Performance Evaluation and Modeling of Reduction Operations on the IBM RS/6000 SP Parallel Computer

Paraskevi Fragopoulou[1], and Ole H. Nielsen[1,2]

[1] UNI•C, Technical University of Denmark, Bldg. 304, DK-2800 Lyngby, Denmark.
[2] Center for Atomic-scale Materials Physics (CAMP), Physics Dept., Technical University of Denmark, Bldg. 307, DK-2800 Lyngby, Denmark.

Abstract. We discuss algorithms for global reduction (or combine) operations (e.g., global sums) for numbers of processors that need not be a power of 2, and implement these using standard message-passing techniques on distributed-memory parallel computers. We present performance results measured on an IBM RS/6000 SP parallel computer at UNI•C. Significant performance improvements are obtained by using a recursive doubling method with a vector splice/gather approach.

1 Introduction

Consider a parallel code where, initially, each of N processors holds a vector of data items; finally the result of the *reduction* operation over a commutative and associative operator is available on all of the N participating processors. Such combine operations are also synonymously denoted as *combine* operations. This type of operation has special importance because it constitutes a building block in several types of production codes. The performance issue is often important because global communication operations tend to impose a heavy load upon the inter-processor communication network.

We have implemented and compared the performance of three different global reduction algorithms on the IBM RS/6000 SP. The first algorithm is based on the simple master/slave approach where a processor, designated to be the master, collects the vectors from all the remaining (slave) processors, reduces them, and subsequently broadcasts the result to the slaves. The second algorithm is based on a variation of the recursive doubling approach, generalized to handle the situation were the number of processors is not a power of 2 [1]. The final algorithm is based on a two phase approach [1, 2]. During the first phase a vector splicing technique is employed; at the end of this phase each processor holds a partial result. The second phase completes the reduction with the gathering of the total result on each processor. Since latency is unimportant at the level provided by the IBM RS/6000 SP, the last algorithm achieves the best performance. Performance measurements were carried out on the UNI•C 40-node IBM RS/6000 SP using the IBM PVMe, IBM's enhanced Parallel Virtual Machine message passing library.

The implemented algorithms were used in a parallel Car-Parinello *ab-initio* molecular dynamics code [5]. This parallel code required the global sum of a large array and the broadcast of the summed array to all processors, which turned out to be the limiting step in the parallel speedup. Replacing the simple-minded sum-plus-broadcast with the reduction described in the present paper solved the problem, and an almost linear speedup curve was achieved for the number of processors of interest.

2 Reduction algorithms

Assume that there are n processors, labeled p_i, $0 \leq i < n$, each one holding a vector v_i, $0 \leq i < n$, respectively, of size m. Given an associative and commutative operator, denoted by $\oplus$, the goal of the global reduction is to reduce the n vectors element-wise, i.e., the operation $v_0 \oplus v_1 \oplus \ldots \oplus v_{n-1}$. The result, which is also a vector of size m, should be available on all the participating processors. An example of such operation would be a global sum of the vectors v_i, $0 \leq i < n$.

In this paper we describe three different global reduction algorithms. The performance of this type of algorithm is highly dependent on the amount of inter-processor communication required, so we describe the theoretical communication complexity of the algorithms with respect to the following model: We assume that each processor can communicate directly with any other processor as in a fully-connected network, which is the case for the IBM RS/6000 SP. In most message-passing systems, the time to transmit a message of length m between two processors is modeled as $t = t_s + mt_c$, where t_s is the start-up time (latency) and t_c is the per message unit (byte, integer, floating point etc.) transmission time. The communication performance of the algorithms is modeled with respect to the number of communication rounds they require, denoted by C_1, and the sum of the longest message sent or received by any processor during each round, denoted by C_2. Thus, an algorithm has an estimated communication complexity of $T = C_1 t_s + C_2 t_c$ [1]. It will be established in this paper that this communication model successfully describes the performance obtained on an IBM RS/6000 SP system.

2.1 Algorithm 1

In what follows, all subscript and index operations are modulo n and all logarithms are base 2.

The first global reduction algorithm is based on a simple master/slave approach. One of the n processors, designated to be the master, takes the responsibility to receive one-by-one the vectors located in each one of the other processors (slaves) and reduces them with the vector it holds. As soon as the master processor completes the reduction of the vectors, it broadcasts the result to the slaves. In our implementation the final vector is transmitted by the master to each one of the $n-1$ slaves in sequence, since the PVM library's broadcast subroutine is believed to be implemented in this way.

The number of communication rounds required for this algorithm is $n - 1$ for both the collection phase and the broadcasting phase. During each round, a message of length m is transmitted by each processor, and thus the theoretical communication complexity of this algorithm is $2(n-1)(t_s + mt_c)$. The maximum number of operations performed by a processor during the course of this algorithm is $(n-1)m$, since all the vector reductions are performed by the master.

2.2 Algorithm 2

The second algorithm was first presented by Bruck and Ho [1] and its theoretical communication complexity was analyzed under the *k-port* communication model, i.e., assuming that each processor can send and simultaneously receive distinct messages through k of its ports during a communication round. Here, we give a description of this algorithm under the more restricted *one-port* model, which is the appropriate one for the IBM RS/6000 SP.

This algorithm is a variation of the recursive doubling approach [1], extended to handle the situation where the number of processors n is not necessarily a power of 2. Assume that the binary representation of $n-1$ is $n-1 = n_{k-1} \ldots n_1 n_0$, where $k = \lceil \log n \rceil$. Here, $\lceil x \rceil$ denotes the smallest integer greater than or equal to x (the "ceiling" function). The basic idea of this algorithm is to treat $n-1$ according to Horner's rule as follows:

$$n - 1 = 2 * (2 * (\ldots (2 * (2 * n_{k-1} + n_{k-2}) + n_{k-3}) \ldots) + n_1) + n_0.$$

This algorithm completes in $k = \lceil \log n \rceil$ rounds. At the end of round r, $0 \leq r < k$, processor p_i holds the following three variables:

1. $c = 2 * (\ldots (2 * (2 * n_{k-1} + n_{k-2}) + n_{k-3}) \ldots) + n_{k-r-1}$ denotes the number of vectors that have already been reduced (initially $c = 0$).
2. v_i^0 holds the reduction result of the c vectors located in the c processors cyclically preceding p_i, i.e., $v_i^0 = v_{i-1} \oplus v_{i-2} \oplus \ldots \oplus v_{i-c}$
3. v_i^1 holds the reduction result of $c+1$ vectors, namely, vector v_i and the c vectors located in the c processors cyclically preceding p_i, i.e., $v_i^1 = v_i \oplus v_{i-1} \oplus v_{i-2} \oplus \ldots \oplus v_{i-c} = v_i \oplus v_i^0$

In the standard recursive doubling approach, which is valid only for n equal to a power of 2, the number of vectors which have been reduced doubles with each round. In this variation of the algorithm, the number of vectors c which have been reduced changes with each round r as $c = 2c + n_{k-r-1}$, which means that it either doubles (if $n_{k-r-1} = 0$), or it becomes double plus one (if $n_{k-r-1} = 1$).

The following pseudo-code describes the steps executed by processor p_i during round r, $0 \leq r < k$:

if (n_{k-r-1} .eq. 0) then
    Send v_i^0 to processor p_{i+c}
    Receive v_{i-c}^0 from processor p_{i-c} and store in v_{tmp}

else if (n_{k-r-1} .eq. 1) then
  Send v_i^1 to processor p_{i+c+1}
  Receive v_{i-c-1}^1 from processor p_{i-c-1} and store in v_{tmp}
endif
$v_i^0 = v_i^0 + v_{tmp}$
$v_i^1 = v_i^1 + v_{tmp}$
$c = 2 * c + n_{k-r}$

In the beginning of round r, $0 \le r < k$, each processor holds the reduction result of c vectors in v^0. During round r, each processor receives either vector v^1 (if $n_{k-r-1} = 1$), which is the reduction result of $c+1$ other vectors, or vector v^0 (if $n_{k-r-1} = 0$), which is the reduction result of c other vectors, from another appropriate processor. Thus, at the end of the round, each processor holds in v^0 the reduction result of $2 * c + n_{k-r-1}$ vectors. At the end of the algorithm, the final reduction result is stored in variable v_i^1 of each processor p_i. An inductive proof of the correctness of this algorithm can be found in the paper by Bruck and Ho [1] where this algorithm was first presented.

During each round, m data elements are received by each processor. Therefore, the communication performance of this algorithm is $\lceil \log n \rceil (t_s + mt_c)$. The total number of operations performed on each processor is $(2\lceil \log n \rceil - 1)m$. This factor of 2 in the number of operations per processor is the penalty paid for extending the algorithm to work when the number of processors is not necessarily a power of two, since two vectors v^0 and v^1 need to be updated at each round.

An example of this algorithm for $n = 6$ can be seen in Table 1. Since all index operations are modulo n, the final v^1 vector on all processors are identical, given that $\oplus$ is an associative operator.

	Before Round 1 ($c = 0$)		Round 1 (c=1,n_2=1) $p_i \xrightarrow{v_i^1} p_{i+1}$		Round 2 (c=2,n_1=0) $p_i \xrightarrow{v_i^0} p_{i+1}$		Round 3 (c=5,n_0=1) $p_i \xrightarrow{v_i^1} p_{i+3}$
	v^0	v^1	v^0	v^1	v^0	v^1	v^1
p_0	-	0	5	5:0	4:5	4:0	1:0
p_1	-	1	0	0:1	5:0	5:1	2:1
p_2	-	2	1	1:2	0:1	0:2	3:2
p_3	-	3	2	2:3	1:2	1:3	4:3
p_4	-	4	3	3:4	2:3	2:4	5:4
p_5	-	5	4	4:5	3:4	3:5	0:5

Table 1. An example of algorithm 2 for $n = 6$. The entry $i : j$ for v^0 or v^1 at a specific round means that the corresponding processor holds the reduction result of vectors $v_i, v_{i+1}, \ldots v_j$ in v^0 or v^1, respectively, at the end of this round.

2.3 Algorithm 3

The third algorithm was also first presented by Bruck and Ho in [1] for the *k-port* communication model. Here, we give a description of this algorithm under the *one-port* model.

This algorithm is based on a two phase approach. Each processor splices its vector into n blocks, each of size $\lceil \frac{m}{n} \rceil$ or $\lfloor \frac{m}{n} \rfloor$. We denote by $b_{i,j}$, $0 \leq i, j < n$, the j^{th} block of vector v_i placed in processor p_i. The first phase of the algorithm is a partial reduction that results in each processor p_i holding the final reduction result only for block b_i. During the second phase of the algorithm, each processor has to inform all other processors of the partial result that it holds. In other words, the second phase is nothing more than a concatenation algorithm.

The first phase requires $k = \lceil \log n \rceil$ communication rounds. For simplicity, we describe this phase of the algorithm for the case where n is a power of 2. If this is not the case, a single extra initial round is required [1]. We first show how processor p_i computes the final reduction result for block b_i at the end of this phase. The reduction of each individual block is performed using a simple recursive doubling technique. For example, for processor p_{n-1} to receive the result for block b_{n-1} the reduction proceeds as follows: in the first round processors $p_0, p_1, \ldots p_{\frac{n}{2}-1}$ send block b_{n-1} to processors $p_{\frac{n}{2}}, p_{\frac{n}{2}+1}, \ldots, p_{n-1}$, respectively. The receiving processors reduce the block they receive with their b_{n-1} block. In the second round only the last $\frac{n}{2}$ processors which received a block in the previous round participate in the reduction of block b_{n-1}. In general, in round r, $r = \log n, (\log n) - 1, \ldots$, the last 2^r processors participate in the reduction of block b_{n-1} and from these, the first half processors send b_{n-1} to the second half processors, respectively. The same process is simultaneously performed for each other block. The recursive doubling for block b_i with destination p_i, $0 \leq i < n$, is described with the following pseudo-code:

```
for r=(log n)-1 down to 0 do
    1. Processors p_{i-2^r}, ... p_{i-2^{r+1}-1} send block b_i to processors
       p_i, ... p_{i+2^r-1}, respectively.
    2. Each processor that receives a b_i block reduces it with the
       b_i block it currently holds.
end for
```

In Table 2 we can see the partial reduction phase for each block separately on $n = 4$ processors.

During round $r = (\log n)$-1, $\ldots, 0$, a total of 2^r processors participate in the reduction of each block. The partial reduction phase is performed simultaneously for all blocks. Thus, during round $r = (\log n)$-1, $\ldots, 0$, each processor participates in the reduction of 2^r blocks. More specifically, during round $r = (\log n)$-1, $\ldots, 0$, processor p_i sends to processor p_{i+2^r} the following 2^r blocks: $b_{i+2^r}, \ldots, b_{i+2^{r+1}-1}$. In Table 3 we can see the complete (for all blocks) partial reduction phase on $n = 4$ processors. A more general example for this phase of algorithm 3 on $n = 6$ processors can be seen in Table 4.

	b_0 reduction	b_1 reduction
Round 1	$p_2 \xrightarrow{b_{2,0}} p_0\ (b_{0,0} \oplus b_{2,0})$ $p_1 \xrightarrow{b_{1,0}} p_3\ (b_{3,0} \oplus b_{1,0})$	$p_3 \xrightarrow{b_{3,1}} p_1\ (b_{1,1} \oplus b_{3,1})$ $p_2 \xrightarrow{b_{2,1}} p_0\ (b_{0,1} \oplus b_{2,1})$
Round 2	$p_3 \overset{b_{3,0}\oplus b_{1,0}}{\dashrightarrow} p_0\ (b_{0,0} \oplus b_{2,0} \oplus b_{3,0} \oplus b_{1,0})$	$p_0 \overset{b_{0,1}\oplus b_{2,1}}{\dashrightarrow} p_1\ (b_{1,1} \oplus b_{3,1} \oplus b_{0,1} \oplus b_{2,1})$

	b_2 reduction	b_3 reduction
Round 1	$p_0 \xrightarrow{b_{0,2}} p_2\ (b_{2,2} \oplus b_{0,2})$ $p_3 \xrightarrow{b_{3,2}} p_1\ (b_{1,2} \oplus b_{3,2})$	$p_1 \xrightarrow{b_{1,3}} p_3\ (b_{3,3} \oplus b_{1,3})$ $p_0 \xrightarrow{b_{0,3}} p_2\ (b_{2,3} \oplus b_{0,3})$
Round 2	$p_1 \overset{b_{1,2}\oplus b_{3,2}}{\dashrightarrow} p_2\ (b_{2,2} \oplus b_{0,2} \oplus b_{1,2} \oplus b_{3,2})$	$p_2 \overset{b_{2,3}\oplus b_{0,3}}{\dashrightarrow} p_3\ (b_{3,3} \oplus b_{1,3} \oplus b_{2,3} \oplus b_{0,3})$

Table 2. The first phase of algorithm 3 for $n = 4$ and each block separately.

Round 1 (all blocks)	Round 2 (all blocks)
$p_0 \overset{b_{0,2},b_{0,3}}{\dashrightarrow} p_2\ (b_{2,2} \oplus b_{0,2},\ b_{2,3} \oplus b_{0,3})$	$p_0 \overset{b_{0,1}\oplus b_{2,1}}{\dashrightarrow} p_1\ (b_{1,1} \oplus b_{3,1} \oplus b_{0,1} \oplus b_{2,1})$
$p_1 \overset{b_{1,3},b_{1,0}}{\dashrightarrow} p_3\ (b_{3,3} \oplus b_{1,3},\ b_{3,0} \oplus b_{1,0})$	$p_1 \overset{b_{1,2}\oplus b_{3,2}}{\dashrightarrow} p_2\ (b_{2,2} \oplus b_{0,2} \oplus b_{1,2} \oplus b_{3,2})$
$p_2 \overset{b_{2,0},b_{2,1}}{\dashrightarrow} p_0\ (b_{0,0} \oplus b_{2,0},\ b_{0,1} \oplus b_{2,1})$	$p_2 \overset{b_{2,3}\oplus b_{0,3}}{\dashrightarrow} p_3\ (b_{3,3} \oplus b_{1,3} \oplus b_{2,3} \oplus b_{0,3})$
$p_3 \overset{b_{3,1},b_{3,2}}{\dashrightarrow} p_1\ (b_{1,1} \oplus b_{3,1},\ b_{1,2} \oplus b_{3,2})$	$p_3 \overset{b_{3,0}\oplus b_{1,0}}{\dashrightarrow} p_0\ (b_{0,0} \oplus b_{2,0} \oplus b_{3,0} \oplus b_{1,0})$

Table 3. The complete partial reduction phase of algorithm 3 for $n = 4$.

The concatenation phase also requires $k = \lceil \log n \rceil$ rounds. In the beginning of this phase, processor p_i holds a single final block b_i. During each one of the first $\lfloor \log n \rfloor$ rounds, each processor doubles the number of the final blocks it holds. In the beginning of round r, $0 \le r < \lfloor \log n \rfloor$, processor p_i holds the final blocks $b_i, \ldots, b_{i+2^r-1}$. During round r, processor p_i receives from processor p_{i+2^r} all the final blocks that this processor currently holds, namely, blocks $b_{i+2^r}, \ldots, b_{i+2^{r+1}-1}$. Thus, at the end of round r processor p_i holds the 2^{r+1} final blocks $b_i, \ldots, b_{i+2^{r+1}-1}$. After $\lfloor \log n \rfloor$ rounds, processor p_i holds the $2^{\lfloor \log n \rfloor}$ final blocks $b_i, \ldots, b_{i+2^{\lfloor \log n \rfloor}-1}$. At this point the algorithm completes if n is a power of 2. In any other case, an extra round is required to complete the concatenation of the final result. During this last round, processor p_i receives from processor $p_{i+2^{\lfloor \log n \rfloor}}$, the $n - 2^{\lfloor \log n \rfloor}$ blocks $b_{i+2^{\lfloor \log n \rfloor}}, \ldots, b_{i+n}$. An example of the second phase of this algorithm on $n = 6$ processors can be seen in Table 5.

This algorithm completes in $2\lceil \log n \rceil$ communication rounds. The number of message transmissions performed is $C_2 = 2\lceil \frac{m}{n} \rceil (n-1)$. Thus, the theoretical communication complexity of the algorithm is $2\lceil \log n \rceil t_s + 2\lceil \frac{m}{n} \rceil (n-1) t_c$. The number of operations performed by each processor is $\lceil \frac{m}{n} \rceil (n-1)$. Although this algorithm requires double the rounds of algorithm 2, it performs better with

	Round 1 $p_i \overset{b_{i+4},b_{i+5}}{\dashrightarrow} p_{i+2}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0	0	0,4	0,4	0	0
p_1	1	1	1	1,5	1,5	1
p_2	2	2	2	2	2,0	2,0
p_3	3,1	3	3	3	3	3,1
p_4	4,2	4,2	4	4	4	4
p_5	5	5,3	5,3	5	5	5

	Round 2 $p_i \overset{b_{i+2},b_{i+3}}{\dashrightarrow} p_{i+2}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0,4,2	0,4,2	0,4	0,4	0	0
p_1	1	1,5,3	1,5,3	1,5	1,5	1
p_2	2	2	2,0,4	2,0,4	2,0	2,0
p_3	3,1	3	3	3,1,5	3,1,5	3,1
p_4	4,2	4,2	4	4	4,2,0	4,2,0
p_5	5,3,1	5,3	5,3	5	5	5,3,1

	Round 3 $p_i \xrightarrow{b_{i+1}} p_{i+1}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0,4,2,5,3,1	0,4,2	0,4	0,4	0	0
p_1	1	1,5,3,0,4,2	1,5,3	1,5	1,5	1
p_2	2	2	2,0,4,1,5,3	2,0,4	2,0	2,0
p_3	3,1	3	3	3,1,5,2,0,4	3,1,5	3,1
p_4	4,2	4,2	4	4	4,2,0,3,1,5	4,2,0
p_5	5,3,1	5,3	5,3	5	5	5,3,1,4,2,0

Table 4. An example of the partial reduction phase of algorithm 3 on $n = 6$ processors. The entry $i_1, i_2, \ldots, i_j$ for a block at a specific round means that the processor holds in this block the reduction of the corresponding blocks of vectors $v_{i_1}, v_{i_2}, \ldots, v_{i_j}$ at the end of the round.

	Round 1 $p_i \xrightarrow{b_i} p_{i-1}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0:5	0:5				
p_1		0:5	0:5			
p_2			0:5	0:5		
p_3				0:5	0:5	
p_4					0:5	0:5
p_5	0:5					0:5

	Round 2 $p_i \overset{b_i,b_{i+1}}{\dashrightarrow} p_{i-2}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0:5	0:5	0:5	0:5		
p_1		0:5	0:5	0:5	0:5	
p_2			0:5	0:5	0:5	0:5
p_3	0:5			0:5	0:5	0:5
p_4	0:5	0:5			0:5	0:5
p_5	0:5	0:5	0:5			0:5

	Round 3 $p_i \overset{b_{i+2},b_{i+3}}{\dashrightarrow} p_{i+2}$					
	b_0	b_1	b_2	b_3	b_4	b_5
p_0	0:5	0:5	0:5	0:5	0:5	0:5
p_1	0:5	0:5	0:5	0:5	0:5	0:5
p_2	0:5	0:5	0:5	0:5	0:5	0:5
p_3	0:5	0:5	0:5	0:5	0:5	0:5
p_4	0:5	0:5	0:5	0:5	0:5	0:5
p_5	0:5	0:5	0:5	0:5	0:5	0:5

Table 5. An example of the concatenation phase of algorithm 3 on $n = 6$ processors.

respect to the number of message transmissions and the number of operations performed.

A further advantage of algorithm 3 is that all processors will hold identical results upon its completion, since each block b_i is uniquely determined by processor p_i prior to the concatenation phase. Algorithm 2 does not possess this property, in case numerical roundoff errors violate the assumption of associativity (e.g., addition on binary computers is not necessarily associative).

3 IBM RS/6000 SP message-passing performance

As a prelude to performance measurements of the reduction algorithms, we need to establish the "raw" point-to-point message-passing performance of our parallel computer system. In the following we discuss the IBM RS/6000 SP system and software, and discuss the bandwidth and latency of our IBM RS/6000 SP system.

The IBM RS/6000 SP parallel computer system [6] (formerly named SP2), contains from 2 up to 512 standard IBM RS/6000 workstation CPUs mounted in *frames* containing from 2 to 16 CPUs. All of the CPU nodes may be interconnected by an *SP Switch* [7] communications network. The *SP Switch* allows the user to view all nodes in the system as being equidistant, owing to very low hardware latencies in the switch. The latest version of the *SP Switch* achieves user-space bandwidths in excess of 100 MB/sec, and latencies down to 37 μsec.

The UNI•C IBM RS/6000 SP system [8] consists of 40 essentially identical "wide" nodes with 66-MHz POWER2 CPUs. Each CPU has at least 256 MB of RAM memory with a 2.1 GB/sec memory bandwidth. All nodes are interconnected by an IBM *High Performance Switch* (the predecessor of the latest *SP Switch*), with a peak bandwidth of 40 MB/sec for each node. The UNI•C IBM RS/6000 SP system currently runs the IBM AIX 4.1.4 operating system.

The message-passing library chosen in this work is *PVM* (Parallel Virtual Machine) version 3.3, in IBM's PVMe [9] version 2.1 implementation which is highly tuned for the *SP Switch*.

The measurements in this section record the communication time for a node to send a packet of data of a given length to another node. It must be noted that we employed the so-called "user-space" communication method. Alternatively, TCP/IP communication can be used with the *SP Switch*, yielding a lower performance due to the protocol overhead. Therefore, measurements using TCP/IP were not of interest in the present work.

The PVM library packs data into communication buffers before sending, and unpacks these buffers upon receiving. The sending node calls the `pvmfpack` and `pvmfsend` subroutines, or alternatively the combined pack-and-send `pvmfpsend` operation. The receiving node calls the reciprocal `pvmfrecv` and `pvmfunpack` subroutines, or alternatively the combined `pvmfprecv` operation.

The PVM data encodings `PvmDataDefault` and `PvmDataRaw` are identical in the PVMe implementation, whereas the `PvmDataInPlace` encoding is expected to achieve better performance because one less internal data copying operation takes place.

In Fig. 1 the measured PVMe node-to-node communication time is shown for short messages. We notice distinct steps in the timing for messages of about 200, 400, 640, and 880 bytes. The zero-length timing defines the *latency*, which is given in Table 6. We find reasonable latencies of less than 150 microseconds, with higher values for the combined `pvmfpsend` operation than for the separate `pvmfpack` and `pvmfsend` operations. This is surprising, but we have no information about the internal PVMe design that would allow us to understand this result.

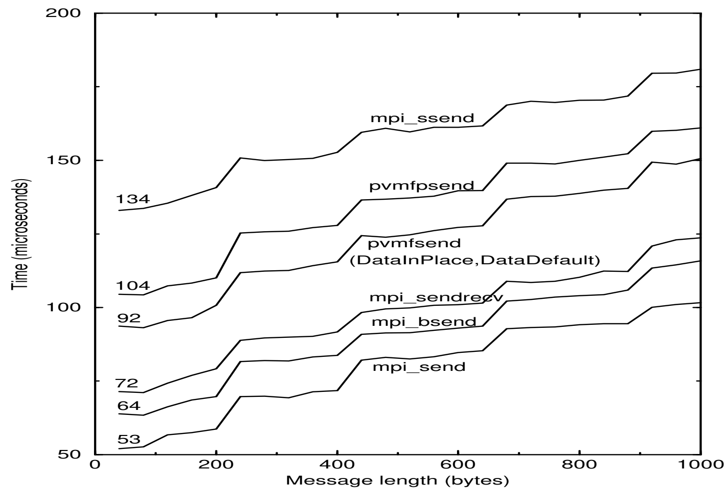


Fig. 1. Time in microseconds of a single node-to-node communication operation, as a function of the message length in bytes. The PVMe subroutines `pvmfsend` and `pvmfpsend` (and corresponding receive subroutines) are shown. The timings for the `pvmfsend/PvmDataDefault` and the `pvmfsend/PvmDataInPlace` were identical, thus a single line appears for the `pvmfsend` subroutine. The MPI subroutines are: `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` and `MPI_Sendrecv`. The timings are averages over 1000 iterations.

The bandwidth measurements are shown in Fig. 2 for longer messages. As expected, the communication efficiency (bandwidth) increases with message length for short messages, but at about 60 kBytes we see a slight step up in bandwidth, and then a constant decrease until about 200 kBytes, where the asymptotic bandwidths are achieved (see Table 6). We performed measurements of single messages of length up to 20 MBytes, but saw no deviation whatsoever from the asymptotic values in Fig. 2. The step at about 60 kBytes may be related to PVMe's internal use of 64 kByte receive-buffers, and it may be imagined that the use of several such buffers (i.e., for messages exceeding 64 kBytes) may increase the efficiency.

We note, as expected, that the `pvmfpsend` and `pvmfsend/PvmDataInPlace` achieve an asymptotic bandwidth higher than that of `pvmfsend/PvmDataDefault`.

Upon IBM's recent delivery of an MPI message-passing library [10] we were able to perform the same measurements as shown above for PVMe. It is of interest to compare these two competing implementations, since the choice of library may often be based upon performance.

The MPI specification [4] defines a number of send/receive operations. We have tested the basic `MPI_Send`, the buffered `MPI_Bsend`, the synchronous `MPI_-Ssend` and the combined send-and-receive `MPI_Sendrecv`. We did not time the non-blocking `MPI_Isend` operation, since it is not relevant for the present benchmark.

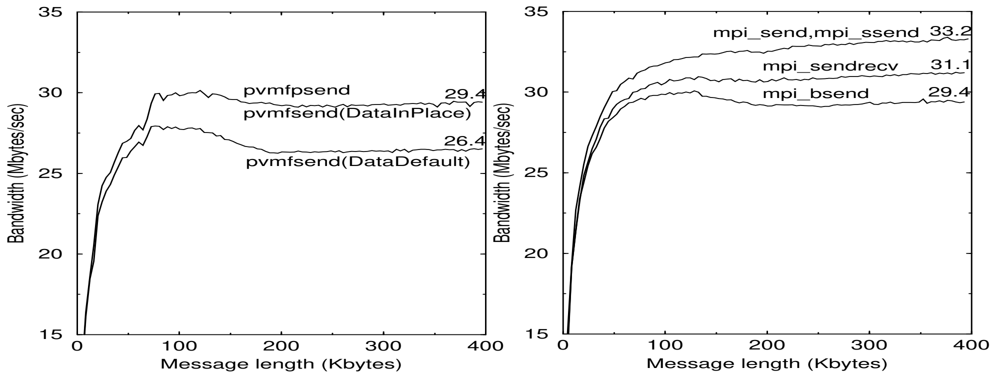


Fig. 2. Bandwidth of node-to-node communication in MBytes/sec as a function of the message length. The PVMe subroutines `pvmfsend` and `pvmfpsend` (and corresponding receive subroutines) are shown. The bandwidth of the `pvmfsend` subroutine using the `PvmDataInPlace` data encoding is identical to this of the `pvmfpsend` subroutine. The MPI subroutines are: `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` and `MPI_Sendrecv`. Notice that the measurements for the `mpi_send` and `mpi_ssend` subroutines are identical. The timings are averages over 500 iterations.

Fig. 1 shows the timings of the MPI subroutines. We note that the MPI latency for `mpi_send`, `mpi_bsend` and `mpi_sendrecv` is roughly half of that found for PVMe (see Table 6). Similarly to the PVMe case, we see steps in the timings at about 200, 400, 640, and 880 bytes. The `MPI_Ssend` is distinguished by a higher latency than that of the PVMe.

The MPI bandwidth is shown in Fig. 2. A bandwidth decrease from 60 to 200 kBytes is evident in `mpi_bsend` and `mpi_sendrecv`, as for PVMe. The asymptotic bandwidths are given in Table 6.

Send operation	Latency (μsec)	Bandwidth (MB/s)
`pvmfsend/PvmDataDefault`	92	26.4
`pvmfsend/PvmDataInPlace`	92	29.4
`pvmfpsend`	104	29.4
`MPI_Sendrecv`	72	31.1
`MPI_Bsend`	64	29.4
`MPI_Send`	53	33.2
`MPI_Ssend`	134	33.2

Table 6. Latency (in microseconds) and asymptotic bandwidth for very long messages (in MBytes/sec) for different implementations of *send* operations (with the reciprocal receive operations).

In conclusion, the IBM RS/6000 SP *High Performance Switch* (also denoted as "TB2") yields latencies of the order of 50 microseconds (MPI) to 130 microseconds (PVMe), and asymptotic bandwidths for long messages of 26-33 MBytes/sec. Some interesting structure is seen in the timings, but given the level of documentation provided by IBM it may not be possible to understand this structure. The latest generation of switch, the *SP Switch* (also denoted as "TB3") is now available at some sites, and is reported [7] to have a latency down to about 37 microseconds, and peak bandwidths of over 100 MBytes/sec, using the MPI message-passing library.

4 Reduction performance on IBM RS/6000 SP

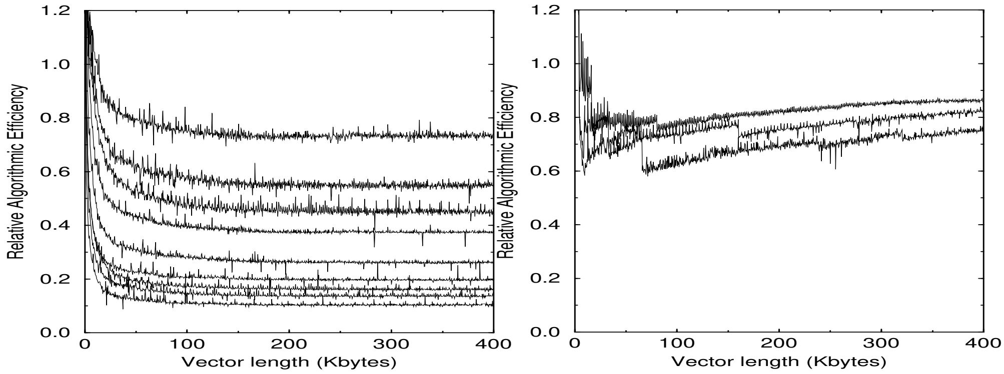
We implemented the three algorithms described in section 2 on an IBM RS/6000 SP parallel computer. We chose the IBM PVMe message-passing library (PVM version 3.3) which is highly tuned for the IBM RS/6000 SP Switch. Furthermore, we used the combined pack-and-send **`pvmfpsend`** and the reciprocal **`pvmfprecv`** subroutines for transmitting and receiving messages.

The relative performance of algorithms 1, 2 and 3 was experimentally established. We compared only the message passing time of the algorithms, since this is the dominating factor in their performance, and since we mainly want to establish the communication performance. The top three curves in Fig. 3(a) display the ratio of the message passing time required for algorithm 3 over the message passing time required for algorithm 2. The tests were performed on 4, 6, 8, 10, 12, and 16 processors and for vector sizes of up to 400 kBytes. The ratios on 6 and 8 processors were similar (with small deviations), and thus we plotted a single curve. The same was true for the tests on 10,12, and 16 processors. From Fig. 3(a), we notice that for small vector sizes of up to 10 kBytes, algorithm 2 performs slightly better than algorithm 3. However, as the vector size increases, algorithm 3 performs better than algorithm 2. On 4 processors, algorithm 3 is asymptotically 25% more efficient than algorithm 2, on 6 and 8 processors, it is 44% more efficient, and on 10, 12, and 16 processors algorithm 3 is up to 54% more efficient than algorithm 2. These performance measurements can be verified through the theoretical communication model established in section 2. The theoretical communication performance of these algorithms, if we neglect the start-up time, which is insignificant at the level provided on the IBM RS/6000 SP for long messages, is $\lceil \log n \rceil m t_c$ for algorithm 2, and $2\lceil \frac{m}{n} \rceil (n-1) t_c$ for algorithm 3. Thus, their ratio is:

$$\text{Ratio} \;=\; \frac{2\;\lceil \frac{m}{n} \rceil\;(n-1)\;t_c}{\lceil \log n \rceil\;\, m\;t_c} \;=\; \frac{2(n-1)}{n\lceil \log n \rceil}$$

This ratio for different numbers of processors can be seen in the following table:

n	4	6	8	10	12	16
Ratio	0.750	0.555	0.583	0.450	0.478	0.469



(a) (b)

Fig. 3. (a) Ratio of the message passing time required for algorithm 3 over the message passing time required for algorithms 1 and 2 for vector size of up to 400 kBytes. The top three curves are the ratio of algorithm 3 over algorithm 2 on $n = 4$, $n = 6, 8$ and $n = 10, 12, 16$ processors (from top to bottom). The lower six curves are the ratio of algorithm 3 over algorithm 1 on 4, 6, 8, 10, 12, and 16 processors (from top to bottom). (b) Ratio of the total time required for algorithm 3 over the total time required for the `MPI_Allreduce` subroutine for vector sizes of up to 400 kBytes. From top to bottom, the three lines correspond to timings on 4, 8, and 16 processor. The timings of all algorithms are averages over 500 iterations.

The ratios on different numbers of processors, derived from the theoretical model, agree asymptotically with the relative efficiency of algorithm 3 over algorithm 2, which we observed experimentally in Fig. 3(a). This relative efficiency would be even more dramatic if the total time required by the algorithm were compared, including the time spent for the reduction operations. Algorithm 3 performs only $\lceil \frac{m}{n} \rceil (n-1)$ reduction operations, compared to the $2(\lceil \log n \rceil - 1)m$ reduction operations performed by algorithm 2, which has to update two buffers with intermediate results at each round.

The bottom 6 curves in Fig 3(a) are the ratio of the message passing time required for algorithm 3 over the message passing time required for algorithm 1 on 4, 6, 8, 10, 12, and 16 processors, respectively from top to bottom, and for vector sizes of up to 400 kBytes. We notice as expected that algorithm 3 performs much better than the naive algorithm 1, and for 16 processors algorithm 3 is asymptotically a factor of 10 more efficient.

The performance of the most efficient of the algorithms described in section 2, i.e. algorithm 3, was further compared with the performance of the `MPI_Allreduce` subroutine from the IBM MPI library. This subroutine implements the reduction over a commutative and associative operator of a number of vectors, each one located on a different processor, and in the end the reduction result is available on all participating processors. The implementation details of this subroutine, as well as the algorithm used, are not documented

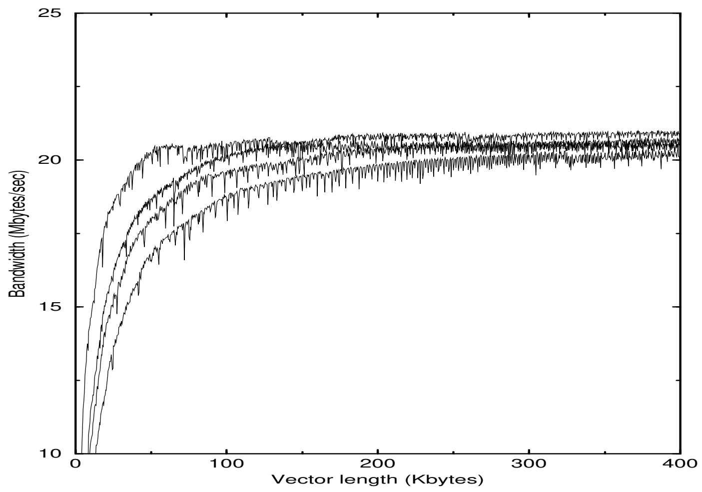


Fig. 4. The top curve is the bandwidth (in MBytes/sec) achieved by algorithm 2 on any number of processors for vector sizes of up to 400 kBytes. The three lower curves are the bandwidth achieved by algorithm 3 on 4, 6, and 12 processors, respectively, for the same vector sizes. The timings of all algorithms are averages over 500 iterations.

by IBM, and as a consequence we can only compare the total execution time of our implementation of algorithm 3, with the total execution time of this subroutine, rather than their message passing time. In Fig. 3(b) is displayed the ratio of the total execution time of algorithm 3, over the total execution time of the `MPI_Allreduce` subroutine. We notice that for vector sizes of more than 5 kBytes, algorithm 3 is more efficient than the `MPI_Allreduce` subroutine. On 16 processors and for vector sizes of 100-200 kBytes, algorithm 3 is about 40% more efficient than the MPI subroutine. Finally, on 16 processors algorithm 3 is asymptotically 25% more efficient than the `MPI_Allreduce` subroutine.

The same tests were performed for all three algorithms and the `MPI_Allreduce` subroutine, for vector sizes of up to 20 MBytes, but no deviations from their asymptotic behavior shown in Fig. 3 was observed.

In Fig. 4 we display the bandwidth in MBytes/sec achieved by algorithms 2 and 3 for vector sizes of up to 400 kBytes. The bandwidth is the ratio of the total size of the messages transmitted over the total time spent for these transmissions. The bandwidth for algorithm 2 was plotted only as one curve, because the measurements on different numbers of processors produced the same curve with almost no deviations. For algorithm 2 we notice that the asymptotic bandwidth, which is about 21 MBytes/sec was achieved for vector sizes of more than 50 kBytes.

For algorithm 3, we notice that a slightly different bandwidth curve was obtained for different numbers of processors: For $n = 4$, $n = 6, 8$, and $n = 10, 12, 16$ three different curves were produced. The reason is that the effect of the latency is more obvious for algorithm 3, since it requires $2\lceil\log n\rceil$ rounds, rather than the $\lceil\log n\rceil$ rounds required by algorithm 2. Furthermore, because of the message splicing technique employed in algorithm 3, messages are transmitted in smaller chunks, in contrast to algorithm 2 where only messages of the initial vector size m are transmitted. For this reason, for algorithm 3 the asymptotic bandwidth is achieved slightly slower as the number of processors increases. Soon the effect of the latency becomes negligible and eventually almost the same asymptotic bandwidth of 21 MBytes/sec is achieved on any number of processors. We tested both algorithms for vector sizes of up to 20 MBytes and we did not observe any deviation from the asymptotic bandwidth behavior. The behavior of algorithms 2 and 3 can be easily verified from the theoretical communication model established in section 2 as follows:

$$\text{(algorithm 2)} \qquad \text{Bandwidth} = \frac{\lceil\log n\rceil\ m}{\lceil\log n\rceil\ t_s + \lceil\log n\rceil\ m\ t_c} \tag{1}$$

$$\text{(algorithm 3)} \qquad \text{Bandwidth} = \frac{2\ \lceil\frac{m}{n}\rceil\ (n-1)}{2\ \lceil\log n\rceil\ t_s + 2\ \lceil\frac{m}{n}\rceil\ (n-1)\ t_c} \tag{2}$$

Equation (1) verifies that the number of processors does not affect the bandwidth performance of algorithm 2, since the $\lceil\log n\rceil$ factor cancels out, which is exactly what we observe experimentally. This is not the case for algorithm 3, where we see that the number of processors affect the performance, so that the asymptotic bandwidth is reached slightly slower as the number of processors increases. If we take $t_s = 140\mu$sec and $t_c = 1/21$ (bandwidth of 21 MBytes/sec), equations (1) and (2) accurately fit the corresponding curves in Fig. 4. This means that our implementations achieve a satisfactory bandwidth of 21 MBytes/sec, compared to the 29 MBytes/sec achieved through the round trip tests (see Fig. 2).

5 Conclusions

In this paper we have investigated in detail the algorithms by Bruck and Ho for the global combine operations on distributed-memory parallel computers, which scale as the logarithm of the number of processors. Such efficient algorithms are important for important classes of production codes on parallel computers.

We have measured the raw message-passing latency and bandwidth using IBM's PVMe and MPI libraries. The implemented combine algorithms achieve a satisfactory bandwidth of 21 MBytes/sec, compared to the 29 MBytes/sec that the round trip test showed for PVMe.

The performance measurements on the IBM RS/6000 SP show that algorithm 3 is 25-44% faster than algorithm 2, and that the naive algorithm 1 is far inferior to the other algorithms when the vector length exceeds a few kBytes. Algorithm 3 implemented using IBM's PVMe implementation of the PVM library is about 30% more efficient than the IBM MPI library's `MPI_Allreduce` subroutine on the

IBM RS/6000 SP. Thus it may be a good idea for IBM to incorporate algorithm 3 in future versions of their MPI library.

An important property of algorithm 3 is that it would lead to identical result on all processors even if the reduction operator were not associative, in contrast to algorithm 2. This can be of practical importance, for example, when the *addition* operator is used.

References

1. J. Bruck and C.-T. Ho, *Efficient Global Combine Operations in Multi-Port Message-Passing Systems*, Parallel Processing Letters vol. 3(4), pp. 335, 1993.
2. J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, *Efficient Algorithms for All-to-All Communications in Multi-Port Message Passing Systems*, manuscript.
3. H. Casanova, J. Dongarra, and W. Jiang, *The Performance of PVM on MPP Systems*, manuscript.
4. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, 1994.
5. B. Hammer and Ole H. Nielsen, *Parallel Ab-Initio Molecular Dynamics* in proceedings of the *Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering Science* (PARA'95), August 21-24, 1995, ed. J. Wasniewski, Springer Lecture Notes in Computer Science, vol. 1041, pp. 295.
6. IBM's RS/6000 SP documentation is available on WWW: URL:http://www.rs6000.ibm.com/software/sp_products/sp3.html.
7. The IBM RS/6000 SP Switch is documented on WWW: URL:http://www.rs6000.ibm.com/tech/sp_sw1/spswp1.book_1.html.
8. The UNI•C SP system documentation is available on WWW: URL:http://www.sp2.uni-c.dk.
9. IBM's PVMe product is described on WWW: URL:http://www.rs6000.ibm.com/software/sp_products/pvme.html.
10. IBM's Parallel Environment (PE) product, which includes the MPI library, is described on WWW: URL:http://www.rs6000.ibm.com/software/sp_products/pe.html.
11. Oak Ridge performance measurements available on WWW: URL:http://www.epm.ornl.gov/pvm/perf-graph.html.

This article was processed using the LaTeX macro package with LLNCS style