Remote memory access: A case for portable, efficient and library independent parallel programming

---

Alexandros V. Gerbessiotis
Seung-Yeop Lee

Department of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102

Research Report: CS-03-12

New Jersey Institute of Technology

Remote memory access : A case for portable, efficient and library independent parallel programming

Alexandros V. Gerbessiotis and Seung-Yeop Lee *
CS Department
New Jersey Institute of Technology
Newark, NJ 07102
USA.

July 27, 2003

Abstract

In this work we make a strong case for remote memory access (RMA) as the only way to program a parallel computer by proposing a framework that supports RMA in a library independent way. If one uses our approach the parallel code one writes will run transparently under MPI-2 enabled libraries but also bulk-synchronous parallel libraries. The advantage of using RMA is code simplicity, reduced programming complexity, and increased efficiency. We support the latter claims by implementing under this framework a collection of benchmark programs consisting of a communication and synchronization performance assessment program, a dense matrix multiplication algorithm, and two variants of a parallel radix-sort algorithm and examine their performance on a LINUX-based PC cluster under three different RMA enabled libraries: LAM MPI, *BSPlib*, and PUB. We conclude that implementations of such parallel algorithms using RMA communication primitives lead to code that is as efficient as the message-passing equivalent code and in the case of radix-sort substantially more efficient. In addition our work can be used as a comparative study of the relevant capabilities of the three libraries.

*Contact Author: S. Y. Lee, sxl5937@oak.njit.edu

1 Introduction

In the past years several parallel computing models have been proposed such as the CGM [7], LogP [6], BSP [24], and QSM [14] for the design of parallel algorithms and the programming of parallel computers. At the same time a number of parallel libraries have become available that allow portable programming on a variety of parallel hardware platforms. Most of the these libraries are totally independent of these programming models; libraries based on the Message Passing Interface (MPI) such as the freely available LAM MPI [17] and MPICH [22], or commerical ones such as WMPI [5], and other libraries such as the Parallel Virtual Machine (PVM) [9] fall into this category all of which offer extensive library features of several hundred function calls. Other libraries such as the Oxford BSP library [21], *BSPlib*, The Oxford BSP Toolset [16, 23], and PUB-Library [2] (Paderborn University BSP-Library) are tied to a specific programming model and are quite compact thus offering a mere 30-40 function calls.

Parallel programming is still viewed as a complex task not only because it involves understanding and resolving issues such as task parallelism and data distribution but also because it requires effective processor communication and synchronization. Users of popular parallel programming libraries such as the ones based on the Message Passing Interface (MPI) are offered an extensive collection of message passing alternatives to realize two-way interprocessor communication. For example sending information under MPI can be performed by no less than eight different blocking or non-blocking function calls. Receipt of information can be dealt with by a variety of approaches as well that also depends on the sending method used. The choice of multiple methods to perform a single task leads many times to programmer overload and confusion, increasing programming complexity, deadlocked programs, and results in program inefficiencies as the most efficient way to communicate information may not be used. MPI for example offers the programmer more than 200 library functions even though most programmers use only a small subset of these functions.

Remote memory access (RMA) one-sided communication has been available since the introduction of the Cray SHMEM routines. Because of the simplicity of these routines, their elegance and their efficiency, RMA was quickly and successfully adapted into bulk-synchronous parallel libraries such as the Oxford BSP library, *BSPlib*, and PUB-Library. Subsequently it was incorporated into MPI under the MPI-2 standard. The current support of RMA under MPI libraries varies, however. Some freely available libraries such as LAM MPI support it but others do not. Because of this nonuniformity of support programmers stay away from this style of parallel programming because, it is thought, it could lead to non-portable code. Although a programmer is offered at least eight ways to send information through message passing in MPI, there is only one way to send information through RMA by utilizing a `put` instruction; a symmetric `get` instruction is also available. In

BSPlib and PUB-Library the programmer also has two choices for a `put` (or `get`).

In this paper we make a strong case for remote memory access as the one and only way to program a parallel computer. We support our case by first proposing a robust programmatic framework that supports RMA in a library independent, simple, and portable way using the C programming language; extending this framework for C++ and Fortran programming would be easy. We claim that if one uses our approach the parallel code one writes will run transparently not only under MPI-2 RMA enabled libraries including LAM MPI and Critical Software's WMPI but also on bulk-synchronous parallel libraries such as *BSPlib* and PUB-Library thus making one's code library independent as well. We claim that the advantage of using RMA is code cleanliness, reduced programmer confusion and overload and increased code efficiency because two-sided communication is much more tedious than one-sided communication. In addition, the programmer does not need to choose the best way for interprocessor communication as there is only one method to do so. Moreover, a communication library implementor can more easily provide an efficient implementation of such a method rather than optimize eight or more apparently equivalent ones.

We support our case by implementing using the proposed framework three sets of benchmark quality parallel programs: (a) a set of tests that assess the communication and synchronization performance of a parallel hardware platform that will thereafter be referred to as the As suite, (b) a set of parallel dense matrix multiplication programs, the Mult suite, and (c) a set of parallel integer radix-sort programs, the Rdx suite. We then examine the performance of these programs on a LINUX-based PC cluster under three different RMA enabled libraries: LAM MPI, *BSPlib* and PUB-Library. For comparative purposes variants of these programs have been made to work using two-sided communication in LAM-MPI.

Through the As suite, we can test the synchronization capabilities of a parallel platform under each one of the three libraries. We can also test the communication capabilities of a parallel platform under any of these libraries by benchmarking total-exchanges (also known as p-relations) implemented by using a `put` operation in all these libraries and through a variety of `send` operations in LAM MPI. In the Mult suite, we implemented a blocked parallel dense matrix multiplication algorithm using one-sided `put` or `get` functions calls in all three libraries and two-sided `send/receive` library calls in LAM MPI. Finally the Rdx suite, implements two integer radix-sort algorithms. The first algorithm is a straightforward parallelization of ordinary radix-sort for 32-bit integers. An optimized version of this algorithm that allows for round-robin key value-based delayed communication was also implemented. The second algorithm is an enhanced version of the straightforward parallelization where messages before being sent out are combined so that each processor sends and receives one long message.

We carry out this experimental study for one additional reason: it offers a comparison of the communication performance of the three libraries on a cluster of PC workstations under a realistic set of benchmark programs. In that respect our work serves similar but more general objectives than the work of [8] that compares SHMEM and MPI functionality and [18, 19, 20] that evaluate MPI-2 one-sided functions on a variety of high performance parallel machines.

We draw some interesting conclusions first for the case of one-sided communication. In the AS suite, there was some noticeable deterioration of performance in LAM MPI for `put`-based compared to `send`-based communication. In the MULT suite however, the `put`-based implementations were as good as the `send`-based ones; surprisingly enough there was not much difference between `put`-based and `get`-based implementations. In *BSPlib* and PUB-Library, however, these differences were quite noticeable. In the RDX suite, the `put`-based implementations were substantially faster than the `send`-based ones in LAM MPI; for this reason we have not included results for the latter ones. In conclusion, our experiments support the case for one-sided communication. RMA is easier to incorporate in parallel programs, it can be quite efficiently implemented if the overall performance of *BSPlib* could serve as an indicator, and overall, it is quite well implemented in complex libraries such as LAM MPI compared to the two-sided communication primitives.

In addition we make some interesting observations about the performance of the three communication libraries tested. It seems that LAM MPI exhibits the most consistent behavior among the three, even though it may not be the most efficient of the three libraries. For the communication patterns of parallel radix-sort the PUB-Library exhibits perhaps an erratic and inconsistent behavior, a possible indication that is probably not fine-tuned in a scalable way. In the AS suite of tests, *BSPlib* and PUB-Library were more efficient than LAM MPI; *BSPlib* had quite inconsistent default performance that could be improved, however, substantially by setting appropriate values to some communication control variables. PUB-Library slightly exceeded *BSPlib* for small processor configurations ($p = 4$) but it did not scale well otherwise. Some surprising results were observed in the RDX suite. The enhanced algorithm was consistently slower than the straightforward algorithm and timing results for the former are not reported here. It seems that software based message combining may not be a good idea for the average programmer. Under LAM MPI the `put`-based implementations were much faster than the `send`-based ones; for this reason we do not report results for the latter ones. In the RDX suite, *BSPlib* was more efficient than LAM MPI and at least 50% more efficient than PUB-Library. It is for this reason that we conclude that the PUB-Library may not be properly fine-tuned.

Our conclusions seem to consistently confirm our claims. One-sided communication is easier to deal with than two-sided communication in programs. *BSPlib* and PUB-Library have no more than 30-40 library calls and seem to do as good job as a more general library such as LAM MPI. In the case of LAM MPI the single choice of an one-sided function call is always the best or as good a choice as the best of a collection of two-sided alternatives. In complex communication patterns that resemble total-exchanges where each processor sends all other processors multiple messages, one-sided communication seems to outperform two-sided communication at least in LAM MPI. MPI in general and LAM MPI in particular need to be downsized into a smaller set of library calls that are more optimized and give less freedom of choice to the average programmer.

2 A framework for parallel programming

In this section we show how one could write parallel code using remote memory access one-sided communication that is both portable among several parallel platforms but also library independent in the sense that the same program can be compiled with out any rewriting under RMA enabled libraries which include MPI-2 based libraries such as LAM MPI and Critical Software's WMPI, and programming model specific libraries such as *BSPlib* and PUB-Library. Towards this we introduce few generic primitives that one could use for such portable and library independent parallel programming. For the most important of these primitives equivalent MPI-2 translations are given as well as *BSPlib*/PUB translations that adhere to the BSPWorldwide standard [16].

All three libraries considered in this study have similar methods for initializing an SPMD parallel program, terminating it, aborting it, and accessing the number of processors and the processor identification number of an individual processor. In our framework we use the generic wrapper functions AIBEGIN(AINPROCS()), AIEND(), AIABORT(), AINPROCS(), and AIPID() to represent these common functions. Such functions can be mapped easily to individual library function calls. We omit this by noting that the source code of the programs written for this study contains a file named `ai.h` that includes detailed mappings for these functions in all three libraries. For the sake of an example, in LAM MPI, the first of the generic primitives maps for example to an *MPI_Init* (the AINPROCS() argument is an idiosyncratic feature of *BSPlib* that is ignored), the second to an *MPI_Finalize*, the third to an *MPI_Abort*, the fourth to an *MPI_Comm_size*, and the last one to an *MPI_Comm_rank* function call.

We provide two ways for RMA-based programming that we explain in detail below. Suppose that source processor *spid* intends to send information to a destination processor *dpid*. The source information is stored in consecutive memory locations starting at address *srcaddr* and is *len* bytes long. The information is to be stored in processor *dpid* starting at offset of *off* bytes from the

	Comment: AIPUT mapping under *BSPlib*/PUB	
	Our Interface	The *BSPlib*/PUB equivalent code
1.	AIREGISTER(desaddr, len);	bsp_pushregister(desaddr,len);
2.	AIINIT();	;
3.	AICOMMIT();	bsp_sync();
4.	AIPUT(dpid,srcaddr,desaddr,off,len);	bsp_put (dpid,srcaddr,desaddr,off,len);
5.	AISYNC();	;
6.	AIBARRIER();	bsp_sync();
7.	AIDEREGISTER(desaddr);	bsp_popregister(desaddr);

Figure 1: An AIPUT mapping under *BSPlib*/PUB.

starting address *desaddr*. This operation can be realized by issuing a `put` instruction at *spid* that has the following format: *AIPUT(dpid, srcaddr, desaddr, off, len)*.

Certain actions must precede this operation and the operation is guaranteed completion by a second set of actions that take place after the *AIPUT* is issued. The actions preceding the operation involve registration of the destination *desaddr* and the amount of information *len* that will be transferred. Such registration is split into three steps: (a) a registration call *AIREGISTER(desaddr,len)*, followed (b) by a communication initialization call *AIINIT()*, and finally a (c) registration commit call *AICOMMIT()*. Not all three are always needed. The communication initialization call can be located well-after the registration call as long as it precedes the actual communication call *AIPUT*. After the *AIPUT* has been issued, communication is realized by issuing a communication completion call using an *AISYNC()* instruction and concluding the communication by a barrier call *AIBARRIER()*.

This common interface that we provide to the three libraries may look similar to some to the one agreed in [16] for a common programmatic interface for one-sided communication. There are however some differences intended to support the MPI-2 one-sided communication interface.

We show the mapping of our proposed interface to the one supported by both *BSPlib* and PUB-Library for the example above in Figure 1 and to the one supported by LAM MPI in Figure 2.

With reference to Figure 2 we note that RMA operations under MPI-2 are non blocking. The *MPI_Win_fence* call is one of three different mechanisms for marking completion of an one-sided communication in MPI-2. The instruction of line 2 opens a window for communication and closes it by effecting the communication of line 4 by a similar instruction in line 5. A fence is essentially a barrier synchronization, similar to the Cray SHMEM routine shmembarrier. Therefore *AIBARRIER*

	Our Interface	The MPI-2 equivalent code
	COMMENT: AIPUT mapping under MPI-2	
		MPI_Win win;
1.	AIREGISTER(desaddr,len);	MPI_Win_create(desaddr,len,1,MPI_INFO_NULL, MPI_COMM_WOLRD,&win);
2.	AIINIT();	MPI_Win_fence(0,win);
3.	AICOMMIT();	;
4.	AIPUT(dpid,srcaddr,desaddr,off,len)	MPI_PUT(srcaddr,len,MPI_CHAR, dpid,off,len,MPI_CHAR,win);
5.	AISYNC();	MPI_Win_fence(0,win);
6.	AIBARRIER();	MPI_Barrier(MPI_COMM_WORLD);
7.	AIDEREGISTER(desaddr);	;

Figure 2: An AIPUT mapping under MPI-2.

may map to a null instruction in such a case; yet the equivalence under the BSP model, if one insists on it, of the two mappings of AIPUT under *BSPlib*/PUB and MPI-2 still remains. In all cases registration becomes effective after the commit call in line 3.

The introduced example assumes that only one active window is present. In *BSPlib* and PUB-Library all communication within one superstep (segment of computation that can be completed using locally available information) can be associated with a single such window. Under LAM MPI, however, if two communication operations are performed these must be registered and realized sequentially one after the other to a single window or registered to different windows. For this to occur we need to be able to map distinct destination addresses to distinct windows. One way to do this is by what we call "labeled windows" and the mapping of our generic framework to the two sets of libraries is depicted in Figure 3 and Figure 4.

As it is shown in Figure 3 the mapping does not change for the pair of *BSPlib*/PUB if we use "labeled windows". It is more involved in Figure 4 for MPI-2. We first assume that we have elsewhere (eg. initialization of the SPMD program) allocated an array *aiwin* of communication windows of sufficient dimension. Each *desaddr* must uniquely identify such a window by implementing functions *aiinsert(desaddr)*, *aisearch(desaddr)* and *aidelete(desaddr)* that create a unique window index, search for the unique window index of address *desaddr*, and free the unique window index associated with desaddr. Other than that Figure 4 is similar to Figure 2. The mapping for AIGET

	COMMENT: A labeled AIOPUT window mapping under *BSPlib*/PUB	
	Our Interface	The *BSPlib*/PUB equivalent code
1.	AIOREGISTER(desaddr,len);	bsp_pushregister(desaddr,len);
2.	AIOINIT(desaddr);	;
3.	AIOCOMMIT();	bsp_sync();
4.	AIOPUT(dpid,srcaddr,desaddr,off,len)	bsp_put (dpid,srcaddr,desaddr,off,len);
5.	AIOSYNC(desaddr);	;
6.	AIOBARRIER();	bsp_sync();
7.	AIDEREGISTER(desaddr);	bsp_popregister(desaddr);

Figure 3: A labeled AIOPUT window mapping under *BSPlib*/PUB.

	COMMENT: A labeled AIOPUT window mapping under MPI-2	
	Our Interface	The MPI-2 equivalent code
		MPI_Win aiwin[MAX_AI_WINDOWS];
1.	AIOREGISTER(desaddr,len);	MPI_Win_create(desaddr,len,1,MPI_INFO_NULL, MPI_COMM_WOLRD,&aiwin[aiinsert(desaddr)]);
2.	AIOINIT(desaddr);	MPI_Win_fence(0,aiwin[aisearch(desaddr)]);
3.	AIOCOMMIT();	;
4.		
6.	AIOPUT(dpid,srcaddr,desaddr,off,len)	MPI_PUT(srcaddr,len,MPI_CHAR, dpid,off,len,MPI_CHAR,aiwin[aisearch(desaddr)]);
8.	AIOSYNC(desaddr);	MPI_Win_fence(0,aiwin[aisearch(desaddr)]);
9.	AIOBARRIER();	MPI_Barrier(MPI_COMM_WORLD);
10.	AIDEREGISTER(desaddr);	MPI_Win_free(&aiwin[aidelete(desaddr)]);
end AIOPUT UNDER MPI-2		

Figure 4: AIOPUT under MPI-2.

and AIOGET primitives under the two sets of libraries is defined similarly.

With the aforementioned simple common mappings of the different methods employed for one-sided communication in the three libraries, one is able to use a common interface and thus write code that is library independent and works on all three libraries. We have employed this approach not only in developing the code that is described in this paper but also in other work; such code is available at the first author's Web-page.

3 The As suite

The As suite tests the synchronization capabilities of a parallel platform under any library that supports or can be made to support our RMA framework. It also tests the communication capabilities of a parallel platform for a tested communication library by measuring the realization of total-exchanges (also known as p-relations).

The first set of tests in this suite measures latency related performance of the communication library including the cost of barrier synchronization. The tests in this suite are described below.

(1) Test 1 (Empty). This test times a barrier synchronization if it is available in the communication library.

(2) Test 2 (Comp). This test times a barrier synchronization that is preceded by an elementary computational operation; the purpose is to assess what the effect of such a computational operation will be to the time reported by Test 1.

(3) Test 3 (Full). This is the time it takes to perform a total-exchange of p integers followed by a barrier synchronization, where p is the number of available processors. Thus each processor sends and receives p integers. The purpose of this test is to assess what the effect of a simple communication operation that involves all processors will be to the time reported by Test 1.

(4) Test 4 (Simple). This is the time it takes to perform a simple communication of one integer between processor 0 (sender) and processor $p-1$ (receiver) followed by a barrier synchronization. The purpose of this test is to assess what the effect of a very simple communication operation that involves the least number of processors will be to the time reported by Test 1.

(5) Test 5 (Scatter). This is the time it takes for processor 0 to scatter one integer to each of the remaining $p-1$ processors, followed by a barrier synchronization. The purpose

of this test is to compare the time of this test to the timing results reported by Test 3 and Test 4.

These tests can be useful to someone who wants to assess the cost of barrier synchronization and understanding the latency behavior of the combination of the parallel hardware platform and the communication library tested. In addition, these tests can be useful to those who are interested in deriving the bulk-synchronous parallel model characteristics of the hardware platform. We note that in the BSP model [24] a parallel platform is modeled by a tuple (p, L, g), where p is the number of processors, L is the synchronization periodicity, and g the cost of communication per word of information (inverse of router throughput). More formally g is defined so that the cost of realizing an h-relation (an instance of communication where each processors sends and receives at most h equal-sized messages) in continuous message usage (for large h) is gh; for smaller values of h the cost of communication is latency bound by L. Thus these tests can help us determining the value of L.

Finally, the AS suite includes tests that measure the communication performance of the parallel hardware and the communication library under realistic assumptions. We measure such performance by timing total-exchanges where each processor communicates a total of h integers to every other processor (including itself) for increasing value of h. If the communication time for such an operation is t, the cost of communication is considered to be t/h. We perform this test for h that is a power of two starting from a value equal to the number of available processors and doubling it after every completed test. Under all three libraries used in this work the total-exchange is realized by performing a series of `put` instructions. In addition in LAM MPI we realize the total-exchange by using two-sided communication `send/receive` instructions. Among the variety of such `send/receive` combinations available in MPI we timed and report the one that offered the best overall performance; this was the *MPI_Irecv* and *MPI_Send* combination. Thus we are able to compare one-sided and two-sided communication in LAM MPI.

The programming language used in the implementations is ANSI C and the code was tested for scalability and portability on a cluster of 16 dual-processor PC workstations (PentiumII 350Mhz, Redhat Linux 7.1-running, 128MB RAM equipped) with communication performed through 100Mbit 3Com-905B Ethernet cards and a 24-port CISCO Catalyst 2924M-XL-EN switch connecting the workstations. The default GNU Project's gcc compiler is used through the LAM MPI, *BSPlib*, and PUB-Library front-ends, and the source-code is compiled with the `-O3` compiler option set. Timing is obtained through the use of real-time wall-clock time. All results and variables used for communication and computation involve ANSI C `int` 32-bit data types. The code developed for this experimental study is publically available at the author's Web-page [13]. The tested platform is either the 2-node, 4-node, or 16-node PC cluster utilizing only one CPU per node (non-SMP configuration) or both CPUs (SMP configuration). By setting up the cluster this way we also examined the effects of symmetric multiprocessing in parallel programming. Version 1.4 of *BSPlib*, the default version of LAM MPI for Red Hat Linux 7.1 (LAM 6.5.1), and PUB-Library version 8.0 were used in the experiments.

Table 1 shows the synchronization performance of the PC cluster. It seems that LAM MPI gives the worse results in SMP and non SMP configurations. PUB-Library is better than *BSPlib* in Test 3, and in all other tests it is worse or in some cases equivalent (Test 5, non SMP configuration) to *BSPlib*. One interesting observation is that Test1, Test2, Test4, and Test 5 of *BSPlib* are better than those of the other two libraries. The performance of Test 3 in *BSPlib*, and the one that matters most is worse than that of PUB-Library although better than that of LAM MPI.

Communication performance varies with processor size. For $p = 4$ and $p = 8$, PUB-Library seems to be consistently better than *BSPlib* and LAM MPI is slightly worse than the other two if its `send` performance is considered; the `put` based performance seems to be slightly worse. For $p = 16$ and $p = 32$, however, *BSPlib* is more scalable and gives better results but in several cases this requires that the programmer set appropriate values to some communication control variables of *BSPlib*. LAM MPI is close second but is much better tuned, consistent and predictable. The performance of the PUB-Library quite surprisingly is significantly worse than the other two in both SMP and non SMP configurations. In addition it seems that the PUB-Library's performance deteriorates for a range of h that varies between $2K$ and $16K$; for large value it is still at least a factor of two worse than the other two libraries. We note that some tests were not carried over because the corresponding configurations were unavailable (eg. 32-processor non-SMP configuration).

The As suite offers us a first glimpse on the performance of the three libraries in the PC cluster of our experiments. These benchmarks are complemented by the two suites Mult and Rdx that are described in the next few sections.

4 The Mult suite

The Mult suite contains various implementations of a parallel dense matrix multiplication algorithm that is optimal in computation, it is memory efficient but is neither synchronization nor communication efficient. It exhibits however satisfactory computational performance.

A precursor to this algorithm is one that was originally described in [24] in the context of the BSP model of computation. Let two matrices A and B of dimension $n \times n$ be distributed evenly among the $p = \sqrt{p} \times \sqrt{p}$ processors of a parallel computer so that each processor gets a contiguous submatrix of dimension $n/\sqrt{p} \times n/\sqrt{p}$ of A and B and will also compute one identically dimensioned submatrix of the product C of A and B. Let $P = \sqrt{p}$ and $N = n/P$; the algorithm assumes that $p \leq n^2$. In the algorithm described in [24], each processor m in a single round of communication obtains all the required data consisting of P blocks of A and B and computes a result associated with a submatrix of C. The resulting algorithm is neither memory nor communication efficient; for example each processor requires local memory enough to accommodate $2P$ blocks of dimension $N \times N$. We can convert this algorithm into a memory efficient version that only requires additional local memory for two such $N \times N$ blocks by performing the computation in $\sqrt{p}$ rounds of communication so that in each such round a single $N \times N$ block of A and B is communicated and a partial result of a block of C is computed. This algorithm was developed in [12] and depicted below in Figure 5 for processor *pid*. All matrices are stored in a column major format in a one-dimensional array; element $A[i, j]$ is thus $A[j * n + i]$. The communication of lines 5 and 6 can be implemented by `put` or `send` instructions. Lines 5 and 6 guarantee that each processor sends two $N \times N$ and receives two $N \times N$ blocks into a and b that will be required for matrix multiplication. Because of the "pipelined" fashion of the communication, every processor receives the required blocks in line 7 that will be used in the multiplication of line 9. In line 8 a is transposed into a^t to optimize the computation and take advantage of the column oriented storage of a and b. In the standard loop of Canon's algorithm rows of a would be multiplied with columns of b. Since we store a, b in column major orientation, if we transpose a before the multiplication, then the multiplication will involve columns of a^t and b. This requires a reorientation of Canon's matrix multiplication loop which is indicated by the $\times^t$ in line 9. Figure 6 shows how the "pipelined" just in time delivery of a, b can be performed by using `get` operations instead.

We implemented MatMul by using `put` instructions and MatMulG using `get` instructions in all three libraries. In addition, in the case of LAM MPI, we implemented MatMul using *MPI_Irecv* and *MPI_Send* two-sided communication instructions. We performed the experiments in non SMP configurations for $p = 4$ and $p = 16$ and on SMP configurations for $p = 4$, $p = 16$, and $p = 25$. Given that dense matrix multiplication is a communication intensive operation the SMP experiments are expected to stress the communication network and show whether such matrix operations are suitable for SMP clusters as well. The methodology of the As suite was also followed in writing, compiling and executing the Mult tests noting that the involved arrays are ANSI C `double` data type aggregates.

Table 3 contains speedup results and timing results for $p = 1$ for $p = 4, 16$ processors in a non-SMP configuration of the cluster, and Table 4 contains results for SMP configurations of $p = 4, 16, 25$ processors. In both tables we include sequential running time results for $p = 1$; timing results for

begin MATMUL (C_{pid},A_{pid},B_{pid},n,p)

1. $P = \sqrt{p}$; $N = n/P$;
2. $p_i = pid \bmod P$; $\quad p_j = pid/\sqrt{p}$;
3. $C_{pid} = 0$;
4. **for** $0 \leq l < \sqrt{p}$
5. *put/send* A_{pid} *to processor* $((2 * P - p_i - p_j + l) \bmod P) * P + p_i$.
6. *put/send* B_{pid} *to processor* $((2 * P - p_i - p_j + l) \bmod P) + p_j * P$.
7. *Receive blocks a, b* from remote processors.
8. *Transpose* a into a^t;
9. $C_q = C_q + a^t \times^t b$;

Figure 5: Procedure MATMUL.

various processor configurations can be obtained from these and the included speedup figures.

In the non-SMP configuration, for $p = 4$ the communication performance of the three libraries as tested in the AS suite is reflected in the speedup results we obtained. PUB-Library and LAM MPI have almost the same performance with *BSPlib* a close second for the `put` implementations. In `get` implementations LAM MPI is superior to PUB-Library which in turn outperforms *BSPlib*. An interesting observation is that in the PUB-Library and *BSPlib* there is an asymmetry in the performance of the `put` and `get` implementations. In LAM MPI, however, the two exhibit almost the same performance. For $p = 16$ *BSPlib* seems to be more scalable and has the best performance of the three with PUB-Library a close second and LAM MPI an even closer third. The `get` implementation in *BSPlib* seems to be faster than the other two except for large problem sizes where LAM MPI is better. It seems that the library latency of LAM MPI is higher than the other two. The `put` implementation in LAM MPI seems to be as fast as the `send` or `get` implementations.

In the SMP configuration, for $p = 4$ and the `put` implementation LAM MPI and PUB-Library have a slight advantage over *BSPlib* and over the `send` implementation in LAM MPI. The `get` implementation in LAM MPI again is as good as the `put`. The `get` implementation of the other two libraries is much worse than the `put` or the LAM MPI `get` implementation. Similarly to the non-SMP configuration, for $p = 16$ and $p = 25$ processors *BSPlib* seems to have an advantage over the other two libraries for the `put` implementation. Surprisingly however, the `get` implementation in LAM MPI seems to outperform slightly the corresponding `put` implementation, and it is much faster than the other two `get` implementations. The `send` implementation in LAM MPI is just a little faster than the `put` and `get` LAM MPI implementations but slightly slower than the `put`

begin MATMULG (C_{pid},A_{pid},B_{pid},n,p)
1. $P = \sqrt{p}$; $N = n/P$;
2. $p_i = pid \bmod P$; $p_j = pid/\sqrt{p}$;
3. $C_{pid} = 0$;
4. **for** $0 \le l < \sqrt{p}$
5. *get into* a *the* A *block of processor* $((p_i + p_j + l) \bmod P) * P + p_i)$.
6. *get into* b *the* B *block of processor* $((p_i + p_j + l) \bmod P) + p_j * P)$.
7. *Receive blocks* a, b from remote processors.
8. *Transpose* a into a^t;
9. $C_q = C_q + a^t \times^t b$;

Figure 6: Procedure MATMULG.

implementations of *BSPlib* and PUB-Library.

Peak performance wise *BSPlib* and PUB-Library look better than LAM MPI. Consistency wise (`put` vs `get` for example) LAM MPI is solid. As a side note, if we had not performed the transposition in line 8 of Figure 5 or Figure 6 with the corresponding change of line 9, then the performance results we report would have been three to seven times worse compared to the reported ones. The SMP configurations gave satisfactory results compared to the non-SMP configurations thus indicating the suitability of dense matrix computations for SMP clusters.

The low speedup figures for $n = 320$ are quite natural and easily explained through the AS derived figures. Sequential matrix multiplication performs $2n^2(n-1)$ floating point operations. For $n = 320$ and the running time of 0.93 seconds this translates roughly to 1/70 microseconds per floating point operation. Let $A = 1/70$. The sequential time is thus $2n^2(n-1)A$. The parallel time of algorithm MATMUL and MATMULG is $2n^2(n-1)A/p$ for computation and $2(n^2/p)\sqrt{p}g$ for communication if g is the cost in microseconds per `double` of a total-exchange of $2n^2/p$ `double` data types (or $4n^2/p$ `int` data types). The ratio of sequential running time to parallel running time gives the speedup of the parallel execution; this is $p/(1 + \sqrt{p}g/(A(n-1)))$. For $p = 16$, $n = 320$, $A = 1/70$ and g twice the figure (as a `double` is a 64-bit quantity) read from Table 2 for an $h = 4n^2/p = 25600$, we obtain that for the SMP and non-SMP configurations in *BSPlib* and LAM the corresponding speedup figures should be around 8 and 5 for the non-SMP `put` implementation and 6 and 4.6 for the SMP `put` implementation; the actual speedup figures are 10.33 and 5.17 for the former and 7.75 and 5.47 for the latter implementation.

5 The RDX suite

We have finally implemented and studied the performance of a parallel version of the sequential radix-sort algorithm. As opposed to comparison-based sorting algorithms such as quicksort, merge-sort, and heapsort that can sort any input keys, radix-sort and its base algorithm count-sort [3] can sort only keys that take fixed values by counting the occurrences of each key value. In radix-sort we perform a number of rounds of the elementary count-sort algorithm [3]. If we view a 32-bit as four radix-256 digits then radix-sort will require four rounds of a count-sort algorithm. If each key is viewed as two radix-65536 digits, then we will need two rounds of count-sort. For the test range of the experiments to be presented below, the four-round algorithm was faster than the two-round case in all sequential and parallel experiments and it is the one that was parallelized.

A brief description of the parallel radix-sort algorithm is sketched in Figure 7. Parallel RDX-SORT for 32-bit unsigned integers consists of four rounds of parallel countsort PCOUNTSORT that works on distinct bits of the input keys starting from the least significant bits. Parallel countsort PCOUNTSORT calls the sequential count-sort algorithm in line 2, which counts the occurrences of each key, and uses this information in the parallel prefix (scan) step of line 3 to determine the final destination of each input key. Since as a by product of line 2, the keys are sorted based on the values of the 8 bits under consideration, the routing of line 5 can be performed in blocked fashion for each value of the 8 bits.

An optimization was incorporated into this algorithm; the optimized algorithm is RDXSORTOPT shown in Figure 8. RDXSORTOPT works similarly to RDXSORT except that the processors route their keys by considering distinct range values at a time. In line 4 processor 0 still sends first the keys with value 0; however processor 1 sends first the keys with value $256/NProcs$, where $NProcs$ is the number of available processors. *Pid* is the identification number of the processor executing the statement, and the term "wrappedto" means that when variable range reaches 255 it wraps around to zero and get incremented until the value to the right of the "wrappedto" is reached. At the end of the routing, an artificial delay in the form of a for loop of several microseconds is inserted in line 6.

Among the three test suites we have examined, RDXSORT is the one that tests to the limit the communication performance of the hardware platform and the capabilities of the communication library used. In lines 4-5, each processor issues 256 communication requests to as many as p processors. Timing results for $p = 1$ and speedup results for non-SMP and SMP configurations of the cluster are reported in Figure 5 and Figure 6. Note that $1M = 1024000$. Problem size is the number of integers that are evenly distributed over all the processors. We make sure that the same input set is sorted for increasing p. The input consists of uniformly at random generated unsigned

begin RDXSORT (*Input*,n,p)
1. **for** $0 \le i < 3$
2. PCOUNTSORT(*Input*,n,p,i);

begin PCOUNTSORT (*Input*,n,p,*rnd*)
1. *Consider the* $(rnd + 1) * 8 \ldots rnd * 8$ *least significant bits of Input in line 2*;
2. COUNTSORT (*Input*,n,*rnd*);
3. *Determine through parallel prefix the destination of each key in the final output sequence*;
4. **for** *range* $= 0$ **to** 255
5. *Using the information of line 3 route the keys with value range to their destination*;
6.
7. **return;**

Figure 7: Procedure RDXSORT.

32-bit integers. The sequential time for $n = 32M$ is indicated by an asterisk because the indicated value is an estimate of actual CPU time. This is because the setup of our cluster (memory limited to 128Mbytes) caused the sequential algorithm to swap and its effective running time was several minutes even though CPU time was substantially less. For the same reason, some entries in the tables are left empty; some limited swapping affected the time of RDXSORT and RDXSORTOPT in LAM MPI for the SMP configuration for $n = 32M$.

One-sided communication was at least 50% better than two-sided communication. Because of the performance of the `send`-based implementations we only report figures for `put`-based implementations in LAM MPI. The enhanced algorithm did not offer any improvements. It seems that software based message combining may be counter productive or messages become so large that they can not be dealt with efficiently by the communication library. Therefore figures for the enhanced algorithm are not reported either. RDXSORTOPT made marginal difference in the PUB-Library and *BSPlib* and its contributions were more noticeable in LAM MPI. In non-SMP configurations *BSPlib* was a clear winner over LAM MPI, with the PUB-Library suffering considerable degradation in performance for processor sizes $p > 2$. In SMP configurations *BSPlib* had a slight advantage in the performance of RDXSORT over LAM MPI. When RDXSORTOPT was tested the advantage was reversed. The PUB-Library was a distant third.

We note that maximum observed speedup for a non-SMP configuration was 6.45 for *BSPlib* and

begin RDXSORTOPT (*Input*,n,p)
1. **for** $0 \leq i < 3$
2. PCOUNTSORTO(*Input*,n,p,i);

begin PCOUNTSORTO (*Input*,n,p,*rnd*)
1. *Consider the* $(rnd + 1) * 8 \ldots rnd * 8$ *least significant bits of Input in line 2*;
2. COUNTSORT (*Input*,n,*rnd*);
3. *Determine through parallel prefix the destination of each key in the final output sequence*;
4. **for** *range* $= Pid * 256/NProcs$ **wrappedto** $Pid * 256/NProcs - 1$
5. *Using the information of line 3 route the keys with value range to their destination*;
6. Artificial Delay;
7. **return;**

Figure 8: Procedure RDXSORTOPT.

6.14 for LAM MPI ($p = 16$, $n = 32M$), and for SMP configurations it was 4.43 for *BSPlib* and 4.70 for LAM MPI ($p = 16$ and $n = 32M$). These figures are substantially smaller than general purpose parallel sorting algorithms such as those reported in [11, 10] which, if executed on the same platform give more than twice the indicated speedup figures. This is understandable however for parallel radix-sort. Although the sequential algorithm has performance $4nA$ (the 4 is the number of rounds of Count-Sort) and A varies between 1/2 microseconds per integer for $n = 4M$ to 1/4 for $n = 32M$, the parallel time includes a $4An/p$ term for parallel computation and a $4n/pg$ term for parallel communication, where g is the cost of a total-exchange of size n/p. The lowest value for g from Table 2 is 0.59 to 0.93 for *BSPlib* and LAM MPI and SMP and non-SMP configurations for $p = 16$. This is twice to four times as much as the A for $n = 32M$. Therefore the speedups we expect are between $p/3$ and $p/5$ and these are the figures we got. The speedup figures provided by the SMP configurations were on the average 30% lower than the figures of the non-SMP configurations. This indicates that parallel radix-sort is not as suitable as matrix multiplication for SMP PC clusters.

6 Conclusion

We made a case for remote memory access as the one and only way to program a parallel computer by proposing a robust programmatic framework that supports RMA in a library independent, simple, and portable way using the C programming language. We claimed that if one uses our approach the parallel code one writes will run transparently not only under MPI-2 RMA enabled libraries including LAM MPI and Critical Software's WMPI but also on bulk-synchronous parallel libraries such as *BSPlib* and PUB-Library thus making one's code library independent as well.

We supported our case by implementing using the proposed framework three sets of benchmark quality parallel programs: (a) a set of tests that assess the communication and synchronization performance of a parallel hardware platform that will thereafter be referred to as the As suite, (b) a set of parallel dense matrix multiplication programs, the Mult suite, and (c) a set of parallel integer radix-sort programs, the Rdx suite. We then examined the performance of these programs on a LINUX-based PC cluster under three different RMA enabled libraries: LAM MPI, *BSPlib* and PUB-Library. For comparative purposes variants of these programs were made to work using two-sided communication in LAM-MPI. In addition this experimental work serves another purpose: it offers a comparison of the communication performance of LAM MPI, *BSPlib* and the PUB-Library on a cluster of SMP PC workstations under a realistic set of benchmark programs.

7 Acknowledgements

The work of the authors was supported in part by the National Science Foundation under NSF MRI grant 9977508.

References

[1] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs. LogP. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 25-32, 1996.

[2] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping The Paderborn University BSP (PUB) Library. Parallel Computing, volume 29, no. 2 (February 2003), pages 187-207.

[3] H. T. Cormen, C. E. Leiserson, and L. R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[4] SHMEM Library Ready Reference, Cray Inc.

[5] Critical Software Inc. WMPI.

[6] D. E. Culler and R. Karp and D. Patterson and A. Sahay and K. E. Schauser and E. Santos and R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation.

Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

[7] F.Dehne A. Fabri and A.Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. Proc. ACM Symposium of Computational Geometry, 1993, pp. 298-307.

[8] R. Dohmen. Experiences with switching from SHMEM to MPI as communication library. Sixth European SGI/Cray MPP Workshop Manchester, UK, 7-8 September 2000.

[9] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jaing, and V. Sunderam. PVM: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Boston, 1994.

[10] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.

[12] A. V. Gerbessiotis. Architecture Independent Parallel Algorithm Design: Theory vs Practice. Future Generation Computer Systems, 18(2002), 573-593, Elsevier Science B.V.

[13] A. V. Gerbessiotis. `http://www.cs.njit.edu/~alexg/`, July 2003.

[14] P. B. Gibbons, Y. Matias and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. Proc. 9th ACM SPAA, 1997, pp. 72-83.

[15] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, `http://www.bsp-worldwide.org/`, April 1996.

[16] J.M.D. Hill, W. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R. Bisseling. *BSPlib*: The BSP Programming Library, Technical report PRG-TR-29-9, Oxford University Computing Laboratory, May 1997.

[17] LAM/MPI Parallel Computing. `http://www.lam-mpi.org`.

[18] G. Luecke, Wei Hu. Evaluating the Performance of MPI-2 One-Sided Routines on a Cray SV1. Technical Report, Iowa State University, December 2002.

[19] G. R. Luecke, S. Spanoyannis and M. Kraeva. The Performance and Scalability of SHMEM and MPI-2 One-Sided Routines on a SGI Origin 2000 and a Cray T3E-600. Performance Evaluation and Modeling of Computer Systems, December 2002.

[20] G. R. Luecke, S. Spanoyannis and J. Coyle. The Performance of MPI Derived Types on a SGI Origin 2000, a Cray T3E-900, a Myrinet Linux Cluster and an Ethernet Linux Cluster. Manuscript, 2001.

[21] R. Miller. A library for Bulk-Synchronous Parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, 1993.

[22] MPICH Parallel Computing. `//http://www-unix.mcs.anl.gov/mpi/mpich/`.

[23] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, Vol 6, pp 249-274, 1997.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* 33(8):103-111, August 1990.

		BSPlib		LAM MPI		PUB	
		nonSMP	SMP	nonSMP	SMP	nonSMP	SMP
Test 1	$p = 4$	330	370	420	350	770	880
Test 2	$p = 4$	320	360	430	350	780	840
Test 3	$p = 4$	1400	1920	3000	3800	1000	1330
Test 4	$p = 4$	588	600	2400	2000	800	850
Test 5	$p = 4$	998	1000	2700	2500	850	1000
Test 1	$p = 8$	600	900	990	740	1200	1300
Test 2	$p = 8$	700	700	1000	710	1200	1300
Test 3	$p = 8$	3000	4000	7300	6500	2000	2800
Test 4	$p = 8$	700	1000	5600	6900	1100	1400
Test 5	$p = 8$	1800	1900	6800	6000	1100	1700
Test 1	$p = 16$	1300	1400	1600	1200	1700	2100
Test 2	$p = 16$	900	1000	1600	1200	1500	1800
Test 3	$p = 16$	6000	8500	16000	15500	4500	6100
Test 4	$p = 16$	900	1500	12000	11500	1600	1800
Test 5	$p = 16$	1800	3600	15000	14500	1700	2000
Test 1	$p = 25$	-	1500	-	1600	-	2200
Test 2	$p = 25$	-	1200	-	1600	-	2100
Test 3	$p = 25$	-	13500	-	28000	-	10000
Test 4	$p = 25$	-	1400	-	20000	-	2200
Test 5	$p = 25$	-	2860	-	27000	-	32000
Test 1	$p = 32$	-	1400	-	2000	-	2500
Test 2	$p = 32$	-	1500	-	2100	-	2500
Test 3	$p = 32$	-	18000	-	42000	-	12000
Test 4	$p = 32$	-	1400	-	40000	-	2700
Test 5	$p = 32$	-	7200	-	40000	-	30000

Table 1: Synchronization and Latency performance of LAM MPI, *BSPlib*, and PUB. Time in μs.

		BSPlib		LAM MPI				PUB	
		nonSMP	SMP	nonSMP		SMP		nonSMP	SMP
p	h	put	put	put	send	put	send	put	put
4	16	75.	20.	140.	82.	130.	72.	66.	90.
4	64	21.	27.	38.	21.	35.	20.	16.	19.
4	256	5.	7.	10.	5.5	9.	4.8	4.4	5.
4	1K	1.4	2.	2.7	1.6	2.4	1.3	1.4	1.3
4	4K	0.60	0.9	1.9	0.58	0.82	0.56	0.62	0.69
4	16K	0.36	0.63	0.48	0.47	0.49	0.48	0.46	0.44
4	64K	0.35	0.56	0.41	0.38	0.49	0.47	0.35	0.41
8	16	90.	250.	360.	210.	330.	200.	121.	180.
8	64	22.	68.	94.	53.	86.	50.	30.	42.
8	256	6.	16.	25.	13.	22.	13.	7.6	11.
8	1K	1.5	4.	7.	3.8	6.3	3.5	2.1	2.7
8	4K	0.66	1.5	1.9	1.2	1.8	1.2	0.96	1.2
8	16K	0.47	0.82	0.76	0.60	0.93	0.90	0.54	0.71
8	64K	0.41	0.67	0.50	0.51	0.71	0.70	0.44	0.63
16	16	354.	530.	820.	500.	800.	480.	230.	340.
16	64	88.	132.	220.	145.	210.	130.	55.	85.
16	256	22.	34.	60.	38.	56.	35.	13.	23.
16	1K	5.6	8.	15.	10.	16.	9.	3.8	5.
16	4K	1.44	2.20	4.	2.8	4.4	2.8	65.	1.5
16	16K	0.65	1.00	1.3	1.0	1.4	1.18	17.2	2.8
16	64K	0.59	0.78	3.8	0.74	1.4	0.93	4.7	1.4
32	64	-	265.	-	-	580.	370.	-	200.
32	256	-	67.	-	-	155.	100.	-	40.
32	1K	-	17.	-	-	42.	30.	-	10.
32	4K	-	4.2	-	-	12.	8.	-	70.
32	16K	-	2.0	-	-	3.5	2.6	-	19.
32	64K	-	1.3	-	-	4.5	4.3	-	8.

Table 2: Communication performance (μs/32-bit word) of LAM MPI, *BSPlib*, and PUB.

Size	Time	PUB		*BSPlib*		LAM MPI		
n	$p = 1$	put	get	put	get	put	get	send
		Speedup for $p = 4$						
320	0.93	4.23	4.04	4.23	4.04	4.43	3.88	4.43
640	7.67	6.45	5.64	6.50	5.22	6.67	6.67	6.50
1280	69.93	9.39	8.05	9.14	6.03	9.36	9.35	9.18
1600	139.60	9.99	8.99	9.72	6.09	9.71	9.77	9.36
		Speedup for $p = 16$						
320	0.93	9.30	3.58	10.33	10.33	5.17	6.20	8.45
640	7.67	16.32	7.91	18.26	16.32	14.75	13.95	15.04
1280	69.93	28.31	19.32	29.38	23.55	26.49	27.32	27.10
1600	139.60	31.51	22.81	32.62	24.07	31.09	30.48	31.02

Table 3: Matrix Multiplication in a non-SMP configuration.

Size	Time	PUB		*BSPlib*		LAM MPI		
n	$p = 1$	put	get	put	get	put	get	send
		Speedup for $p = 4$ SMP						
320	0.93	3.58	3.44	4.65	3.32	5.17	4.43	4.89
640	7.67	7.10	5.48	6.85	4.54	7.10	7.10	6.97
1280	69.93	9.70	8.61	9.46	5.53	9.65	9.66	9.55
1600	139.60	10.33	9.20	10.00	5.81	10.00	9.97	9.74
		Speedup for $p = 16$ SMP						
320	0.93	6.64	5.17	7.75	7.15	5.47	5.17	6.64
640	7.67	12.78	7.59	14.47	12.57	11.45	11.62	11.98
1280	69.93	23.39	17.31	24.71	20.21	21.52	22.20	22.13
1600	139.60	26.79	20.32	27.98	22.05	25.34	25.95	26.00
		Speedup for $p = 25$ SMP						
320	0.93	7.75	3.44	8.45	9.30	4.43	2.16	7.15
640	7.67	15.65	7.17	19.18	16.32	13.00	13.00	14.47
1280	69.93	30.67	22.06	34.45	28.43	28.08	28.31	30.01
1600	139.60	36.17	26.09	39.10	32.62	33.16	32.92	34.73

Table 4: Matrix Multiplication in an SMP configuration.

		Non SMP Configuration							
Size	Time	RdxSort				RdxSortOpt			
	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
		Speedup Results for PUB-Library							
4M	2.13	0.62	0.75	0.94	1.20	0.60	0.74	0.95	1.10
8M	4.30	0.61	0.77	1.01	1.76	0.61	0.74	1.02	1.60
16M	16.32	1.14	1.47	2.11	2.98	1.15	1.46	2.07	2.81
32M	32.64*	-	1.47	2.21	3.21	-	1.50	2.16	3.22
		Speedup Results for *BSPlib*							
4M	2.13	0.73	1.14	1.75	2.96	0.73	1.15	1.72	2.88
8M	4.30	0.73	1.17	1.81	3.14	0.73	1.17	1.81	3.14
16M	16.32	1.39	2.24	3.45	6.35	1.39	2.23	3.48	6.33
32M	32.64*	-	2.24	3.51	6.26	-	2.25	3.52	6.45
		Speedup Results for LAM MPI							
4M	2.13	0.67	0.97	1.44	1.64	0.69	1.12	1.68	2.01
8M	4.30	0.59	1.04	1.60	2.15	0.59	1.12	1.89	2.35
16M	16.32	0.88	1.84	3.27	5.35	0.90	2.02	3.64	5.40
32M	32.64*	-	1.39	3.20	5.75	-	1.63	3.58	6.14

Table 5: RdxSort and RdxSortOpt speedup figures on an non-SMP PC cluster.

		SMP Configuration							
Size	Time	RdxSort				RdxSortOpt			
	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
		Speedup Results for PUB-Library							
4M	2.13	0.79	0.69	0.81	1.75	0.79	0.74	0.83	1.43
8M	4.30	0.81	0.77	0.88	1.31	0.82	0.77	0.88	1.28
16M	16.32	-	1.51	1.81	2.44	-	1.48	1.81	2.47
32M	32.64*	-	-	1.80	2.70	-	-	1.83	2.63
		Speedup Results for *BSPlib*							
4M	2.13	0.68	0.70	1.18	2.03	0.68	0.71	1.18	2.03
8M	4.30	0.69	0.71	1.24	2.15	0.69	0.73	1.24	2.15
16M	16.32	-	1.36	2.38	4.36	-	1.36	2.39	4.33
32M	32.64*	-	-	2.40	4.45	-	-	2.43	4.43
		Speedup Results for LAM MPI							
4M	2.13	0.75	0.83	1.09	1.30	0.77	0.79	1.30	1.72
8M	4.30	0.77	0.86	1.20	1.86	0.78	0.79	1.36	2.03
16M	16.32	-	0.91	2.37	3.93	-	1.05	2.49	4.53
32M	32.64*	-	-	1.53	4.17	-	-	1.73	4.70

Table 6: RdxSort and RdxSortOpt speedup figures on an SMP PC cluster.