shutdown of applications, handling of error conditions), compatibility (different operating systems, different types of PCI-SCI adapters [2], different SISCI implementations), usability (compiling the libraries, configuring the memory setup) and also regarding the extension of SCI-MPICH's capabilities concerning asynchronous message transfer and MPI-IO [3] via SCI.

We have invested a considerable amount of work into the required improvements of both, the MPICH ADI-2 device named *ch_smi* and the underlying SCI shared-memory library *SMI* [4]. This paper intends to document the progress of SCI-MPICH based on the improvements and extensions given above. In this context, it is unavoidable to point to some limitations of the current generation of PCI-SCI adapters and their related driver software which emerged in the course of our development.

A. Platform

If not stated otherwise, all performance numbers in this paper were measured on a cluster consisting of 8 Dual-SMP Pentium-II nodes (450MHz, BX chipset, 32 bit PCI bus) and one Dual-SMP Pentium-III Xeon node (550MHz, NX chipset, 64bit PCI bus). Only the Pentium-II nodes were used for SCI-related measurements. The SCI interconnect consists of one Dolphin D321 PCI-SCI adapter in each node and one Dolphin D512 SCI switch, to which the nodes are connected as two-node ringlets[1]. Additionally, the nodes are connected to a 100Mbit switched ethernet network (3COM). The nodes were running Solaris 7 as operating system, and the driver software from Dolphin released Jan. 7th, 2000. Next to Solaris x86, SCI-MPICH can also be used under Solaris Sparc, Linux and Windows NT.

1. One port has three nodes connected.

Joachim Worringen is with the Lehrstuhl für Betriebssysteme, RWTH Aachen, Kopernikusstr. 16, D-52056 Aachen, Germany.
E-mail: contact@lfbs.rwth-aachen.de, WWW: http://www.lfbs.rwth-aachen.de .

B. Organization of the Paper

The next chapter gives some insights on the internal improvements that have been introduced into SCI-MPICH and illustrates their effects for the user. SCI-MPICH now offers real asynchronous message transfer protocols utilizing DMA and remote interrupts which we present in chapter III. We are currently developing a high-performance MPI-IO implementation; details are presented in chapter IV. The paper closes with chapter V which summarizes the results, points out current bottlenecks of the Dolphin SCI implementation and programming interface [2] and outlines potential improvements of SCI-MPICH.

II Internal Design Improvements

A number of internal design improvements have been made to SCI-MPICH and the underlying SMI library in order to make SCI-MPICH more stable and resource-aware.

A. Improved Startup

The original concept for the startup of an SMI application (SCI-MPICH applications are implicitly SMI applications) was to perform the initial synchronization of the processes of an application purely via SCI shared memory. This concept lead to problems since the identifier for the initial SCI memory segment is unknown prior to actually creating the segment. This led to conflicts if more than one process using SCI-MPICH was run on a single node. Therefore, the basic initialization is now done via TCP/IP by broadcasting the SCI segment ID from the initialization master to all other processes. We feel that this is the better concept even if we had to give up the „SCI-only“ concept as there is no cluster which does not offer TCP/IP.

Another new technique which is used for a faster startup is *delayed segment connection*. As each process of an SCI-MPICH application exports three SCI segments (for the three different message transfmission protocols [1]) to which each other process needs to connect to, the startup takes longer and resources are potentially wasted: depending on the communication pattern of the application, a significant number of these connections will not be used at all. Therefore, the SCI segments for the three protocols are not connected until actually required using a specific segment creation mode of the SMI library. This means that the internal data structures for the segments are set up, but a process does not connect to and import a remote shared memory segment until a message needs to be transferred via this memory segment.

B. Proper Shutdown

Another frequent problem is the proper termination of a multi-process application in case of an abnormal termination of one or more processes. We introduced an SCI based watchdog mechanism in combination with signal handlers to detect abnormal termination of the local process as well as a defunct remote process. This means that e.g. sending a SIGINT to any process of an SCI-MPICH application leads to a proper shutdown of all participating processes. Such a shutdown includes the release of all SCI related resources by the SMI library since these are not always automatically reclaimed by the operating system.

C. Memory Configuration

The memory buffers for the different message transfer protocols are allocated statically during `MPI_Init()`. To be able to improve the performance for an application which has a certain communication pattern (e.g. many messages in the size of the eager protocol), it is now possible to supply a device configuration file to SCI-MPICH which describes, among other runtime parameters, the size and number of all categories of memory buffers. This allows to easily determine the influence of these parameters on the performance of an application.

If not enough SCI shared memory resources are available to configure the buffer setup as demanded by the device configuration file, SCI-MPICH automatically reduces the number and size of the buffers until the demand matches the available resources. The reduced number or size of the buffers may have an impact on the performance, but at least the application can execute at all. The user can verify the actual buffer configuration using a special startup option of the `mpirun` command.

To better utilize SMP nodes in a SCI cluster, SCI-MPICH now uses local shared memory for intra-node communication. This leads to less consumption of SCI resources and delivers good performance. In figure 1, we present the results of point-to-point measurements between two processes on a SMP as round-trip/2 numbers.

The Pentium-III Xeon shows excellent performance if the message fits well into the cache, but shows extreme performance drops if the message size is bigger than half the cache size. This is due to the performance difference between the full-speed clocked caches and the main memory. The benchmark shows amazing results if executed on a Pentium-II 450 under different operating systems: the performance under Linux 2.2.5 (SuSE 6.2 distribution) is much worse than under Solaris 7. The local shared memory implementation of this Linux release seems to need improvement. We also compared SCI-MPICH and ScaMPI [5] on a Siemens hpcLine (Linux 2.2.10), consisting of Pentium-II 400 dual-SMP nodes. It shows that ScaMPI is superior for messages bigger 128kB, while SCI-MPICH has advantages in the range of message sizes between 4kB and 64kB.

D. Support of any PCI-SCI adapter

The first release of SCI-MPICH (and the underlying SMI library) was designed and optimized especially for the Dolphin D310 PCI-SCI adapter. With the availability of new PCI-SCI adapters such as the D321 which differs in performance and memory-consistency related aspects like size and number of stream buffers, adaptions were required due to the direct use of the stream buffers. The SMI library now determines all relevant parameters via `SCIQuery()` calls, and SCI-MPICH adapts itself to these parameters. This lead to the introduction of variably sized control packets which are described in the following paragraph.

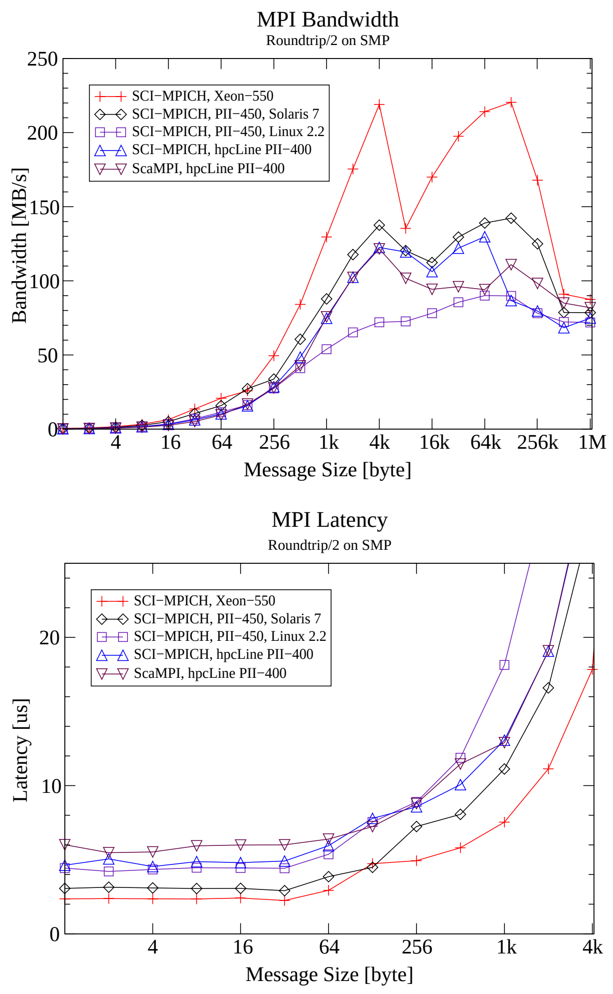


Fig. 1. MPI Bandwidth and Latency in SMP mode

The SMI library and SCI-MPICH have also been adapted to support the SISCI API that the SCI drivers by Scali Inc. offer. This driver is used for ScaMPI, for example on the Siemens hpcLine systems. It is now possible to run SCI-MIPCH applications on these systems. First performance comparisons of point-to-point operations have shown similar performance of ScaMPI and SCI-MPICH. This allows to use MPICH-compatible tools like Vampir [6] or TotalView [7] on the hpcLine[1].

E. Arbitrary Size of Control Packets

Control packets, which are also used to transport small amounts of data (*short* message transfer protocol) are the basic inter-process communication facility for SCI-MPICH. For the first release of SCI-MPICH, the size of these packets was fixed

1. These two tools are not available for ScaMPI on Linux.

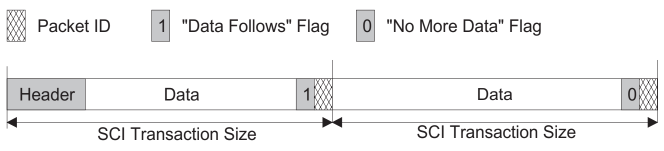


Fig. 2. Format of arbitrary-sized control packets

to the size of a stream buffer. As this size was 64 byte for the D310 adapter boards, it equaled the size of an SCI transaction by which the contained data is to be transferred atomically[1]. This atomicity allowed us to write the packet in a highly efficient, self-synchronizing manner. As the stream buffers of the D321 adapter are sized 128 bytes and with no SCI transaction of matching size available, a new technique for the self-synchronization was required.

We developed an efficient technique which allows the control packets to be sized at any multiple of the size of the biggest SCI *write* transaction available (which is currently 64 bytes, but might be increased to 256 bytes in the future) while maintaining the low minimal latency of 64 byte-sized control packets. The data format of these packets is illustrated in Figure 2: the control packet (which also serves to transmit inlined short messages) is divided into parts which have the length of the largest SCI *write* transaction. Each part is synchronized by the closing *Packet ID* and additionally carries a flag which indicates if this is the last part of the packet or if more data follows. The effect of this technique is shown in figure 3: different sizes of the control packets cause different switch points from the short to the eager protocol (which are the clearly recognizable steps). Bigger control packets cause a smoother transition between the two protocols while only slightly raising the minimal latency. The lowest latency, however, is still achieved using 64 byte control packets.

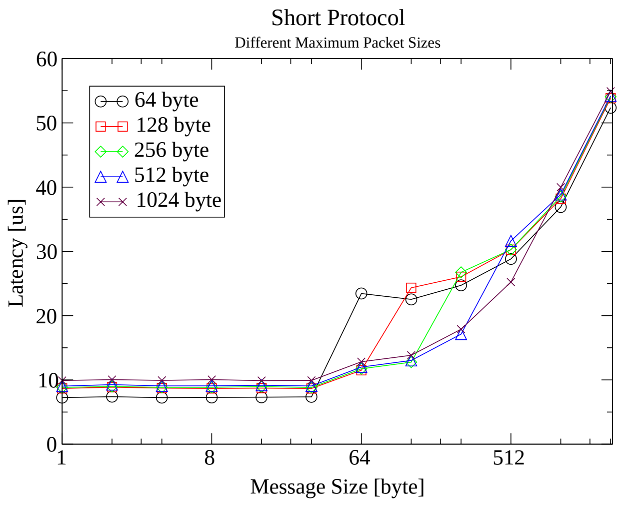


Fig. 3. Latency for differently sized short packets

1. If data is transferred via 2 or more SCI transaction packets, it is not guaranteed that it appears at the receiver in the same order as written by the sender.

III Asynchronous message passing

Latency Hiding is a well known concept which is used in many areas of computing to reduce the performance impact of latencies caused by data access of the CPU. Its actual implementation technique does of course depend on the nature of the data access and the implied latency. For the case of *message passing* via MPI, the data access is the sending and receiving of messages, and the implied latency is the time the CPU is busy with writing the data to a remote node or waiting for the data to appear in the local receive buffer and transfer it to the user buffer. The technique we present to hide this latency is the fully asynchronous implementation of the non-blocking MPI calls `MPI_Isend()` and `MPI_Irecv()` by using the DMA and remote interrupt capabilities of the PCI-SCI adapters. This allows the transmission of data with very little usage of the CPU, freeing it to perform other tasks during the message transmission.

A. DMA via SCI

The current generation of Dolphin PCI-SCI adapters has an integrated DMA engine which allows the movement of data between two nodes. We have measured the key performance values bandwidth and latency for remote writes via DMA for the Dolphin D321 PCI-SCI adapter (see figure 4).

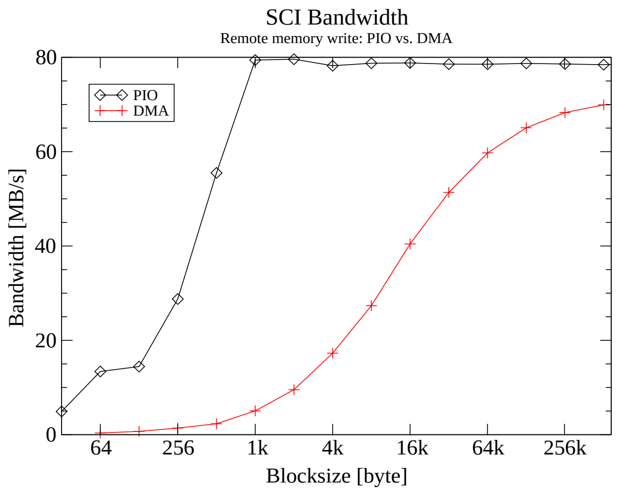


Fig. 4. Bandwidth for remote memory write access

The latency of DMA transfers can be split into the setup time (preparing the transfer and enqueuing the request) and the transfer time itself. The related diagram (figure 5) shows the complete latency and the constant setup time. The other diagram shows the bandwidth up to its saturation point. Both bandwidths are limited by the maximum length of bursts via the PCI bus (128 bytes) which the PCI-SCI adapter is able to perform.

These values promise a comparable performance as to using the CPU to move the data, at least for large messages. However, it has to be considered that DMA transfers are only possible between two SCI memory segments; it is not possible to issue DMA transfers between arbitrarily chosen memory areas. This makes additional data transfers or registering of user memory areas necessary[2]. Also, the two segments involved in a DMA

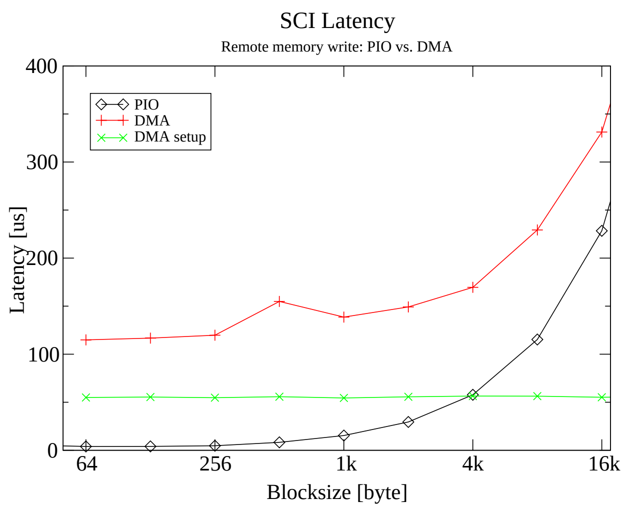


Fig. 5. Latency for remote memory write access

transfer need to be addressed via the same file descriptor. However, usually different file descriptors are used because each file descriptor can only be used for one local and one remote segment. We have introduced a re-connection technique in the SMI library to overcome this limitation: the descriptor of the local segment is used to connect to the remote segment. These connections are cached to avoid the occuring overhead (which is illustrated as a cumulative histogram in figure 6) for the upcoming transfers.

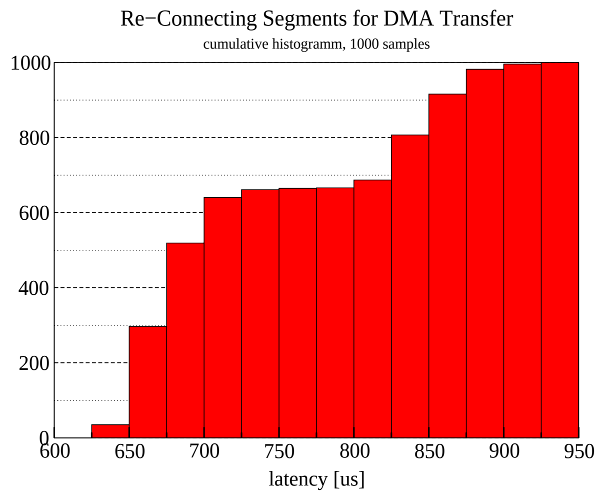


Fig. 6. Overhead of (re)connecting remote segments

The support for memory transfer via DMA is integrated into the SMI library: a transfer is issued by `SMI_Imemcpy()` while `SMI_Memtest()` and `SMI_Memwait()` are used to test or wait for the completion of the transfer. Chaining of multiple DMA transfers is also supported. Unfortunately, the current SISCI library does not yet support the use of callback functions for DMA transfers; we use threads to achieve an equivalent behavior.

2. The SISCI call SCIRegisterSegmentMemory() which is required to perform DMA to or from an arbitrary user allocated memory area is not yet implemented.

B. Interrupts via SCI

To perform the message transfers asynchronously (independently from any MPI calls that the application may or may not perform), an extension of the usual notification technique via *control packets* is required. Incoming control packets are not detected and processed until the process to which the control packets was sent calls a related MPI function. Therefore, we are using remote interrupts via SCI to trigger a special thread of the receiving process which in turn processes the queue of incoming control packets independently from the MPI application thread of the process.

The use of remote interrupts is also supported by the SMI library via the `SMI_Signal_send()` and `SMI_Signal_wait()` functions. It is also possible to install a callback function for an interrupt. The average latency of a remote interrupt without callback (measured as round-trip/2 of the two SMI functions) is currently about $35,6\mu s$ as as can be seen from the cumulative histogramm in figure 7.

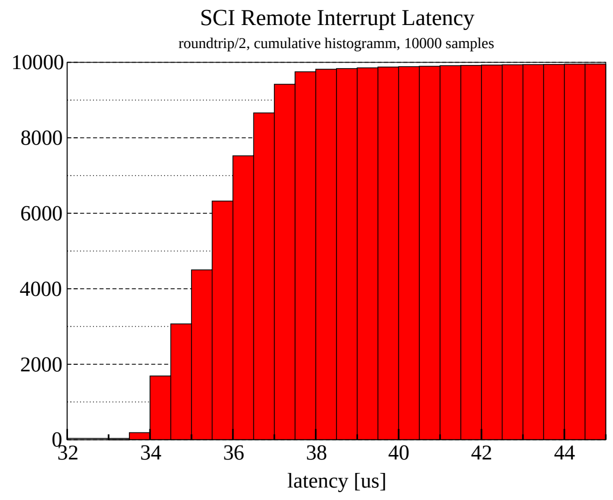


Fig. 7. Remote interrupts via SMI

C. Asynchronous Eager Protocol

A transmission of a message via the *eager* protocol includes two steps: transfering the data from the local user buffer into the remote receive buffer and sending a control packet to inform the receiving process of the arrival of the new message. To perform this transfer asynchronously, a second thread is used. Due to the limitation that DMA is only possible between registered SCI memory segments, it is necessary to first copy the data into a local SCI memory buffer, then issue the DMA transfer and wait for its completion[1]. The control packet to announce the new message is sent and the remote process is signaled.

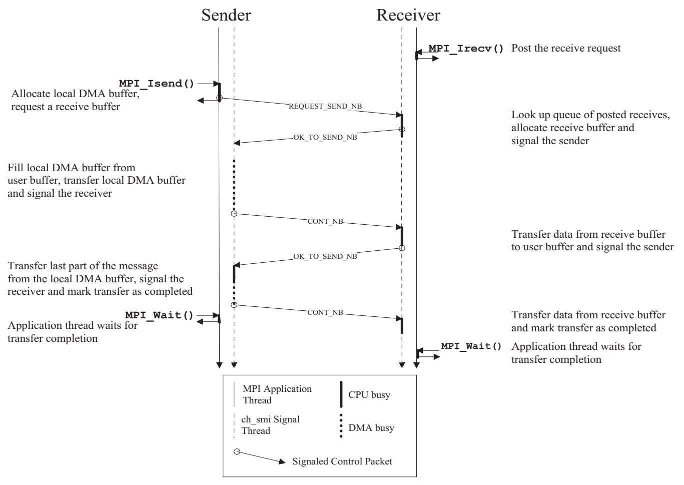
However, the size of messages handled by the eager protocol is usually only up to 32kB which is too small for efficient DMA transfers: the overhead for the DMA is too high compared to the duration of the copy operation performed by the CPU. There-

1. Waiting for the completion of a DMA transfer, and also waiting for a signal to arrive does not consume any CPU cycles.

fore, DMA transfers are disabled by default, but may me activated if desired.

D. Asynchronous Rendez-Vous Protocol

The *rendez-vous* protocol is more complex as it is designed to transfer messages of arbitrary length. Therefore, the protocol must be able to transfer a message in multiple parts in case the intermediate buffers are not big enough to save the whole message. Each of these transmissions has to be synchronized between the sender and the receiver. Figure 8 illustrates the protocol mechanism. It exhibits the case of an *expected receive*, that means the receiver is already waiting for the message that the sender sends via `MPI_Isend()` as he has called `MPI_Irecv()` before.



Sender
Receiver
MPI_Irecv() Post the receive request
MPI_Isend()
Allocate local DMA buffer, request a receive buffer
REQUEST_SEND_NB
Look up queue of posted receives, allocate receive buffer and signal the sender
OK_TO_SEND_NB
Fill local DMA buffer from user buffer, transfer local DMA buffer and signal the receiver
CONT_NB
Transfer data from receive buffer to user buffer and signal the sender
OK_TO_SEND_NB
Transfer last part of the message from the local DMA buffer, signal the receiver and mark transfer as completed
CONT_NB
MPI_Wait()
Application thread waits for transfer completion
Transfer data from receive buffer and mark transfer as completed
MPI_Wait() Application thread waits for transfer completion
MPI Application Thread
CPU busy
ch_smi Signal Thread
DMA busy
Signaled Control Packet

Fig. 8. Asynchronous rendez-vous protocol specification

E. Performance

In its current state, the absolute performance in terms of bandwidth and latency of the asynchronous protocols using DMA is worse than the conventional, CPU driven protocols.

The reason for this comparable low performance becomes obvious if the bandwidth of the chained copy operations is considered. Table 1 gives the related bandwidths for a blocksize of 1 MB. Therewith, the upper limit of the bandwidth B_{rndv} can be calculated to

$$B_{rndv} < \frac{1}{\frac{1}{B_{SD}} + \frac{1}{B_{DD}} + \frac{1}{B_{DR}}}$$

which results in 35.3 MB/s for the given message size of 1MB. This value is covered by our measurements which are depicted in figure 9. It shows the effective bandwidth and the related upper limit of the current 3-copy protocol. The difference between these values is an indicator for the overhead of the protocol. It shows that for messages > 512kB, the efficiency of the protocol is more than 97%, reaching 99% for message sizes of 8MB. The lower efficiency for smaller messages is due to the constant overhead of the rendez-vous protocol: two messages need to be exchanged between the sender and the receiver before the actual memory transfer can be started (see figure 8).

Copy Operation	**Name**	**Bandwidth**
user send buffer to local DMA send buffer	B_{SD}	146 MB/s
local DMA send buffer to remote DMA recv buffer	B_{DD}	69 MB/s
local DMA recv buffer to local user recv buffer	B_{DR}	146 MB/s

Tab. 1. Bandwidth of copy operations for a block size of 1MB

For reference, figure 9 also shows the raw DMA bandwidth and an estimation of the performance of the 2-copy protocol (DMA directly from the user buffer) to be implemented once the drivers provide the required functionality of registering user memory areas.

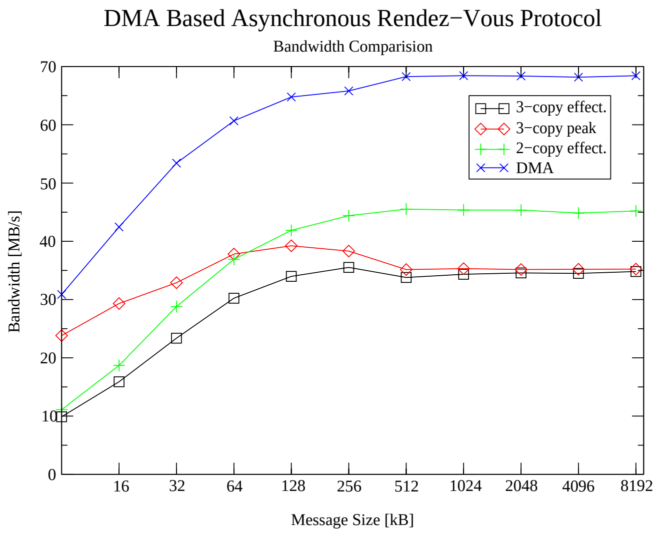


Fig. 9. Asynchronous rendez-vous performance

The lower bandwidth of the DMA protocol is a disadvantage, but the data transfer via DMA does cost only a fraction of the CPU cycles and allows the overlap of computation and communication. To demonstrate this advantage, we have designed a synthetical benchmark *overlap* (see figure 10) which simulates the overlapping of computation and communication (for combinations of job and message sizes). The sender posts an asynchronous send operation (`MPI_Isend()`) and then simulates a computation for a specified amount of time (`jobsize`). When

```
latency = MPI_Wtime()
if (sender)
   MPI_Isend(msg, msgsize)
   while (elapsed_time < jobsize)
      spin
   MPI_Wait()
else
   MPI_Recv()
latency = MPI_Wtime() - latency
```

Fig. 10. Pseudo Code for *overlap* benchmark

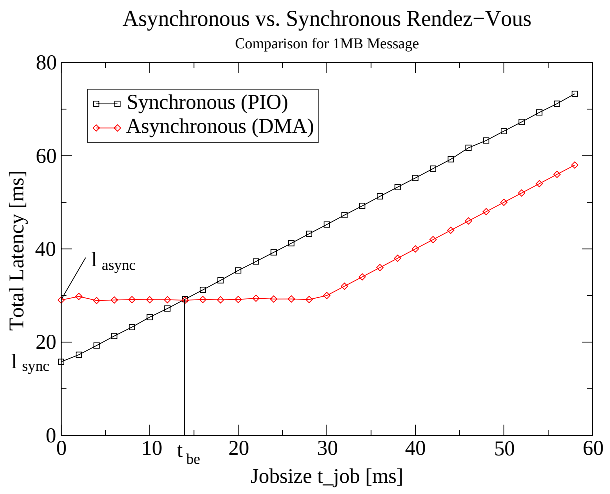


Fig. 11. Effect of overlapping Computation and Communication

the computation is finished, he blocks until the message to be sent has been transferred (actually, `MPI_Wait()` returns when the message buffer is available to be overwritten). The other process polls to receive a message.

The results of this benchmark for job sizes t_{job} between 0 and 60ms and a message size s_{msg} = 1MB is shown in figure 11. The curve for the synchronous protocol starts with an overall latency $l_{total} = l_{sync}$ = 18ms, which is the message transmission latency, and rises linearly with t_{job}. The curve of the asynchronous protocol starts with $l_{total} = l_{async}$ = 29ms, but remains on this level as long as $t_{job} < l_{async}$ is valid. This indicates a good overlapping of computation and communication. The two curves intersect at a t_{job} = 15ms. This intersection represents the performance break-even point t_{be}: for $t_{job} > t_{be}$, it is better to use the asynchronous protocol to minimize the overall latency.

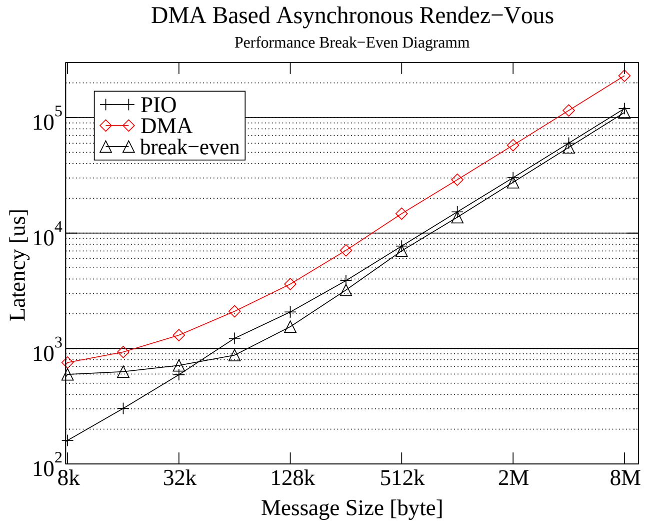


Fig. 12. Break-even point for asychronous rendez-vous via DMA

Obviously, the break-even point can generally be determined as $t_{be} = l_{async} - l_{sync}$. This relation is illustrated in figure 12. It shows the transmission latencies against the message size for the two rendez-vous protocol variants and the break-even points calculated as the difference between the latencies.

However, to make real use of this feature, the programmer of the MPI application has to design its communication pattern accordingly. The well known NAS parallel benchmarks [8], for example, make nearly no explicit use of asynchronous communication. Because the user of an SCI-MPICH application can specify if DMA or PIO transfers are to be used for asynchronous message passing, he is able to determine the appropriate mode by comparing two runs of his application with and without DMA transfers.

IV MPI-IO VIA SCI

Many MPI applications do not only require communication between the processes, but also have to perform a significant amount of file I/O for various purposes [9]. Often the execution time of such an application is not dominated by the communication, but by the I/O (next to the time required for calculation). The usual I/O interfaces offered by operating systems do not support parallelism; and if a system features mechanisms for parallel I/O, they have to be used via a proprietary interface. Therefore, the MPI-2 standard includes a definition of a programming interface for parallel I/O called MPI-IO [3] to allow portable programming of MPI applications with I/O requirements.

A. State of MPI-IO

Currently, at least two portable implementations of MPI-IO exist, next to a number of solutions dedicated to a single type of machine:

- ROMIO [10] was developed at the ANL (free, many file systems supported)
- PMPIO [11] by NASA Ames (free, some file systems supported, no NFS)
- MPI-IO for the PIOFS and GPFS file systems by IBM (freely available, only proprietary file systems)
- Pallas has developed a complete MPI-2 implementation for Fujitsu.

Probably due to its close relationship to MPICH (ROMIO is included in the current MPICH distribution), ROMIO is the most commonly used MPI-IO implementation. It accesses the actual I/O devices via an Interface called *Abstract Device I/O* (ADIO [12]). This approach is similar to MPICH's ADI-2 [13] interface. To have ROMIO support a specific file system or I/O library, it is necessary to glue them with a layer (called ADIO device) which translates the function calls defined by the ADIO into the function calls required by the underlying system. Usually, implementing such an ADIO device is not too complicated due to the inherent similarities between I/O interfaces. This leads to a growing support for ROMIO.

B. Related Work

A lot of work has been done in the area of parallel I/O, let it be parallel file systems or user-level libraries for parallel I/O. As MPI-IO is becoming an important standard interface for parallel I/O in scientific computing, several of the more general solutions are enabled for MPI-IO by supplying a suitable interface.

Todays high-performance clusters are usually networked with an ethernet-type network for TCP/IP based services and a *high-performance interconnect* (HPI) like SCI or Myrinet for inter-process communication. To use MPI-IO on clusters, two solutions can currently be used:

- the ROMIO distribution contains an ADIO device *ad_nfs* to use a file located on a NFS server which is reachable by all processes of the MPI application
- the *Parallel Virtual File System* (PVFS [14]) was recently adapted to ROMIO via a suitable ADIO device [15].

However, all of these solutions use TCP/IP based services for the communication between the clients (the processes which need to use the I/O services) and the servers (the processes which offer and perform the I/O services). So far, there is no approach to directly utilize a HPI using a lean protocol, avoiding the immense overhead of a TCP/IP stack. Having the TCP/IP stack operate on the HPI is no real solution, either, since related approaches have shown that only a fraction of the HPI's performance will be delivered by the TCP/IP stack on top [16].

A possible approach would be to implement an MPI-IO solution in the scope of *SciOS* [17] which offers a file system interface for SCI memory named *SciFS* [18]. However, SciFS only supports non-persistent files and is implemented as a Linux kernel module which hinders portability to other operating systems.

C. Concept & Implementation

We wanted to design an MPI-IO implementation which makes optimal use of the fast SCI interconnect by having minimal protocol overhead. We also wanted to avoid the introduction any potential bottlenecks by using dedicated servers. This lead to the idea of using distributed, memory-mapped I/O between the MPI processes as the foundation of our design.

In a straight-forward approach for an SCI connected cluster, every process would have a part of the whole file available locally (mapped into its address space from a file on its nodes local hard disk), and the remaining parts which are located on other nodes are accessed via memory areas connected by SCI. Accesses to data from the file are simple accesses to memory locations. Therewith, nearly no protocol overhead at all occurs when accessing portions of the file, and all processes are servers as well as clients. Figure 13 illustrates this basic principle for the case of two processes (running on distinct nodes).

However, this straight-forward design has several drawbacks. Read accesses to remote portions of the file would be very slow, and caching is not possible in a simple manner. Additional mechanisms to ensure the required degree of consistency would have to be added. Finally, a lot of address space (matching the size of the file) mapped into the SCI address space would be required which is a problem for all current operating systems and the available drivers for the PCI-SCI adapters.

These problems made the introduction of an additional software layer inevitable which had to

- convert remote read accesses (to access portions of the file located on other nodes) into remote write accesses from the remote node towards the local node.

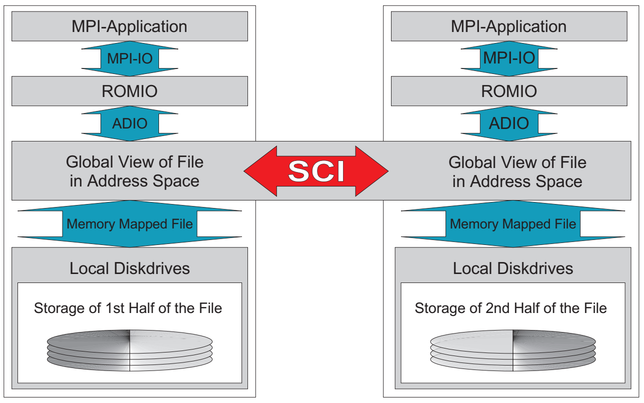

MPI-Application
MPI-IO
ROMIO
ADIO
Global View of File in Address Space
SCI
Memory Mapped File
Local Diskdrives
Storage of 1st Half of the File
MPI-Application
MPI-IO
ROMIO
ADIO
Global View of File in Address Space
Memory Mapped File
Local Diskdrives
Storage of 2nd Half of the File

Fig. 13. Concept of MPI-IO via memory-mapped files and SCI

- install a software-controlled cache for remote portions of the file.
- offer a suitable consistency model for accesses to the memory that makes up the file.
- use SCI for communication, but without having to map the whole file into the SCI address space.

These characteristics indicate that the software layer would be rather complex - it's just a complete DSM system. The Lehrstuhl für Betriebssysteme has developed such a DSM system as a user-level library called *SVMlib* (Shared Virtual Memory library) [19] which uses SCI (or TCP/IP, if SCI is not available) for inter-process communication. We decided to make the SVMlib the basic building block of our MPI-IO concept, which lead to a design of the complete MPI-1 (conventional message passing) and MPI-IO environment as illustrated in figure 14.

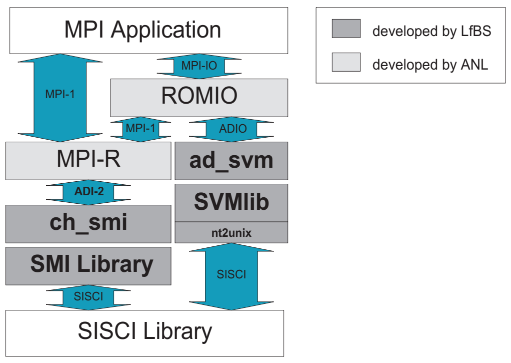


Fig. 14. Software Architecture of MPI-1 and MPI-IO via SCI

The MPI-1 part is not connected with the MPI-IO part and thus did not change. For the MPI-IO part, we use the unmodified ROMIO 1.0.1 implementation and use our ADIO compliant device *ad_svm*. This is based on the slightly modified SVMlib[1], which in turn uses the SISCI API to access the SCI resources. Currently, ROMIO uses some MPI-1 functionality for internal communication.

1. SVMlib had to be modified to use persistent files instead of temporary files, and a protocol for locking portions of the memory had to be introduced to allow *atomic access*.

This concept leads to a three-dimensional data distribution model as illustrated in figure 15. The first dimension are the nodes on which the processes are running and which each store a fraction of the file. The second dimension is the size of each of these fragments (called *segments*). The third dimension is the number of segments on each node. More than one segment is used if the file is growing: the size of a segment that is once mapped into the memory is fixed, thus file enlargement has to be done by creating new segments and mapping them behind the existing ones. The size of these new segments is a critical parameter: the smaller this size is chosen, the more frequently it may be necessary to create a new segment. On the other side, smaller segments improve locality since a smaller segment is more likely to store only data of the local processes.

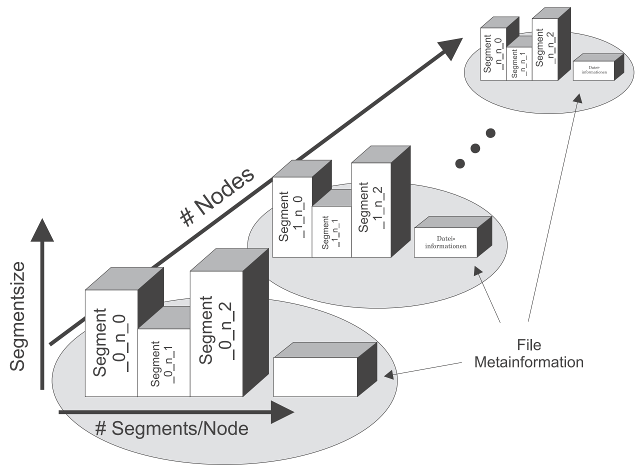


Fig. 15. Data distribution for MPI-IO via SCI

D. Performance

The first prototype which we used to produce the performance numbers below is a fully working MPI-IO implementation, but without any optimizations. We will present the results of two benchmarks: *coll_perf* is a synthetical benchmark included in the ROMIO distribution which collectively writes and reads a block-distributed, three-dimensional array in a non-contiguous manner as illustrated in figure 16. The dimensions of the Array are 128 integers each, resulting in a file size of 8MB. For optimum locality, the segments of the global file would have to be arranged as illustrated in figure 16. However, the current default setting for the segment size is 1MB.

The other benchmark *BTIO* [20] (part of the NAS NPB) is more application-oriented and iteratively solves three block-tri-agonal systems of linear equations. After each iteration step, the results are written into a file. We compare the results of our ad_svm device with the standard ROMIO solution for clusters, ad_nfs (PVFS was not yet available for Solaris).

Before comparing these MPI-IO results, it is necessary to evaluate the performance of the underlying file systems which is UFS for ad_svm and NFS for ad_nfs. The nodes of the cluster are equipped with SCSI hard disks of type IBM DDRS-34560D and SCSI host adapters Adaptec 2940UW, while the NFS server we used is a Quad-SMP Sun Enterprise 450 with hard disks of type IBM DNES-318350. All systems are interconnected via switched fast ethernet. We ran the *Bonnie* [21] disk performance benchmark with 1GB files to determine the raw read/write performance of the used storage devices. The results are given in table 2.

I/O device	block read	CPU	block write	CPU
UFS_{PII}	7,3 MB/s	9,9 %	8,4 MB/s	13,4 %
$NFS_{PII->450}$	2,4 MB/s	4,7 %	6,7 MB/s	12 %
UFS_{450}	20,8 MB/s	23,0 %	14,7 MB/s	21,7%

Tab. 2. Raw I/O performance of UFS and NFS storage devices

UFS_{PII} is the local disk I/O of a cluster node; $NFS_{PII->450}$ is the NFS I/O of such a node towards the NFS server. UFS_{450} is the local disk I/O of the NFS server to exclude its disk performance as a potential bottleneck. The results show, that I/O is not very CPU intensive and is thus well suited to be executed asynchronously.

Table 3 shows the results of the two benchmarks for 4 processes on 4 nodes. The message passing as well as the IO was performed via SCI for the ad_svm version and via FastEthernet for the ad_nfs version. The bandwidth given for the coll_perf benchmark is the collective bandwidth of all nodes, while the time period given for the BTIO benchmark is the execution time for the application including I/O and computation.

Benchmark	ad_svm	ad_nfs
coll_perf: read	25,7 MB/s	3,7 MB/s
coll_perf: write	28,0 MB/s	0,25 MB/s
BTIO, class S	1,62 s	10,13 s

Tab. 3. Benchmark results for MPI-IO via SCI vs. NFS

It was expected that MPI-IO via SCI would be much faster than via NFS over fast ethernet. These first results proof that the presented new concept is indeed vastly superior to the conventional solution, even in the state of a non-optimized prototype.

V Summary & Future Work

The presented improvements to the first release make SCI-

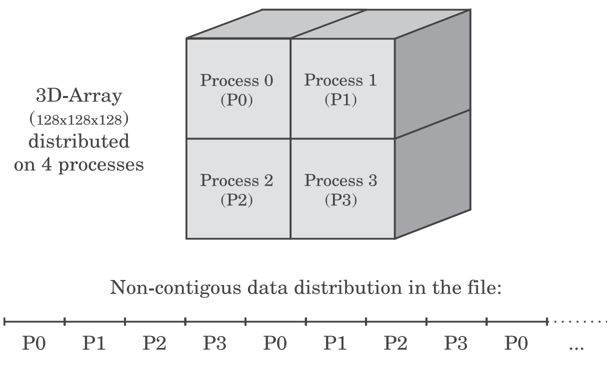


Fig. 16. coll_perf's data organization in memory and on file

MPICH now a easily usable and reliable MPI implementation which delivers high performance for very low cost. The introduction of new features like the asynchronous, DMA based message passing protocol indicates the direction of further development which needs to be exploited by the application programmers. The presented design of MPI-IO via SCI and the performance of the first prototype promises to utilize the performance of SCI not only for message-passing, but also for the important area of file I/O.

A. Fighting the Limitations

When creating the software that makes up SCI-MPICH, we had to discover a number of limitations in the current implementation of SCI for clusters (which is realized by PCI-SCI adapters by Dolphin ICS). Parts of these limitations are due to the hardware, parts are due to the driver software of the operating system. We want to mention these limitations to stimulate the discussion on how to remove them.

A.1 DMA transfers

Our experience showed that DMA transfer is only possible for segments which are created with identical SCI descriptors which, under a UNIX environment, is usually not the case. We solved this problem by re-connecting the concerned segments. Additionally, DMA read accesses and callback functionality was not yet implemented in the driver software. The same is true for the registering of existing memory areas as SCI shared memory which would allow DMA transfers directly from and to user buffers. This feature would greatly enhance the performance of the asynchronous protocols, if 2-copy of even single-copy can be implemented.

A.2 SCI Shared Memory Size

The operating systems impose different limits on the size of the exported SCI shared memory since this memory needs to be non-pageable:

- Solaris x86 needs *low memory pages* for exporting SCI shared memory. For Solaris 7, this currently limits this amount to less than 4MB which is not sufficient for applications with a large number of processes.
- Linux statically allocates a part of the address space for this purpose. This is not the ideal solution, too, as this address space is not available for other purposes, but at least imposes no fixed upper limit.
- NT uses memory from a common pool of non-pageable memory. The size of this pool is configurable.

A.3 Concurrent Performance

The accumulated performance of remote write accesses toward distinct memory locations which are performed concurrently by multiple threads is worse than the related performance of a single thread. This should be avoidable by a suitable scheduling of the available stream buffers which has to be configured by the driver. If DMA and PIO transfers are executed concurrently, the resulting bandwidth is in the same range as the bandwidth for non-concurrent transfers.

A.4 Porting to Linux

Our primary development platform is Solaris on x86-based nodes. However, many of the actual or potential users of SCI-MPICH are running some sort of Linux on their cluster. This made it necessary to port SCI-MPICH including the SMI library to Linux, too. While the basic porting process (adapting and compiling the sources from Solaris to Linux) was done easily, it showed that another category of problems caused a lot of trouble: Linux is not Linux. While SCI-MPICH ran fine on our local Linux-driven cluster on up to 8 nodes, other sites which were running other Linux distributions (or other releases of the same distribution) had severe problems to use more than 2 processes in an SCI-MPICH application due to a different behavior of the `mmap()` function in the C library. This key component has changed frequently and obviously in incompatible ways making it a pain to develop low-level software for Linux.

Finally, it showed that Linux 2.2.5 has a lower memory performance than Solaris 7 (see figure 1). This also shows up for remote memory accesses; the reason has to be determined.

B. Directions for the Future

Although many open issues of the first SCI-MPICH version have been fixed and additional functionality has been introduced, there is still room for improvement. The most important areas are described below.

B.1 Improved DMA performance

The current implementation of the asychronous message transfer protocols are 3-copy protocols which is of course not the optimal solution. We already have the concepts for 2-copy or even single-copy variants of these protocols which can not yet be realized because the required SISCI calls to register user memory areas are not yet implemented. These variants will put the performance of the asynchronous protocols on a level similar to the synchronous protocols with very little CPU load.

B.2 MPI-IO

The current implementation of MPI-IO via SCI is merely a prototype which offers a lot of optimization potential in all layers:

- ROMIO contains some functionality to enhance performance when using traditional file systems, but which is simply overhead in our environment.
- The use of MPI-1 functions has to be changed in a way to safely allow the use of threads for asynchronous MPI-IO calls. Currently, this is not possible since the MPI-R layer of MPICH which processes MPI-1 calls is not yet thread-safe.
- The ad_svm device needs to determine the optimal size for new segments which are to be created to reduce segment creation overhead while maintaining a high degree of locality of the data distribution. It can do so by internal book-keeping and by getting more information from ROMIO or the application.

The system also needs significant enhancements to increase its usability. Among these enhancements are a standard UNIX interface to access the files from any application, daemons to retrieve files from any node in the cluster and the introduction of a backup-server for higher availability and better manageability. The development of this environment is already in progress.

B.3 Fault Tolerance

Although the current SCI-MPICH version allows the application to continue while another node within the cluster is rebooting, the failure handling needs further testing and improvement. A long-term goal is the support of multiple PCI-SCI adapters for redundancy and also improved performance for systems with multiple PCI buses.

B.4 Scheduler

The Lehrstuhl für Betriebssysteme has developed an RPC-based cluster management system with a Java interface for client applications. A useful addition to this system will be a scheduler for SCI-MPICH jobs which also considers the MPI-IO requirements of the applications to run.

References

[1] J.Worringen and Th.Bemmerl: *MPICH for SCI-Connected Clusters.* In Proc. SCI-Europe 1999, pp. 3-11, Toulouse, 1999 http://wwwbode.in.tum.de/events/sci-europe99/proceedings http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH

[2] M.C.Liaaen and H.Kohmann: *Dolphin SCI Adapter Cards.* In *SCI: Scalable Coherent Interface,* Edited by H.Hellwagner and A.Reinefeld, LNCS 1734, Springer, 1999

[3] MPI-2 standard, including MPI-IO specification http://www.mpi-forum.org/docs/docs.html

[4] M.Dormans, K.Scholtyssik, Th.Bemmerl: *A Shared Memory Programming Interface for SCI Clusters,* In *SCI: Scalable Coherent Interface*, Edited by H.Hellwagner and A.Reinefeld, LNCS 1734, Springer, 1999

[5] Scali AS: *Scali MPI - ScaMPI.* http://www.scali.com

[6] Pallas GmbH: *Vampir: Visualization and Analysis of MPI Programs*, http://www.pallas.de

[7] Etnus In.c.: *TotalView Multiprocess Debugger*, http://www.etnus.com

[8] D.H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow: *The NAS Parallel Benchmarks 2.0*, NASA Technical Report NAS-95-020, NASA Ames Research Center, December 1995 http://www.nas.nasa.gov/Software/NPB

[9] R. Oldfield and D.Kotz: Applications of Parallel I/O, Department of Computer Science, Dartmouth College, Hanover, Technical Report PCS-TR98-337, August 1998. http://www.cs.dartmouth.edu/pario

[10] R. Thakur, W. Gropp, and E. Lusk: *On Implementing MPI-IO Portably and with High Performance*, in Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, May 1999, pp. 23--32. http://www-unix.mcs.anl.gov/romio

[11] NASA Ames Research Center: *PMPIO - A portable MPI-2 I/O library* http://parallel.nas.nasa.gov/MPI-IO/pmpio/pmpio.html

[12] R. Thakur, W. Gropp, and E. Lusk: *An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces*, in Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp. 180-187. http://www-unix.mcs.anl.gov/~thakur/adio

[13] E. Lusk and W. Gropp: *The implementation of the second generation MPICH ADI.* MPICH working note (draft), Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997 http://www.mcs.anl.gov/mpi/mpich/workingnote/adi2impl/note.html

[14] W. B. Ligon III, R. B. Ross, and R. Thakur: *PVFS: A Parallel File System For Linux Clusters*, submitted to ICS 2000, December, 1999. http://www.parl.clemson.edu/pvfs

[15] H. Taki and G.Utard: *MPI-IO on a Parallel File System for Clusters of Workstations“*, in Proc. of IEEE Int. Workshop on Cluster Computing, Dec. 1999, http://www.dgs.monash.edu.au/~rajkumar/tfcc/IWCC99

[16] H. Taskin, R.Butenuth: *TCP/IP over SCI under Linux,* In *SCI: Scalable Coherent Interface*, Edited by H.Hellwagner and A.Reinefeld, LNCS 1734, Springer, 1999

[17] P. T. Koch, J. S. Hansen, E. Cecchet and X. Rousset de Pina: *SciOS : An SCI-based Software Distributed Shared memory.* In Proc. 1st Workshop on Software Distributed Shared Memory, June 1999 http://sci-serv.inrialpes.fr

[18] Povl T. Koch, J. S. Hansen, E. Cecchet and X. Rousset de Pina: *Implementing a File System Interface to SCI.* In *SCI: Scalable Coherent Interface*, Edited by H.Hellwagner and A.Reinefeld, LNCS 1734, Springer, 1999

[19] K. Scholtyssik, M. Dormanns: *Simplifying the use of SCI shared memory by using software SVM techniques.* In Proc. 2nd Workshop Cluster Computing, Karlsruhe, March 1999. http://www.lfbs.rwth-aachen.de/users/karsten/projects/SVMlib

[20] R. Carter, B. Ciotti, S.Fineberg and B. Nitzberg: *NHT-1 I/O Benchmarks.* NASA Ames Research Center, Technical Report RND-92-016, Nov. 1992 http://www.nas.nasa.gov/Pubs/TechReports/RNDreports/RND-92-016/RND-92-016.html

[21] T. Bray: *Bonnie, Benchmark for Unix Filesystem Operations*, http://www.textuality.com/bonnie