

The Implementation of MPI-2 One-Sided Communication for the NEC SX-5*

Jesper Larsson Träff, Hubert Ritzdorf, Rolf Hempel
C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10
D-53757 Sankt Augustin
Germany

Abstract

We describe the MPI/SX implementation of the MPI-2 standard for *one-sided communication (Remote Memory Access)* for the NEC SX-5 vector supercomputer. MPI/SX is a non-threaded implementation of the full MPI-2 standard. Essential features of the implementation are presented, including the synchronization mechanisms, the handling of communication windows in global shared and in process local memory, as well as the handling of MPI derived datatypes. In comparative benchmarks the data transfer operations for one-sided communication and point-to-point message passing show very similar performance, both when data reside in global shared and when in process local memory. Derived datatypes, which are of particular importance for applications using one-sided communications, impose only a modest overhead and can be used without any significant loss of performance. Thus, the MPI/SX programmer can freely choose either the message passing or the one-sided communication model, whichever is most convenient for the given application.

1 Introduction

One of the major extensions of MPI-2 to MPI is the introduction of a model for *remote memory access* (RMA) [3]. In the MPI model for RMA, called *one-sided communication*, a process can be allowed to access dedicated segments of memory of another process for reading, writing or updating, without the explicit participation of the other process. Such remote accesses take effect after an appropriate synchronization operation is performed. RMA is often thought of as restricted to shared-memory multiprocessors. That this is not necessarily so is demonstrated in the MPI-2 model of one-sided communication, which mainly targets distributed memory machines and can be implemented with relatively little extra overhead. A distinguishing feature of RMA is that only one process is responsible for initiating communication between two processes, and must alone supply all necessary parameters for the

*0-7803-9802-5/2000/\$10.00 © 2000 IEEE

communication operation. In MPI-2 terminology the initiating process is called the *origin*, while the process whose memory is being accessed remotely is called the *target*. Contrast this to point-to-point message passing where both processes are explicitly involved in the communication. Which model is most suited for a given problem is determined by the algorithmic structure of the application. An important effect of one-sided communication is that communication is explicitly separated from synchronization, allowing for synchronization overhead to be amortized over several communication operations. This feature can be exploited in the efficient implementation of bulk-synchronous parallel (BSP) algorithms, where remote updates are only required to be effective at certain synchronization points [2, 10].

In this paper we describe the implementation of the MPI-2 one-sided communications within the MPI/SX software product for the NEC SX-5 vector supercomputer. The SX-5 is a hierarchical multicomputer composed of shared-memory nodes with up to sixteen vector processors each. The nodes are interconnected via a high-performance crossbar switch (IXS). Each link has a bandwidth of 8 GBytes/second in both directions. The inter-node data transfer is handled by the IXS hardware and takes place concurrently with other CPU activities. Each processor has a peak floating point performance of 8 GFlops (128 GFlops for a sixteen processor node) and a bandwidth to main memory of 32 GBytes/second. For performance reasons cache coherency is not guaranteed. MPI/SX, detailed in [5], is a proprietary, highly optimized, non-threaded implementation of MPI [3, 8], and was originally developed from MPICH [4].

In MPI/SX, MPI processes are mapped onto separate Unix-processes, each with its own private (swappable) address space. We refer to memory allocated in the private address space as *process local memory*. For one-sided communication, processes must logically expose part of their memory to RMA by other processes. This concept is realized in MPI-2 by the definition of communication *windows*. The MPI-2 standard provides an allocator function for window memory, which allows for optimizations on real shared memory systems by appropriate placement of window memory. On the SX-5, this is done by using the Global Memory (GM) facility to create physically shared objects residing in *global shared memory*. The GM areas are shared not only among processes on a single SX-5 node, but can also be accessed by special copy functions across nodes via the IXS. This feature is exploited in the MPI/SX implementation wherever possible.

When the target window has been allocated in global shared memory most RMA is done by shared memory copy operations by the origin alone. According to the MPI-2 standard, the application programmer may also use process local memory for communication windows. In this case the target has to be actively involved in the communication, and one-sided communication is mapped to asynchronous, non-blocking message passing, reusing the machinery of the point-to-point implementation. In the MPI/SX implementation one-sided communication calls will block until the target window has become exposed, that is until a proper synchronization call has been performed by the target. Thus no buffering of messages is necessary. This behavior can be tuned so as to postpone blocking of one-sided communication calls, eventually until the synchronization call performed at the origin.

So far, there are few complete implementations of the MPI-2 chapter on one-sided communication, and even fewer of the full MPI-2 standard. MPI/SX implements the full MPI-2 standard starting with software release R10.2. The MPI-2 implementation for the Fujitsu

VPP5000 vector-parallel computer is outlined in [1], with some attention to one-sided communication, but performance figures for this part were not given. A multi-threaded implementation of one-sided communication for the experimental Windows NT MPI version WMPI is described in [6]. The authors do not give any performance figures either. They discuss various possible optimizations concerning for example the handling of derived datatypes. These and other optimizations are incorporated in the MPI/SX implementation. Partial implementations of MPI-2 one-sided communication have been developed for the parallel systems of Hitachi, HP, IBM (see, for example, [7]), SGI, Cray and Sun.

2 The MPI-2 remote memory access mechanism

The memory that each process exposes to other processes is managed in a data structure called a (local) *window*. A *window object* is the set of windows of processes in the same communication domain (MPI communicator). A window has attributes consisting of base address of the exposed memory segment (in local or global memory), size of the segment, and a displacement unit used for computing offsets within the window. Global shared memory must be allocated with the MPI-2 call `MPI_Alloc_mem`. Local memory allocation can be done with any available allocator function, or forced by setting the key `local_mem` in the `MPI_Info` argument to `MPI_Alloc_mem` to `true`.

A window object is created by the collective communication call `MPI_Win_create`. Each process in the communication domain supplies its local window attributes as parameters to the call. In the MPI/SX implementation the window object is represented in each process by a data structure similar to that of a communicator. It stores the attributes of all local windows, the group of processes to which access is enabled, and for which access is permitted, as well as the locking status of each window.

One-sided communication takes place within so-called *epochs*. In an *exposure epoch* a process has exposed its window for access by (certain) other processes. In an *access epoch* a process is allowed to access the windows of (certain) other processes. An epoch is opened and closed by a synchronization operation, in which one, more, or all processes in the communication domain take part. There are three kinds of synchronization mechanisms in MPI-2:

Fence The fence is a collective operation involving all processes sharing a window object. An `MPI_Win_fence` call closes a preceding epoch and opens a new epoch. In the new epoch each process is exposed to, and has access to, all other processes. When an epoch is closed by a fence, all RMA requests and corresponding target actions must be guaranteed to have completed at origin and target. The fence mode of operation corresponds closely to the BSP model, and is appropriate for applications where each process may need to access memory of many other processes.

Unless it is known that no epochs precede the fence call (assertion `MPI_MODE_NOPRECEDE` to the `MPI_Win_fence` call), all pending RMA requests are completed locally. To ensure also completion of all target actions, each process must compute the number of RMA operations for which it is the target, and for which some action is required. This is done by an `MPI_Allreduce` summation operation with each process supplying the number of

RMA requests submitted to each of the other processes. Each process then locally waits for completion of the computed number of pending actions. If the local window is in global shared memory, a local cache clear operation is needed to ensure cache consistency. It is not performed if the previous fence asserted that no write or update operations had this window as target (assertion `MPI_MODE_NOPUT`). In order to ensure, upon return from the `MPI_Win_fence` call, that for each origin all target action has indeed indeed completed, a barrier synchronization is performed with `MPI_Barrier`. Unless specified otherwise (assertion `MPI_MODE_NOSUCCEED`), the fence opens the next epoch by enabling all processes in the communication domain of the window for access.

Dedicated For applications where each process communicates with only few other processes, synchronization can be restricted to the origin-target pairs actually exchanging information. A process explicitly exposes itself to a group of other processes by a `MPI_Win_post` call, which is matched by a `MPI_Win_start` call by each of the processes demanding access to the memory of the exposed process. The access epoch is closed again by an `MPI_Win_complete` call. When this call completes, all RMA requests must be guaranteed to have completed at the origin, but not necessarily at the target. The exposure epoch is closed by an `MPI_Win_wait` call. When the call completes, all RMA requests are guaranteed to have completed at both the target and origin.

The dedicated synchronization mechanism is implemented as suggested by the standard [3, page 118], with `MPI_Win_post` and `MPI_Win_complete` sending specially tagged messages to be received by the matching `MPI_Win_start` and `MPI_Win_wait` calls. The messages sent upon opening of an exposure epoch by `MPI_Win_Post` are dispensed with if it is known that the corresponding `MPI_Win_start` calls have not yet been performed (asserted by `MPI_MODE_NOCHECK` to both `MPI_Win_post` and `MPI_Win_start` calls). The message sent by the `MPI_Win_complete` call to each target process includes the number of RMA requests with that process as target. The `MPI_Win_wait` call waits for the complete-messages from all processes having access to the process, and uses the sum of the received counts to ensure completion of all target action. If the local window is in global shared memory, a clear cache is again performed to ensure cache consistency, unless the `MPI_Win_post` call guaranteed that no write or update operations had this process as target (assertion `MPI_MODE_NOPUT`).

Locks The last synchronization mechanism provides for so-called *passive target* synchronization. A process can open an access epoch at one other process by requesting either a shared or an exclusive lock. The access epoch is closed by a corresponding unlock call, after which all RMA requests must be guaranteed to have completed at both origin and target.

Since lock and unlock synchronization calls are not matched by corresponding MPI calls in the user program at the target, special communication operations are needed to request asynchronous (hidden) action at the target. The MPI/SX implementation internally uses two such (non-blocking) requests each with additional information for the required target action. These internal messages are intercepted and handled by the MPI/SX mechanism which regularly (at every MPI-communication call) checks for incoming messages.

The MPI/SX implementation of locks is fair in the sense that a process requesting a lock will eventually be granted access to the target window. Lock requests which cannot

immediately be granted do not block at the origin, but a subsequent RMA communication call will block. A lock is always granted if the target window is not locked by other processes. If a shared lock is requested, it is granted if the window is locked by other shared locks and there are no pending (exclusive) lock requests. In all other cases, the request is queued. The next queued request(s) is processed at unlock. If the first pending request is for an exclusive lock, only this request is granted. Otherwise, all pending shared lock requests up to the next exclusive request are granted.

3 One-sided communication operations

In the MPI/SX implementation a one-sided communication call first checks if the request is legal: the target must belong to the group of processes to which access is permitted, and the specified target address range must be fully contained within the target window (see [3, page 101]). These conditions can easily be checked locally using the data stored in the local window object. The call then waits for the target to actually become ready, and thus may block.

Each communication request is handled separately, i.e. requests with the same target are not merged, although this could in some cases lead to a reduction of message latency (see Section 6).

MPI/SX Global shared memory: If the target window is in global shared memory of the same node, the origin directly copies the data to or from the requested target locations. In the case where origin and target are the same process, straight-forward memory copy is used. For contiguous origin and target segments in global memory of different nodes the IXS hardware is used to copy the data if both segments are properly (4-byte) aligned. In other cases, which should be rare, the local memory mechanism (see below) is used for the remote access.

When direct memory copy is used for RMA, only one processor is required for the transfer and can exploit the full memory bandwidth for the operation. In the MPI/SX implementation of point-to-point message passing the same optimization is applied for message buffers in global memory, so that in terms of actual transfer speed both mechanisms should be quite similar. This is evidenced by the comparison in Section 5.

It turned out to be most efficient to always use the local memory mechanism for remote updates (`MPI_Accumulate`) in order to guarantee the atomicity requirement imposed by the MPI-2 standard. As a consequence, concurrent updates to the same target window are atomic at the request level. This is more than the standard requires [3, page 133] (atomicity at the level of single data elements), but it avoids expensive locks on target data elements that would otherwise have been necessary.

MPI/SX Process local memory: For target windows in process local memory, direct memory copy cannot be used. Instead the communication is mapped to message passing. First, the data communication is started with a non-blocking send (for `MPI_Put` and

`MPI_Accumulate`) or receive (for `MPI_Get`). Then, as discussed above for the implementation of locks, a special control message is sent to initiate the proper action on the target side of the RMA operation. The control message is handled by the mechanism which regularly checks for incoming messages. Since this check can be invoked recursively, each pair of corresponding send-receive operations must have a unique tag in order to prevent messages from getting mixed up. This tag is generated at the origin and sent with the control message.

Message counting is used to ensure completion of RMA requests at both origin and target. To this end, special non-blocking send and receive routines are used that signal completion by incrementing a counter given as parameter to the call. At the origin, the total number of RMA requests in an epoch is counted as they are issued. At synchronization, local completion is guaranteed when the number of processed RMA requests reaches this count. Similarly, completion on the target side is ensured by waiting for the target counter to reach the number of requests computed when an epoch is closed.

The use of non-blocking communication for all one-sided communication gives a high degree of asynchronicity which improves the distribution of the one-sided transactions over the whole epoch. The potential drawback that internal message queues may become too long can be overcome by emptying the queues when the number of pending RMA requests reaches a certain threshold. Corresponding parameters for controlling this behavior are set at compile-time of the MPI/SX library.

4 Handling of derived datatypes

Derived datatypes are MPI's way of describing non-contiguous layouts of data in memory. Since all functions for constructing MPI datatypes have local completion semantics, datatypes defined in one process cannot be used in other processes. This has some obvious consequences for the implementation of one-sided communication.

Target window in global shared memory: In the best case, i.e. if the data type is a basic type or otherwise contiguous, the data is transferred by the origin with a direct memory copy. If the same derived datatype is specified for both origin and target, the direct copying is efficiently vectorized using the idea presented in [9]. Otherwise, the data transfer requires packing and unpacking via an intermediate buffer, which again is done efficiently using the flattening-on-the-fly technique in [9].

Target window in process local memory: For windows in local memory, the communication requires action on the target side, and type information must be sent along with the data. This is done by sending a light-weight representation of the type tree, either as part of the control message (if small enough) or as a separate message. When such a type tree representation is received at the target, an internal datatype is reconstructed, to be used in the following send or receive communication call. The reconstructed type is stored (cached) in a table, so that each user defined type is sent at most once, namely with the first communication operation using that type. The cache is a hash table having the global rank of the origin and the index of the type at the origin as combined key. Freeing a type at

the origin poses no problem; if the type index is later reused, the now obsolete type in the target cache is simply deleted. The light-weight representation is built up incrementally as the type is being constructed. The cost for each MPI type constructor is a few extra memory copy operations for copying the light-weight representation of the constituent types.

5 Performance

The difference between the underlying programming models makes it difficult to compare the performance of one-sided communication with that of point-to-point message passing. Many benchmarks favor one and discriminate the other model. The ubiquitous ping-pong message-passing test, for example, requires a synchronization after every single message, and the communication endpoints are fixed. Therefore, it is best programmed in point-to-point style. In an ideal application for one-sided communication, the destination of each data is known at the origin only, and synchronizations are required infrequently.

If both programming models are implemented equally well, the choice of the best-suited paradigm can be made based on its convenience for the given application. To evaluate the quality of the implementation, we identified a loosely synchronized data exchange of processes with their "local neighborhood" as a typical communication pattern that can be programmed either way. Since the (fixed) communication pattern is known in every process and several messages are passed between synchronization points, in this case there is no a priori performance (or convenience) advantage for one programming paradigm over the other.

The test program (written in C) implements a simple exchange pattern. Let p be the number of processes, and n a given number of neighbors, $n \leq p$. For $j = 1, \dots, n$, process i supplies a block of a given length to process $(i + j) \bmod p$, and requests a block of the same length from process $(i - j) \bmod p$. We test 7 implementations of this pattern:

Put-Fence: Data is supplied with `MPI_Put`, epoch opened and closed with `MPI_Win_fence`.

Get-Fence: Data is requested with `MPI_Get`, epoch opened and closed with `MPI_Win_fence`.

Put-Ded: Data is supplied with `MPI_Put`, access epoch opened with `MPI_Win_start`, exposure epoch with `MPI_Win_post`, and closed with `MPI_Win_complete` and `MPI_Win_wait` respectively.

Get-Ded: Data is requested with `MPI_Get`, access epoch opened with `MPI_Win_start`, exposure epoch with `MPI_Win_post`, and closed with `MPI_Win_complete` and `MPI_Win_wait` respectively.

Put-Lock: Data is supplied with `MPI_Put`, access epoch at target opened for exclusive access with `MPI_Win_lock` and closed with `MPI_Win_unlock`.

Get-Lock: Data is requested with `MPI_Get`, access epoch at target opened for exclusive access with `MPI_Win_lock` and closed with `MPI_Win_unlock`.

Sendrecv: Data is supplied and requested with `MPI_Sendrecv`.

Window	p	Fence-1	Fence- p	Ded-1	Ded- p	Lock-1	Lock- p
global	2	67.20	67.21	43.62	44.74	67.77	129.85
	4	85.32	87.40	44.12	80.00	66.97	252.51
	6	91.91	90.98	45.15	132.13	69.13	370.00
	8	99.03	100.93	44.81	164.57	66.97	500.28
local	2	51.91	54.72	24.46	25.82	44.42	79.72
	4	60.57	58.65	25.41	59.50	44.53	152.24
	6	74.46	74.78	25.93	105.38	45.65	238.09
	8	80.68	80.88	25.58	142.19	44.40	297.88

Table 1: Synchronization overhead for one-sided communication on the SX-5. Measurement with $p = 2, 4, 6$ processors, and 1 or p neighbors. All times are in microseconds.

In all cases an arbitrary MPI (derived) datatype can be used. Each variant is repeated for a predefined number of epochs (for the figures presented here, 25 epochs were used), with increasing block lengths, and the SX-5 wall clock time in microseconds for the fastest epoch is reported. Block sizes of single messages are given in bytes.

To estimate the overhead caused by the three different synchronization mechanisms, we first measured the time for opening and closing epochs with fence, dedicated and lock synchronization. Table 1 lists the synchronization costs for all three mechanisms, with $p = 2, 4, 6, 8$ processors, and $n = 1$ and $n = p$ neighbors. The communication window is either in global shared or process local memory. Not surprisingly, the time for fence synchronization is independent of n , but increases (logarithmically) with p . For both dedicated and lock synchronization the time increases with n , but is roughly independent of p . In the latter case, with $n > 1$, the target windows are locked one after the other. The extra (constant) overhead for global shared memory is due to the time needed to clear the cache which is necessary in this case only.

Results of the exchange tests with contiguous blocks of type MPI_INT using eight processes, each one having one or eight neighbors, are shown in Tables 2 and 3, respectively. The results for Put-Fence and Sendrecv for windows in global shared and process local memory are compared graphically in Figures 1 and 2. The tests were performed with windows in both global shared and process local memory.

As discussed in Section 3, for the data transport we did not expect a significant performance difference between point-to-point message passing and one-sided communication. A comparison of the synchronization times in table 1 with the total execution times for Sendrecv in tables 2 and 3, however, shows that the synchronization overhead for one-sided communication is significant and can only be amortized as both the block size and the number of neighbors, i.e. the number of one-sided communication operations per synchronization operation, increase. In table 2 (one neighbor), the break-even point is not even reached for a block length of four Megabytes, while in the case of eight neighbors (Table 3) one-sided communication can be slightly faster for longer messages (break-even slightly above 64 KBytes). This holds for windows in global shared memory. In the case of process local memory, where one-sided communication is mapped to non-blocking send and receive operations, break-even

Blocksize	Put-Fence	Get-Fence	Put-Ded	Get-Ded	Put-Lock	Get-Lock	Sendrecv
1024	100.97	102.58	50.65	50.61	74.85	78.23	8.01
4096	102.07	103.43	50.73	50.79	76.00	79.07	11.11
16384	102.16	103.76	51.20	51.07	76.58	79.33	11.91
65536	103.25	108.70	53.66	53.46	78.79	81.98	16.24
262144	116.03	116.69	63.21	62.89	88.41	91.37	40.06
1048576	151.74	153.36	101.22	100.87	126.17	123.81	77.70
4194304	301.36	303.39	251.33	250.10	272.04	278.45	227.24
1024	107.19	137.87	54.76	71.16	71.98	88.26	8.01
4096	116.71	146.18	61.38	79.93	86.16	100.35	11.11
16384	117.78	145.99	62.32	80.67	87.82	100.77	11.91
65536	121.80	151.80	66.75	85.39	93.45	105.82	16.24
262144	182.05	196.97	127.39	147.45	155.47	156.70	62.98
1048576	259.84	275.33	205.51	223.82	232.62	232.94	141.59
4194304	600.90	665.35	533.99	559.44	448.32	556.41	470.33

Table 2: Wall clock times in microseconds for the exchange test with contiguous data, eight processes with one neighbor each. Windows are either in global shared (upper half) or process local memory (lower half).

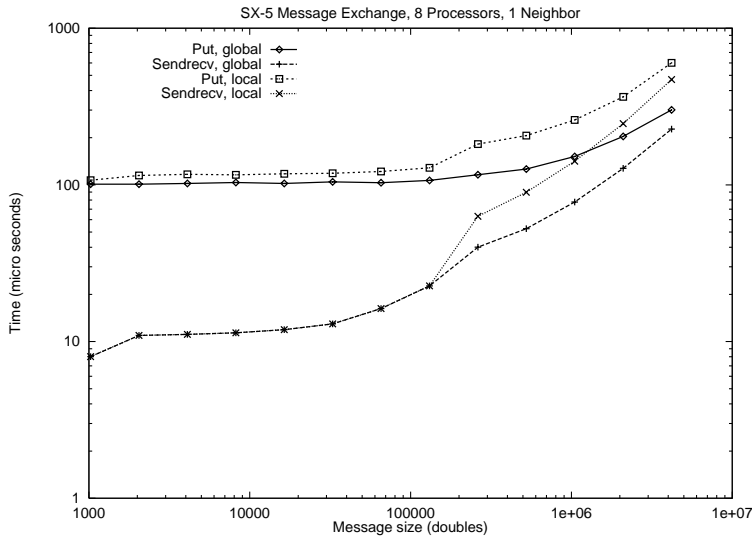


Figure 1: Wall clock times in microseconds for the Put-Fence and Sendrecv exchange tests with contiguous data, eight processes with one neighbor each. Windows are either in global shared or process local memory. Note that scaling is logarithmic on both axes.

Blocksize	Put-Fence	Get-Fence	Put-Ded	Get-Ded	Sendrecv
1024	111.29	110.68	182.56	159.71	61.73
4096	109.18	114.30	183.11	159.30	85.43
16384	119.50	122.15	186.53	162.15	91.45
65536	133.17	132.30	209.99	178.50	125.95
262144	211.02	212.29	286.21	256.16	301.33
1048576	520.62	517.19	594.47	564.11	610.17
4194304	1741.26	1729.79	1805.72	1776.90	1831.98
1024	212.01	285.21	300.78	337.24	61.73
4096	233.23	339.22	307.75	390.30	85.43
16384	237.33	351.08	316.30	396.25	91.45
65536	272.02	396.96	351.06	432.83	125.95
262144	672.06	712.69	743.07	824.97	499.89
1048576	1342.46	1306.62	1504.10	1550.51	1130.89
4194304	5005.97	4493.08	5240.78	4996.72	4067.61

Table 3: Wall clock times in microseconds for the exchange test with contiguous data, eight processes with eight neighbors each. Windows are either in global shared (upper half) or process local memory (lower half).

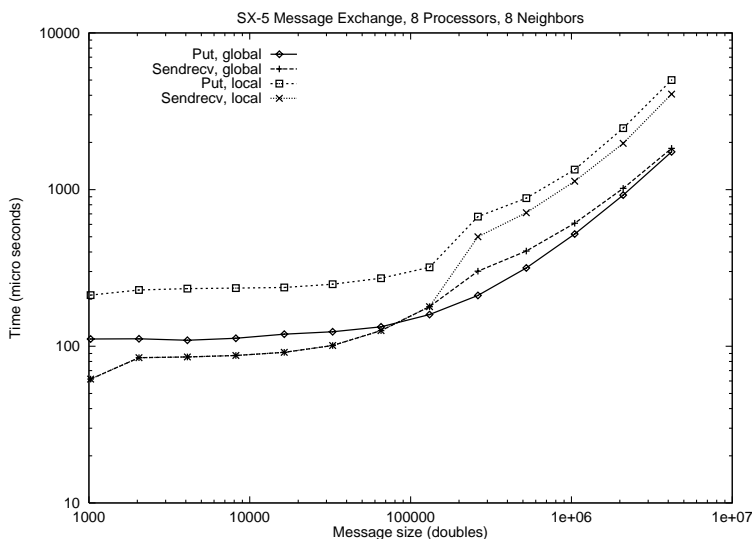


Figure 2: Wall clock times in microseconds for the Put-Fence and Sendrecv exchange tests with contiguous data, eight processes with eight neighbors each. Windows are either in global shared or process local memory. Note that scaling is logarithmic on both axes.

Blocksize	Put, Cache		Put, No Cache		Sendrecv	
	Vector	Struct	Vector	Struct	Vector	Struct
1024	120.12	338.78	156.44	566.02	46.47	235.45
4096	121.44	366.26	159.80	593.53	52.60	270.91
16384	124.25	366.45	159.28	596.29	55.52	271.68
65536	139.97	544.94	174.68	794.68	73.04	436.71
262144	283.24	1835.81	314.41	2052.56	179.11	1707.75

Table 4: Exchange test with blocks of non-contiguous data. For both vector and struct datatypes, wall clock times in microseconds are listed for Put-exchange (with dedicated synchronization) and Sendrecv-exchange. Eight processes have one neighbor each. The window is in process local memory.

cannot be reached because of the explicit synchronization in the RMA model. The one-sided communication model may, however, have an advantage in applications where the communication targets are only known by the origin. In such applications a potentially costly exchange phase can be avoided.

To evaluate the effect of datatype caching (see Section 4), Table 4 shows the results of the exchange test for structured, non-contiguous data with windows in process local memory. Two derived datatypes were used: a vector of 4 blocks, each consisting of one `MPI_INT`, replicated in a contiguous type `256/sizeof(int)` times for a total size of 1 KBytes. The light-weight representation of this data type is small enough to be sent as part of the control message. The second datatype, although having the same total size of 1 KBytes, has a large light-weight representation. It is a structure of `512/sizeof(int)` blocks, each consisting of two `MPI_INT`. The table gives the performance both with and without datatype caching. Especially for the large structured type, there is a significant overhead if type caching is not used, and the light-weight type representation is sent with each communication operation.

6 Concluding remarks

We described the MPI/SX implementation of the MPI-2 chapter on one-sided communication, and compared its basic performance with that of point-to-point message passing using a simple exchange test. The data transfer for both programming models exhibit similar performance, but the explicit synchronization in the one-sided case adds a significant overhead which must be amortized by having many communications calls within an epoch. We conclude that the choice of the programming model to be used for a given application should be based on its particular communication pattern. The implementation within MPI/SX exploits the shared memory of the SX-5 as efficiently as possible and makes use of dedicated, efficient communication routines for local memory data transfers. Efficient handling of derived datatypes results from the use of light-weight representations and from caching datatypes at the target process to avoid unnecessary transfers, together with efficiently vectorized pack and unpack routines. The locking mechanism relies on asynchronous target activity.

The implementation as presented here does not attempt to combine and/or re-schedule

one-sided data transfers within an epoch. This could provide for more efficient utilization of hardware resources such as the IXS switch, but could also have negative performance effects by postponing all communication to the synchronization point. A detailed study of the benefit of such non-trivial algorithmic optimizations will be deferred until actual MPI application programs using one-sided communication become available. In the meantime, the user can to some extent combine transfers by constructing appropriate derived datatypes.

Acknowledgments

We thank Takeshi Hayasaka of NEC Tokyo and Thomas Brandes of GMD for feedback on the implementation at various stages of development. We are also grateful to Kenichi Hori and Yoshiki Seo of NEC Tokyo for useful comments on the manuscript.

References

- [1] N. Asai, T. Kentemich, and P. Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Supercomputing'99*, see <http://www.sc99.org/proceedings/papers/lagier.pdf>.
- [2] M. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Transactions on Computers*, 48(7):670–689, 1999.
- [3] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI –The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.
- [4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- [5] R. Hempel, H. Ritzdorf, and F. Zimmermann. Implementation of MPI on NEC's SX-4 multi-node architecture. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 185–193, 1997.
- [6] F. E. Mourão and J. G. Silva. Implementing MPI's one-sided communications in WMPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 231–238, 1999.
- [7] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI: A new high-performance communication library for the IBM RS/6000 SP. In *1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed*

- Processing (IPPS/SPDP-98)*, pages 260–266, Los Alamitos, 1998. IEEE Computer Society.
- [8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI –The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
- [9] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users’ Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.
- [10] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.