Design and implementation of FMPL, a fast message-passing library for remote memory operations

Osamu Tatebe[1] Umpei Nagashima[1] Satoshi Sekiguchi[1] Hisayoshi Kitabayashi[2]
Yoshiyuki Hayashida[3]

[1] National Institute of Advanced Industrial Science and Technology
1-1-1 Umezono, Tsukuba, Ibaraki 3058568 Japan
o.tatebe@aist.go.jp, u.nagashima@aist.go.jp, s.sekiguchi@aist.go.jp
[2] Hitachi Business Solution, Software Development Department
[3] Hitachi, Ltd., Software Division

ABSTRACT

A fast message-passing library FMPL has been designed and developed to maximize communication performance by utilizing general architectural communication support such as remote memory operations, as well as to maximize total performance by eliminating dynamic communication overhead and overlapping communication and computation. FMPL provides a low-cost general-purpose point-to-point communication and collective communication such as broadcast, barrier synchronization and reduction. On a Hitachi SR8000, FMPL achieves an 8-byte latency of 12.8μsec., while MPI achieves 20μsec. FMPL is designed for building more highly functional message-passing libraries like BLACS as well as applications that need maximum performance.

Keywords

Message-passing library, remote memory operations, hardware broadcast, hardware barrier synchronization, Hitachi SR8000, BLACS, FMPL, BLACS-FMPL

1. INTRODUCTION

Most distributed memory machines including PC clusters have supported remote memory operations to achieve high-performance communication, which enables zero-copy point-to-point communication from a user memory space to another user memory space directly. Zero-copy point-to-point communication reduces the overhead of memory copy and achieves close to the peak network performance[12, 18, 9]. That research attempted to develop a highly efficient MPI using remote memory operations, which is the standard and portable message-passing library, though it is too highly functional and not tuned for remote memory operations. A low-cost message-passing library designed for remote memory operations and low runtime overhead is needed especially for benchmark programs and applications that need high performance.

The parallel linear algebra package ScaLAPACK[2] including matrix solvers, least squared problem and eigenproblem, is built on top of BLACS[4] (Basic Linear Algebra Communication Subprograms) that is designed as a basic communication library for linear algebra. MPI implementation of BLACS is widespread, whereas it includes double message-handling overhead of both BLACS and MPI since there are the packing and unpacking overhead of two-dimensional arrays of BLACS messages and overhead of general-purpose and high-functional MPI. To achieve high performance, it is desirable that BLACS is efficiently built on a light-weight, general-purpose and fast message-passing library.

This paper designs a fast message-passing library FMPL for remote memory operations with low runtime overhead and implements it on Hitachi SR8000. FMPL exploits not only remote memory operations but also underlying hardware support for collective communication such as barrier synchronization and broadcast.

2. FMPL: A FAST MESSAGE-PASSING LIBRARY

The FMPL aims to maximize communication performance by utilizing general architectural communication support as well as to maximize total performance by eliminating dynamic communication overhead and overlapping communication and computation, and provides low-cost point-to-point and collective communication facilities. The basic design principles of the FMPL are listed below:

- High-performance exploitation of general architectural communication support such as remote memory operations, hardware barrier and hardware broadcast,
- Low-latency and high-bandwidth point-to-point communication with sender-side or receiver-side message

matching and zero-copy or one-copy communication using remote memory operations,

- Elimination of dynamic overhead such as dynamic memory allocation, queuing operations, search for unused memory location and interrupt, and
- Flexibility high and overhead low enough to efficiently implement BLACS and MPI.

Since remote memory operations do not synchronize between an origin and a target, explicit synchronization or algorithmic guarantee such as double buffering is necessary in order not to overwrite a target memory area. Programming with remote memory operations has flexibility to reduce communication and message-handling overhead, while too much optimization makes debugging and validation of correctness difficult.

Point-to-point communication consists of message transfer and local synchronization. A sender specifies a send buffer, and a receiver specifies a receive buffer. Send and receive calls are matched using ranks of source and destination, a message tag and so on. Local synchronization of point-to-point communication ensures that the send buffer is not transferred until a send operation issues and safely modified after the completion of the send, and the receive buffer is not overwritten before a receive operation issues and has a valid data after the completion of the receive.

Point-to-point communication is a basic operation for programming on distributed memory machines, and it helps fast collective communication with tree structure such as broadcast and reduction. FMPL point-to-point communication is designed as a primitive operation using remote memory operations and low-overhead local synchronization to efficiently implement a comprehensive message-passing library like MPI and BLACS. Benchmark program and performance-critical application also need FMPL as well as remote memory operations to achieve much high performance.

FMPL also provides collective communication with FMPL point-to-point communication and underlying hardware communication mechanism if available. Usually, there is a trade-off and regulation to utilize hardware support, performance of both implementations of collective communication should be carefully evaluated for adaptive approach.

2.1 Design of point-to-point communication

Point-to-point communication needs matching of corresponding send and receive in FIFO order. Message headers including starting address of buffer, message tag and communicator are usually managed using a queue that needs dynamic operations such as enqueuing, dequeuing, searching an entry and allocating memory that incurs runtime overhead.

Design of the FMPL point-to-point communication is influenced by remote memory operations and sender-side matching mechanism[18, 9] to achieve low latency, low overhead and high throughput, though it is not a requirement. FMPL point-to-point communication specifies an exclusive matching area as a message tag, which is used to notify a sender a message header including address of the receive buffer as well as the corresponding receive has been issued. Because the matching area is exclusively specified by a pair of send and receive, FMPL can remove dynamic overhead such as

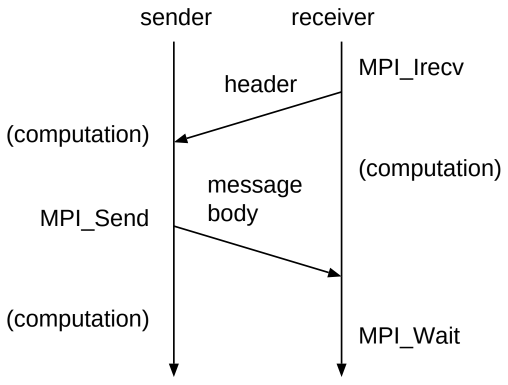


Figure 1: Zero-copy point-to-point communication implemented by immediate remote memory writes and sender-side matching mechanism

dynamic memory allocation, queuing and searching during message matching.

Sender-side message matching reduces point-to-point communication latency and overhead, which is depicted by Figure 1. A receiver sends a message header to a sender using remote memory write, and the sender checks the message header and transfers a message to the receive buffer also using remote memory write. Network latency of the FMPL point-to-point communication takes only one round-trip latency, and that can be hidden by programming such that nonblocking receive calls are issued in advance. A receive call writes the message header remotely without inquiring of a sender, and a send call writes the message body remotely without inquiring of the receiver at the send-time when the corresponding message header has already arrived.

Zero-copy point-to-point communication implies synchronous or rendezvous communication semantics that mean a send call is not completed unless the corresponding receive call is issued. When a receive call is issued much later than the corresponding send call, the waiting overhead of the sender is not acceptable. FMPL also provides one-copy communication such that the sender copies the message body to a temporary buffer, and it will be copied to the receiver when the corresponding receive call is issued. For the one-copy communication, FMPL does not provide separate APIs but provides an API for specifying timeout to change from the zero-copy protocol to the one-copy protocol.

For one-copy communication, a temporary buffer is desirable to be allocated in the receiver with receiver-side message matching, which is called *eager* protocol. However, dynamic protocol changing from sender-side matching to receiver-side matching needs additional queues for message headers and additional runtime overhead[18]. On which side the temporary buffer is allocated is implementation-dependent.

Sender-side matching needs a receiver to specify a rank of source explicitly, not a wildcard, while there is a situation such that receiving a message from any process is indispensable. In this case, message matching should be done on the receiver side, and dynamic protocol change is needed again. To avoid this complexity, we design another API for point-to-point communication that specifies wildcard as a rank of source, which has another communication domain and is assumed to be implemented using receiver-side message matching. This design is based on the observation that a sender should know whether a receiver is waiting for a message by specifying a specific rank of source or a wildcard.

In the case of a wildcard message tag, it is possible to be implemented using sender-side matching unless a rank of source is also specified by a wildcard. FMPL provides a wildcard for a message tag.

2.2 Initialization and finalization

```
fmpl_init(ma, fa, count, ier)
fmpl_finalize(ier)
```

fmpl_init initializes the execution environment of the FMPL message-passing library and specifies the matching area `ma` and the completion-flag area `fa` with `count` elements, and returns a return code `ier` that is set `FMPL_SUCCESS` on success and an error code on error. fmpl_finalize terminates the execution environment.

2.3 Point-to-point communication

```
fmpl_send(buf, size, dst, mai, comm, ier)
fmpl_recv(buf, size, src, mai, comm, ier)
```

fmpl_send sends a message from the send buffer specified by the starting address `buf` and the buffer size `size` to the rank `dst` of the communicator `comm` with the index of matching area `mai`, and returns a return code `ier`. The communicator will be described in Subsection 2.4. fmpl_recv receives a message to the receive buffer specified by the address `buf` and the buffer size `size` from the rank `src` of the communicator `comm` with the index of matching area `mai`, and returns a return code `ier`.

An index of matching area is similar to a message tag, while the same index cannot be used by two pairs of send and receive at the same time since it is assumed that there is no queue management in the matching area. The wildcard message tag FMPL_ANY_TAG can be specified as an index of matching area to receive a message with any message tag.

fmpl_send and fmpl_recv are blocking communication interfaces, fmpl_send terminates when the send buffer can be safely modified and fmpl_recv terminates when all data is received to the receive buffer. Using zero-copy implementation, blocking communication has the same semantics as the synchronous mode in MPI such that a receive and a send do not terminate until the corresponding send and receive issues, respectively. FMPL also provides one-copy buffered-mode communication by specifying the send buffer area in advance to copy out send messages.

```
fmpl_sendbuf_set(buf, size, ipara, ier)
fmpl_sendbuf_check(nsend, nspool, ier)
```

fmpl_sendbuf_set sets a send buffer `buf`, the buffer size `size` and a timeout `ipara` in milliseconds. If there is no corresponding message header, fmpl_send waits at least `ipara` milliseconds. When fmpl_send call is timed out, the send message is copied into the send buffer. The send buffer is managed by the FMPL library and should not be modified by the user process.

fmpl_sendbuf_check checks the send buffer and sends messages whose corresponding receive has been already issued, and returns the number of send messages `nsend` and the number of pending messages in the send buffer `nspool`.

FMPL provides nonblocking communication interfaces:

```
fmpl_isend(buf, size, dst, mai, comm, ier)
fmpl_irecv(buf, size, src, mai, comm, ier)
fmpl_isend_wait(dst, mai, comm, ier)
fmpl_irecv_wait(src, mai, comm, ier)
```

fmpl_isend and fmpl_irecv return even if the corresponding call has not been issued. FMPL nonblocking communication works very well when the nonblocking receive call is issued in advance. In this case, the corresponding send call sends a message immediately without inquiring of the receiver at send-time and reduces waiting overhead.

The following interfaces are for point-to-point communication whose rank of source is specified by a wildcard.

```
fmpl_send_any(buf, size, dst, mai, comm, ier)
fmpl_recv_any(buf, size, mai, comm, ier)
```

Communication domains between fmpl_send_any and fmpl_send are separated. Any message sent by fmpl_send cannot be received by fmpl_recv_any, and any message sent by fmpl_send_any cannot be received by fmpl_recv. That is because fmpl_send_any and fmpl_recv_any can be implemented by a different approach, such as receiver-side matching, from fmpl_send and fmpl_recv.

2.4 Collective communication

FMPL collective communication provides barrier synchronization, broadcast and reduction within a group of nodes called *communicator*. All collective communication calls should be called by all processes in a communicator.

```
fmpl_comm_create(comm, key, ier)
```

fmpl_comm_create is called by all processes and returns a new communicator which consists of a group of nodes that specify one for `key`. `FMPL_COMM_WORLD` is defined as an initial communicator that consists of all processes.

FMPL collective communication utilizes FMPL low-cost point-to-point communication and communication hardware support adaptively if available.

2.4.1 Broadcast

```
fmpl_bcast(buf, size, root, comm, ier)
```

fmpl_bcast broadcasts a message specified by the address `buf` and the size `size` on the node `root` to all other processes in the communicator `comm`. All processes within the communicator `comm` should call fmpl_bcast with the same root process.

2.4.2 Reduction

```
fmpl_allreduce(buf, count, func, comm, work, ier)
fmpl_reduce(buf, count, func, root, comm,
            work, ier)
```

fmpl_allreduce reduces the array `buf` with the reduction function `func` on all processes within the communicator `comm` using the working buffer `work` and returns the reduced array `buf`. fmpl_reduce returns the reduced array `buf` only on the process `root`.

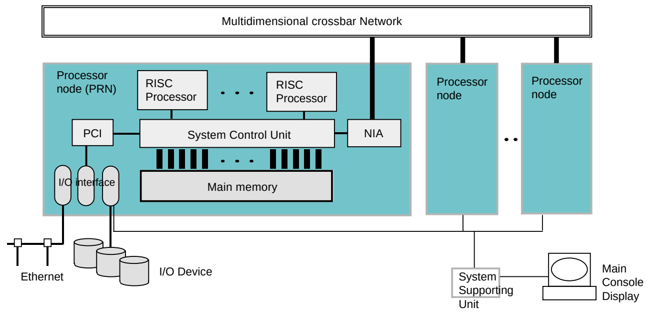


Figure 2: Processor node diagram of SR8000

2.4.3 *Barrier synchronization*

```
fmpl_barrier(comm, ier)
```

fmpl_barrier blocks until all processes in the communicator `comm` have issued this routine.

3. ARCHITECTURAL SUPPORT FOR COMMUNICATION ON SR8000

The node of SR8000 is basically an 8-way SMP, each node facilitating PowerPC-based processors enhanced with pseudo vector processing[10], extended registers, extended instructions, LTLB and so on. Each node has a special hardware called a *cooperative microprocessors mechanism* to support invoking several processes, which helps parallel loop execution.

The diagram of the processor node of SR8000 is shown in Figure 2. Major components of the processor node are PowerPC-based processors, a main memory and NIA (Network Interface Adapter) that controls the interconnect communication.

The interconnection network of SR8000 is called a *multidimensional crossbar network* that is a smooth extension of a multidimensional binary hypercube network to n-ary in each dimension[3]. The interconnect of SR8000 is up to three dimensions with each dimension up to 8 nodes, the maximum bandwidth achieving half-duplex 1000MB/s. SR8000 supports the remote DMA transfer, the hardware barrier synchronization and the hardware broadcast mechanism exploiting the multidimensional crossbar network.

3.1 Remote DMA transfer

SR8000 facilitates a fast user-level one-sided message transfer mechanism called a *remote DMA transfer* and provides a special *combuf* communication library. The remote DMA transfer directly transfers from a send area to a receive area in user program space on different nodes using the multidimensional crossbar network. Before processing the remote DMA transfer, it is necessary to map a data area in user virtual address space onto a contiguous remote DMA area in kernel physical address space on both sender and receiver and to get a transmission right for a target remote DMA area.

Message transfer is initiated by storing the starting address of a *transmission control word* (TCW) to the control register of NIA. TCW includes the data transfer information such as a starting address of send data, a data size, a destination node, a remote DMA identifier, an offset and so on.

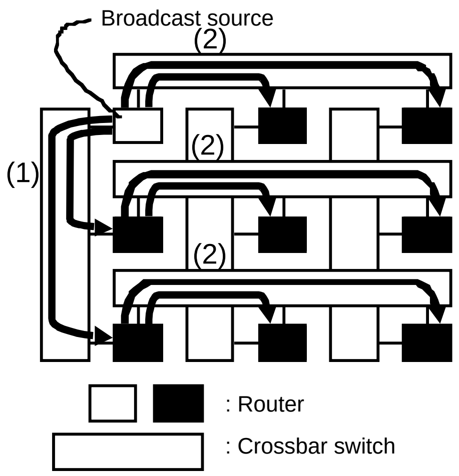


Figure 3: Hardware broadcast mechanism

3.2 Hardware broadcast mechanism

The hardware broadcast mechanism is shown in Figure 3. “Serialized crossbar” is uniquely determined in the system so that two or more hardware broadcasts may not be performed simultaneously. In the case of a two-dimensional crossbar network, a broadcast source node sends a data to the “serialized” y-crossbar, which will be transferred via all x-crossbars to all other nodes. Since each crossbar is a complete crossbar, the “Serialized crossbar” can send the data to all x-crossbars in parallel.

The hardware broadcast requires the following conditions:

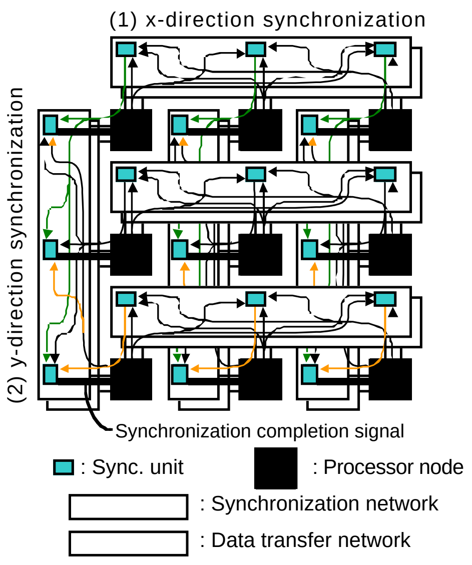
1. Processes are executed on different nodes.
2. A partition in which the processes are executed, is needed to have both exclusive attribute and global attribute.
3. A partition in which the processes are executed, should not be overlapped on a partition that has shared attribute.
4. Nodes on which processes are allocated shape a rectangle or a hypercube.

In addition to the conditions above, the offset of both sending area and receiving area in each node should be the same.

3.3 Hardware barrier mechanism

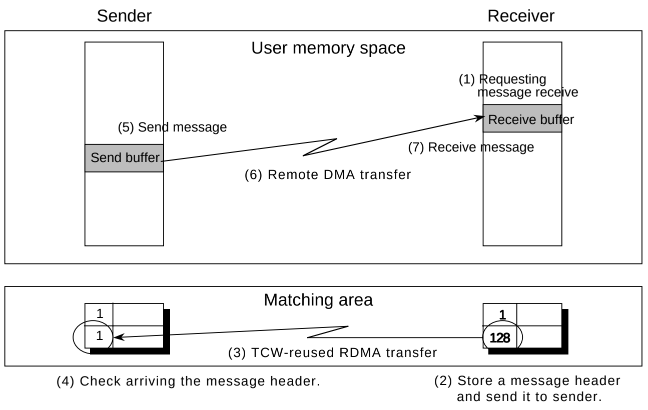
Figure 4 shows the hardware barrier mechanism in the case of a two-dimensional crossbar network. At first, the hardware barrier determines that all nodes on the same x-crossbar get to the synchronization point using a feature of completeness of the crossbar, and transmits it to all y-crossbars. The synchronization signal on x-crossbars is not gathered to a specific y-crossbar but transmitted to all y-crossbars simultaneously. Second, each y-crossbar determines the synchronization on the xy-plane concurrently, and transmits the synchronization signal to all nodes without using x-crossbars.

The hardware barrier can be utilized by hmpp_barrier in the SR8000 remote DMA transfer library. hmpp_barrier can specify one of eight colors for a synchronization factor. The



(1) x-direction synchronization
(2) y-direction synchronization
Synchronization completion signal
: Sync. unit
: Processor node
: Synchronization network
: Data transfer network

Figure 4: Hardware barrier mechanism



Sender
Receiver
User memory space
(1) Requesting message receive
Receive buffer
(5) Send message
Send buffer
(7) Receive message
(6) Remote DMA transfer
Matching area
1
1
1
128
(3) TCW-reused RDMA transfer
(4) Check arriving the message header.
(2) Store a message header and send it to sender.

Figure 5: Implementation of zero-copy blocking point-to-point communication

hardware barrier is available on the same condition as the hardware broadcast.

4. IMPLEMENTATION OF FMPL AND BLACS-FMPL ON SR8000

4.1 Point-to-point communication

Remote DMA transfer on SR8000 is invoked by creating a TCW and storing the address of the TCW in the control register of the NIA. TCW can be reused if the original and target addresses and message size are not changed. In this case, remote DMA transfer is invoked by storing the address of the TCW in the control register using combuf_kick_tcw or combuf_kick_tcw_fast. This fast transfer is called *TCW-reused remote DMA transfer*. Transfer of a message header to a matching area can exploit this fast transfer mechanism since this has a fixed communication pattern and fixed size.

Figure 5 shows the process of zero-copy point-to-point communication. It is assumed that the send buffer, the receive buffer and the matching area are allocated in remote DMA regions. When a receive call is issued, the message header is stored to the matching area on the receiver, and sent to a sender using combuf_kick_tcw_fast by the TCW-reused remote DMA transfer. On the other hand, the sender sends a message by remote DMA transfer using the received message header in the matching area that will be reset for reusing the matching area.

When a send call exceeds a wait time limit, the message body is copied to a temporary buffer on the sender side. This approach is less efficient than the eager protocol because the receiver takes longer time to get the message at receive-time. However, we prefer the benefit of sender-side message matching.

4.2 Collective communication

Hardware broadcast has strict regulations described in Subsection 3.2, whose runtime-check incurs heavy overhead. To cope with these regulations, two buffers are initially set up for hardware broadcast, which are alternately used in pipeline to overlap memory copying and hardware broadcasting. Software binary-tree broadcast using the FMPL point-to-point communication is also implemented, which will be carefully compared with the hardware broadcast implementation for adaptive broadcast approach.

For barrier synchronization, hardware synchronization using hmpp_barrier and software binary-tree synchronization using the FMPL low-cost synchronization are compared for adaptive synchronization. Hardware barrier synchronization also has regulations. However hmpp_barrier can handle the case not to apply the hardware barrier synchronization using the hardware broadcast and the remote DMA transfer.

For reduction, only the software binary-tree approach using FMPL point-to-point communication is implemented because there is no hardware support for reduction.

4.3 BLACS-FMPL

BLACS is a communication layer of the parallel linear algebra package ScaLAPACK[2] and PBLAS, which defines basic communication subroutines for linear algebra in the same way as BLAS defines basic linear algebra subroutines.

BLACS-FMPL is a BLACS implementation on top of the FMPL. BLACS-FMPL is basically implemented based on the MPI implementation of BLACS (MPIBLACS) available from the Netlib software repository[11]. The semantics of the BLACS point-to-point communication are blocking communication and assumed to be buffered mode implicitly; a send call terminates locally to ensure that the send buffer can be modified safely. FMPL point-to-pint communication satisfies this semantics by specifying timeout appropriately.

In BLACS point-to-point communication, a message buffer is specified by a partial matrix or a lower/upper triangular matrix. The following interfaces are BLACS send calls.

```
vGESD2D( ICONTXT,
         M, N, A, LDA, RDEST, CDEST )
vTRSD2D( ICONTXT, UPLO, DIAG,
         M, N, A, LDA, RDEST, CDEST )
```

Both interfaces specify the communication context ICONTXT. The send buffer is specified as a two-dimensional matrix using M, N, A and LDA. Even when the send buffer is a one-dimensional array, it is specified in the same way. RDSET and CDEST are process ranks in a two-dimensional process grid of a context. `UPLO` indicates whether the matrix is upper triangular (trapezoidal) or lower triangular. `DIAG` indicates whether the diagonal of the matrix is included or not. The first letter `v` of an interface name is one of `I`, `S`, `D`, `C` and `Z`, which stands for a datatype of integer, single precision real, double precision real, single precision complex and double precision complex, respectively.

MPIBLACS creates a user-defined datatype to send and receive a partial matrix or a triangular matrix, though the creation of a new user-defined datatype incurs heavy overhead. BLACS-FMPL reduces this creation overhead and packing/unpacking overhead as much as possible. BLACS-FMPL checks at first whether an input two-dimensional array is contiguous or not to avoid packing the already contiguous input array. Even if packing or unpacking is necessary, two statically allocated remote DMA regions are used in pipeline for a packing or unpacking area to reduce dynamic memory allocation overhead and to overlap packing and transmitting the array.

5. PERFORMANCE EVALUATION

This section evaluates basic performance of the FMPL and BLACS-FMPL using 16 nodes among 64 nodes of Hitachi SR8000 in the Tsukuba Advanced Computing Center (TACC), the National Institute of Advanced Industrial Science and Technology (AIST). Each node has a peak performance of 8000MFlops, and the total 64-node system has a peak performance of 512GFlops. The interconnect for up to 64 nodes is a two-dimensional crossbar network, whose bandwidth of each link is half-duplex 1000MB/s.

5.1 Point-to-point communication

Figure 6 shows the throughput of point-to-point communication of TCW-reused RDMA, FMPL, BLACS-FMPL, MPI and MPIBLACS, which is measured by ping-pong communication. TCW-reused RDMA is not actually point-to-point communication because there is no receive call but a polling loop for the remote memory write, but shows the peak communication performance of the SR8000. The MPI used in this evaluation is a product of Hitachi, and MPIBLACS is compiled with the Hitachi MPI. `Rdma` with parenthesis indicates the program is compiled with the `-rdma` option in order that the static area in the user program is allocated in the static remote DMA area, whose send and receive calls are executed using remote DMA transfer. `Nordma` indicates send and receive calls are executed in buffered mode such that the data is copied between the user memory space and the remote DMA area on both sender and receiver.

FMPL, BLACS-FMPL and MPI with zero-copy communication attain the performance of almost 1000MB/s that is the maximum bandwidth of the network. At the message size of 16MB, FMPL and BLACS-FMPL attain 999MB/s and MPI(rdma) attains 997MB/s. Since MPI(nordma) needs to copy data from a user memory space to a remote DMA area in a sending process and to copy data from a remote DMA area to a user memory space in a receiving process, the throughput attains 791MB/s that is about 80% of the peak bandwidth. MPIBLACS creates a new user-defined datatype for input and output two-dimensional array before sending and receiving the array. When the underlying MPI implements locally-blocking MPI_Send in every data size, the array is sent using MPI_Send with the user-defined datatype, otherwise the array is packed and sent

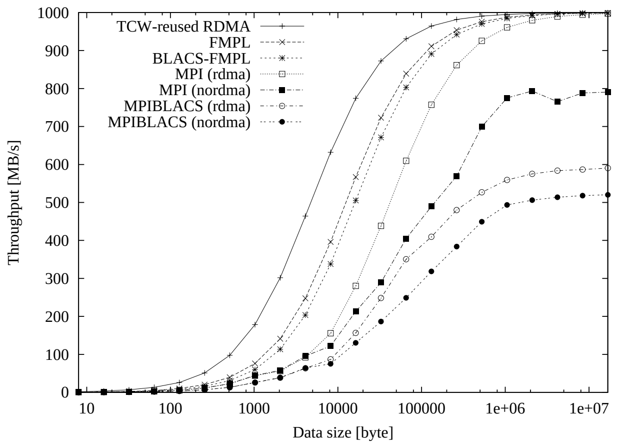


Figure 6: Throughput of point-to-point communication

Table 1: Breakdown of point-to-point communication

	Init.	Recv header	Send mesg
Send	0.5	5.0	8.1
	Init.	Send header	Recv mesg
Receive	0.5	6.1	5.6

[μsec]

using nonblocking send call MPI_Isend. MPIBLACS always needs the creation overhead of a new user-defined datatype and additional message copy on both sender and receiver. MPIBLACS(rdma) and MPIBLACS(nordma) attain only 591MB/s and 522MB/s, respectively. On the other hand, BLACS-FMPL avoids creating a new datatype and packing and unpacking the input message because the input data is contiguous, and attains close to the maximum bandwidth.

8-byte one-way latency is 12.8μsec. in FMPL, 15.7μsec. in BLACS-FMPL, 20μsec. in MPI and 34μsec. in MPIBLACS. Since an MPI send call issues combuf_get_sendright to get the right for sending to a receive field at the first invocation, the measurement excludes the first case.

FMPL needs a round-trip message transfer for point-to-point communication; a receiver sends a message header to a sender's matching area and the sender sends a message body to the user's receive buffer both using remote DMA transfer. Hitachi MPI chooses an eager protocol for a short message that also needs a round-trip message transfer where a sender sends both message header and body to a receiver's temporary buffer using remote DMA transfer and receives the acknowledgement from the receiver to notify the temporary buffer can be safely reused. However MPI requires much more overhead, such as searching unused temporary area and heavy message matching.

Table 1 shows the breakdown of the FMPL point-to-point communication at the transfer size of 8bytes, which is measured using the machine cycle counter of the processor. Since the latency of the TCW-reused remote DMA transfer is 4.7μsec., it proves that the point-to-point communication is implemented at low cost. The message body transfer takes longer than the header transfer because the TCW is not

Table 2: 4-byte one-way latency of point-to-point communication with hundreds of outstanding non-blocking receive calls

	Outstanding irecv	One-way latency
FMPL	7.03	9.90
MPI	370.1	945.8

[μsec]

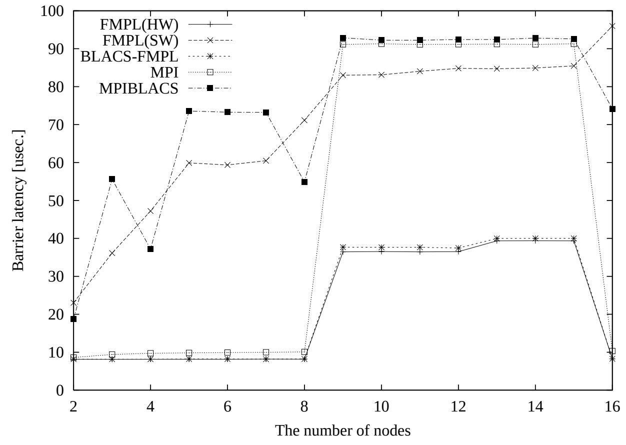


Figure 7: Barrier synchronization

reused but modified each time.

Table 2 shows the 4-byte one-way latency of point-to-point communication with several hundreds of outstanding non-blocking receive calls. In this case, one-way latency of the FMPL achives 9.9μsec., since every send call can send a message immediately because the corresponding message header has already arrived. The outstanding irecv shows the gap between consecutive FMPL nonblocking receive calls. Hitach MPI has quite large overhead when many outstanding nonblocking receive calls are issued in advance because it checks all outstanding receive calls at every nonblocking receive invocation, while FMPL does not incurs this overhead since there is no outstanding receive queue.

5.2 Barrier synchronization

The latency of barrier synchronization is shown in Figure 7. FMPL(HW) uses hmpp_barrier in the SR8000 remote DMA library for the hardware barrier mechanism, which can be utilized with 2 to 8 nodes and 16 nodes since the node partition for 16 nodes is divided in the direction of the y-crossbar like 8 $\times$ 2 and 2 to 8 processes and 16 processes are physically allocated in a rectangular shape. The latency of hardware barrier synchronization is 8μsec. FMPL(SW) utilizes the tree-based FMPL local synchronizations, which takes $O(\log p)$ with number of processes p. This software barrier utilizes the same number of buffers as the processes, one stage of synchronization can be overlapped except when the number of processes is a power of two. Since FMPL(HW) is faster than FMPL(SW) in every case, FMPL always selects FMPL(HW). FMPL-BLACS shows almost the same performance as FMPL(HW).

MPI barrier exploits the hardware barrier synchronization with 2 to 8 nodes and 16 nodes, while it is slower than the

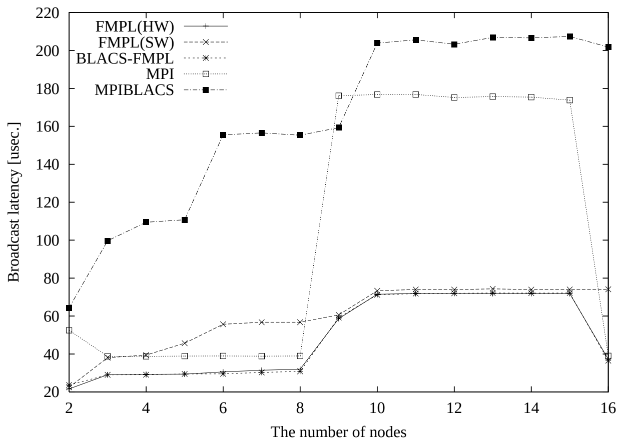


Figure 8: Broadcast latency (8KB data)

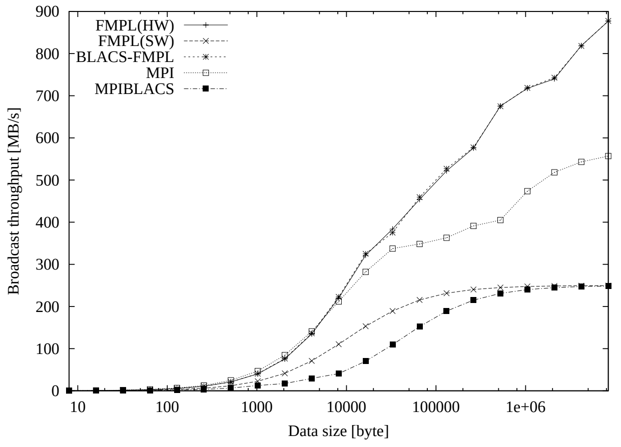


Figure 9: Broadcast throughput with 16 nodes

FMPL(SW) with 9 to 15 nodes.

5.3 Broadcast

The broadcast latency of an array with a length of 8KB is shown in Figure 8, which is measured by changing a root process every iteration. FMPL(HW) utilizes two library buffers alternately to overlap hardware broadcasting and copying. FMPL(SW) communicates with the binary tree, broadcast time takes $O(\log p)$ with number of processes p. In this case, FMPL(HW) is always faster than FMPL(SW). MPI utilizes a hardware broadcast like a barrier synchronization with 2 to 8 nodes and 16 nodes, while the broadcast latency is quite large in other cases.

Figure 9 shows the throughput of broadcast with 16 nodes, which is also measured by changing a root process every iteration. The throughput is calculated by the numnber of data size divided by the latency of broadcasting. FMPL(HW) achives 877MB/s with a length of 8MB and is still approaching the peak performance, on the other hand, FMPL(SW) only achives 250MB/s. FMPL-BLACS shows almost the same performance as FMPL(HW). FMPL improves the broadcast throughput 57% with a length of 8MB than MPI.

5.4 Reduction

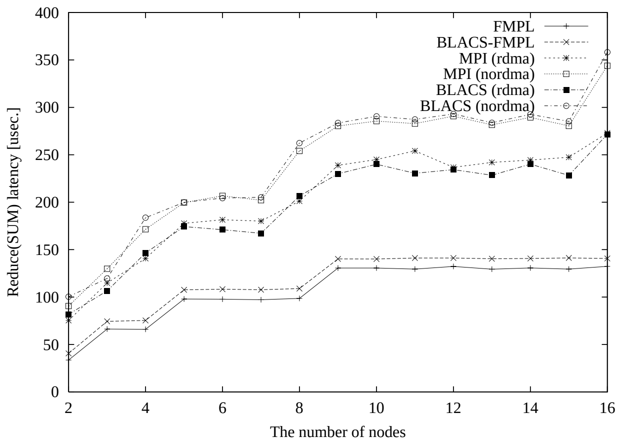


Figure 10: Array reduction of total sum (8KB data)

Table 3: Performance of FT class A and class B [sec.]

#nodes	Class A		Class B	
	MPI	FMPL	MPI	FMPL
2	17.94	17.56	193.07	189.25
4	8.63	8.53	97.42	96.75
8	4.43	4.30	47.93	47.87
16	2.43	2.19	24.32	24.09

Figure 10 shows the elapsed time of fmpl_reduce that calculates the total sum of each array element with a length of 8KB. The reduction time is measured by changing a root process every iteration similar to the broadcast.

Since reduction is calculated using the binary tree in every library, reduction time takes $O(\log p)$ with number of processes p. With a small array of 8KB, reduction time is greatly influenced by communication latency compared with the operation time of each process, that of both FMPL and BLACS-FMPL is about half of other libraries.

5.5 NAS parallel benchmarks

This subsection evaluates FMPL using the NAS parallel benchmarks 2.1, FT and MG. Table 3 shows the performance of FT class A and class B. FT needs all-to-all communication to transpose a three-dimensional array. As evaluated in Subsection 5.1, the throughput of the point-to-point communication is advantageous to the FMPL when the message size is between several KB and several hundred KB. With 16 nodes on class A, FMPL gains 11% performance improvement, in which the message size is about 500KB. Figure 11 shows the breakdown of computation and communication in execution time of class A. With 16 processes, the communication overhead is about 5% in FMPL, and 13% in MPI.

Table 4 shows the performance of MG class A, B and C, which needs point-to-point communication of various size of data from 8B to hundreds of KB. FMPL gains 10% to 30% performance improvement for class A and B, while 3% to 10% for class C. Since MG has many message transfers of the small size, the performance improvement is greater than the case of FT. Figure 12 shows the breakdown of computation and communication in execution time of class B. With 16 processes, the communication overhead is about 36% in MPI, while 9% in FMPL.

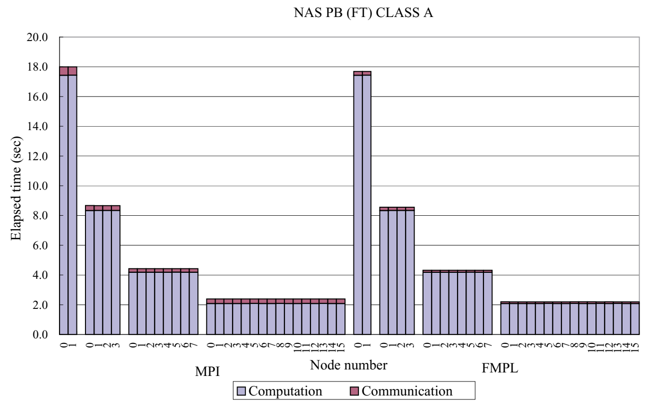


Figure 11: Breakdown of FT class A

Table 4: Performance of MG class A, B and C [sec.]

#nodes	Class A		Class B		Class C	
	MPI	FMPL	MPI	FMPL	MPI	FMPL
2	1.02	0.95	4.81	4.45	32.2	31.5
4	0.59	0.53	2.80	2.50	16.8	16.4
8	0.37	0.31	1.74	1.46	8.9	8.6
16	0.26	0.19	1.22	0.86	5.1	4.6

5.6 PDGEMM

PDGEMM is a subroutine for parallel general matrix-matrix multiplication $C = \alpha\mathrm{op}(A)\mathrm{op}(B) + \beta C$ in PBLAS, where A, B and C are $M \times K$, $K \times N$ and $M \times N$ distributed matrices, respectively, and $\mathrm{op}(A) = A$ or A^T. Table 5 shows the performance result of PDGEMM with $M = N = K = 640$ and a block size of 40. We do not apply a transpose operation for both A and B. The performance improvement of the broadcast depends on the message size as described in Subsection 5.3. BLACS-FMPL achieves more than five times improvement using 16 nodes, in which the message size is about 50KB. Figure 13 shows the breakdown of execution time of PDGEMM. Computational time

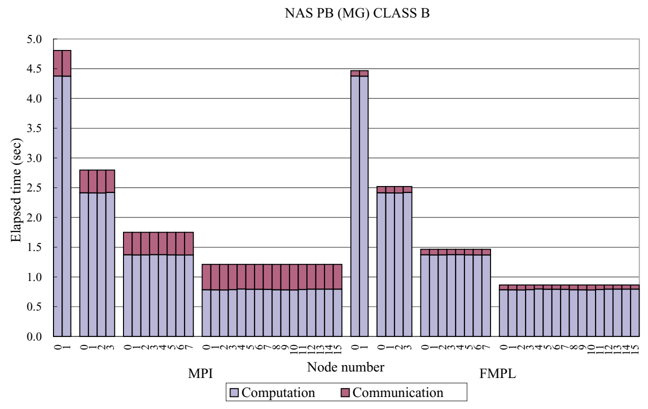


Figure 12: Breakdown of MG class B

Table 5: **Performance of PDGEMM** ($M = N = K = 640$) **[GFlops]**

#nodes	MPIBLACS	BLACS-FMPL	Message size
2	5.64	6.27	200KB
4	7.87	10.74	100KB
8	6.18	17.24	100KB, 50KB
16	4.85	24.45	50KB

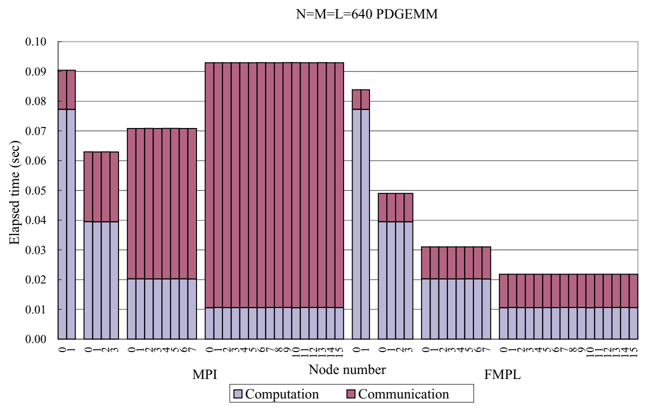


Figure 13: Breakdown of PDGEMM ($M = N = K = 640$)

is almost equal between BLACS-FMPL and MPIBLACS, however MPIBLACS incurs significantly large communication overhead especially with 16 processes. The problem size is obviously too small for the 16-node system of the SR8000, however, BLACS-FMPL exploits the performance of hardware broadcast and reduces the communication overhead.

6. RELATED WORK

Active Messages[20] (AM) and Fast Messages[13] (FM) are a fast communication mechanism, where each message includes the address of a message handler that processes the message body, and will be handled by the receiver just calling the handler without investigation of message tag and so on. AM and FM do not have a feature of local synchronization with respect to execution of message handler, MPI-AM[21] and MPI-FM[7] that are based on MPICH[5] require introducing a queue for message matching on the receiver side, while FMPL message-passing approach removes the queue operations for point-to-point communication, although this FMPL approach can also be implemented on AM and FM. MPI-FM utilizes *upcall* and *gather* techniques to eliminate memory-to-memory copy on both sender and receiver sides.

PMv2[17, 19, 16] facilitates low-cost message-passing communication for a heterogeneous environment of Myrinet, Ethernet and shmem, and optionally remote memory operations, using multiple *channels* for virtualized networks. Message-passing communication of PMv2 does not have a message tag for message matching but provides a FIFO queue within a library space between any two processes. MPICH-SCore[17] is implemented using MPICH and PMv2, which also needs a queue for message matching on the receiver side. For rendezvous protocol, MPICH-SCore utilizes remote memory write for zero-copy point-to-point communication.

MPI-EMX[18] is designed and implemented on the ETL EM-X hybrid dataflow machine[6], which supports sender-side and receiver-side message matching protocols including dynamic protocol changing that needs two message queues on each side and additional control messages. MPI-EMX utilizes not only remote memory operations and remote thread invocations but also I-structures[1] for message matching without polling and interruption.

MPI/MBCF[9] is an MPI implementation on top of the MBCF[8], which also supports sender-side and receiver-side message matching protocols using memory-based FIFO facility provided by the MBCF.

MPICH[5] is the most prevalent implementation of the MPI standard, which has three protocols; eager, rendezvous and get, for the point-to-point communication. In every protocol, control messages are sent from a sender to a receiver at first, and the message matching is done by the receiver. Even when MPI_Irecv is issued in advance, the communication latency is greater than the sender-side matching implementation because the sender should inquire of the receiver at send-time. Eager protocol is useful especially when MPI_Send is issued in advance, though additional copying overhead and reservation overhead for remote temporary buffer are necessary.

MPIAP[14, 15] is a native implementation on Fujitsu multicomputer AP1000, AP1000+ and AP3000, which supports eager and get protocols using remote memory read using receiver-side message matching.

7. SUMMARY AND FUTURE WORK

We have been developing a fast message-passing library FMPL and BLACS-FMPL. FMPL provides a general-purpose point-to-point communication using remote memory operations and low-cost local synchronization without message queues, and collective communication using FMPL pint-to-pint communication and communication hardware mechanism. FMPL is designed for building a more highly functional message-passing library like BLACS as well as applications that need maximum performance.

On SR8000, local synchronization is implemented by the TCW-reused remote DMA transfer from a receiver to a sender. This implementation achieves the 8-byte latency of 12.8μsec., while MPI achieves 20μsec. When the corresponding receive call has already issued, the latency of FMPL is 9.9μsec. Throughput of the point-to-point communication is advantageous to the FMPL especially when the message size is between several KB and several hundred KB. For collective communication, FMPL exploits hardware communication support and improves the performance of MPI. FMPL broadcast improves the broadcast throughput 57%. NAS parallel benchmarks show the improvement depends on how the program includes short messages compared with the computation. Using 16 nodes, MG class B are improved 30%. Evaluation of PDGEMM shows the difference of communication performance between BLACS-FMPL and MPIBLACS. BLACS-FMPL improves a factor of 7 with $M = N = K = 640$ and 16 nodes.

We now continue to develop a complete set of optimized BLACS-FMPL, which will be distributed from TACC. FMPL is not dedicated to SR8000, and we plan to implement FMPL on other platforms.

8. ACKNOWLEDGMENTS

We are grateful to Prof. K. Taura at the University of Tokyo for discussion about point-to-point communication with a wildcard. We are thankful for plenty of valuable comments from anonymous reviewers. We appreciate precious discussion with members of the high-end computing group of Information Technology Research Institute, AIST. We also give great thanks for Dr. K. Terakura, director of Tsukuba Advanced Computing Center, AIST for supporting this research.

9. REFERENCES

[1] R. Arvind, S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[3] T. Boku, K. Itakura, H. Nakamura, and K. Nakazawa. CP-PACS: a massively parallel processor for large scale scientific calculations. In *Proceedings of the 1997 International Conference on Supercomputing (ICS97)*, pages 108–115. ACM, 1997.

[4] J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.1. Technical Report CS-95-281, University of Tennessee, 1995.

[5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[6] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 14–23, 1995.

[7] M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[8] T. Matsumoto and K. Hiraki. MBCF: A protected and virtualized high-speed user-level memory-based communication facility. In *Proceedings of the 1998 International Conference on Supercomputing (ICS98)*, pages 259–266. ACM, July 1998.

[9] K. Morimoto, T. Matsumoto, and K. Hiraki. Implementing MPI with the memory-based communication facilities on the SSS-CORE operating system. In *Proceedings of 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98)*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230, 1998.

[10] H. Nakamura, H. Imori, K. Nakazawa, T. Boku, I. Nakata, Y. Yamashita, H. Wada, and Y. Inagami. A scalar architecture for pseudo vector processing based on slide-windowed registers. In *Proceedings of the 1993 International Conference on Supercomputing (ICS93)*, pages 298–307. ACM, 1993.

[11] netlib. http://www.netlib.org/, http://phase.hpcc.gr.jp/mirrors/netlib/.

[12] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *Proceedings of the 1998 International Conference on Supercomputing (ICS98)*, pages 243–250. ACM, July 1998.

[13] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[14] D. Sitsky and K. Hayashi. Implementing MPI for the Fujitsu AP1000/AP1000+ using polling, interrupts and remote copying. In *Proceedings of Joint Symposium on Parallel Processing 1996 (JSPP'96)*, pages 177–184, June 1996.

[15] D. Sitsky and K. Hayashi. An MPI library which uses polling, interrupts and remote copying for the Fujitsu AP1000+. In *Proceedings of International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'96)*. IEEE, June 1996.

[16] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. The design and evaluation of high performance communication using a gigabit ethernet. In *Proceedings of the 1999 International Conference on Supercomputing (ICS99)*, pages 260–267, 1999.

[17] T. Takahashi, S. Sumimoto, A. Hori, H. Harada, and Y. Ishikawa. PM2: A high performance communication middleware for heterogeneous network environments. In *Proceedings of Conference on High Performance Networking and Computing (SC2000)*, 2000. CD-ROM.

[18] O. Tatebe, Y. Kodama, S. Sekiguchi, and Y. Yamaguchi. Highly efficient implementation of MPI point-to-point communication using remote memory operations. In *Proceedings of the 1998 International Conference on Supercomputing (ICS98)*, pages 267–273. ACM, July 1998.

[19] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In *Proceedings of High-Performance Computing and Networking 97*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717, 1997.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[21] F. C. Wong and D. E. Culler. *Message Passing Interface Implementation on Active Messages.* `http://now.CS.Berkeley.EDU/Fastcomm/MPI/`.