# Optimizing User-Level Communication Patterns on the Fujitsu AP3000

Jeremy Dawson and Peter Strazdins,
Department of Computer Science,
Australian National University
Acton ACT 0200 Australia
{jeremy,peter}@cs.anu.edu.au

## Abstract

*In this paper, we present techniques and algorithms to improve the performance of various communication patterns on message-passing platforms where, for reasons of safety, user-level communications must be buffered in (special) memory on both the send and the receive. These algorithms can not only minimize message copying but overlap the copying to/from the special memory with the actual transfer, enabling full bandwidth to be achieved. These patterns include tree broadcast and reductions, (ring-based) multiple broadcasts and reductions, pipelined broadcast and buffered point-to-point sends. In each case, the messages may have a simple stride. All of these patterns are used in dense linear algebra applications, although they are also used in many other contexts.*

*These algorithms are implemented and their performance evaluated on the Fujitsu AP3000, a message passing multicomputer having many characteristics of the cluster model. Some aspects, such as the performance characteristics of the special memory, are specific to the AP3000; however, the algorithms still apply to any platform using a similar mode of user level communications. Worthwhile performance increases are obtained, especially for patterns involving moderate-large number of processors.*

## 1 Introduction

The Fujitsu AP3000 [3] is a distributed memory multicomputer, comprised of RISC scalar processors (UltraSPARC) with a deep memory hierarchy (having a 16KB top-level data cache and a 1MB 2nd-level cache, both direct-mapped, and a 64-entry TLB). Each node supports a standard Solaris operating system. It has communication networks with characteristics shared by most other state-of-the-art distributed memory computers, that is, high communication costs relative to floating point speed, and row or column broadcasts having to be simulated by point-to-point messages. The AP3000 also has many properties of the cluster computing model; this extra flexibility contributes to its communication costs.

Under the APruntime V2 runtime system [5], the AP3000 effectively behaves as a cluster: many parallel and serial jobs can run simultaneously, and on overlapping subsets of processors. Furthermore, information may be transferred using a system level 'transport' (TCPIP) or a user level transport; for programs using message passing libraries (eg. MPI [2]), the transport can be selected transparently [5]. Thus high availability of the system is achieved; however, the current lack of global scheduling means that overlapping parallel jobs cause overall performance degradation.

MPI is implemented in APruntime in terms of a lower-level library called VPPLib, which is in turn implemented, with very low additional overhead, in terms of the LWSLT low-level library [5]. This library has yielded latencies of $26.3\mu s$ and bandwidths (for large messages) of 85 MB/s on a U170-based AP3000 [5].

The main original contributions of this paper are to describe techniques and algorithms for optimizing communication patterns on platforms where messages must be buffered upon both send and receipt, and the actual transfer of the messages occurs in chunks (which can be overlapped with the buffering). It evaluates these algorithms on the AP3000.

This paper is organized as follows. Section 2 gives details of user-level communication on the AP3000, and Section 3 describes the communication patterns to be optimized. Their implementation is described in Section 4, with their performance on the AP3000 is given in Section 5 and conclusions being given in Section 6.

## 2 User Level Transfers on the AP3000

On the AP3000, a message transfer to and from user memory must occur in three stages, as the message must

be copied to and from 'special memory', that is, (message buffering) memory that can be accessed by each node's Message Controller (MSC). The MSC is connected to the node by an SBUS. This memory includes 12 MB of SDRAM memory directly attached to the MSC, and 12 MB of 'KMEM', a reserved part of the node's memory, allocated for use by the MSC [5]. Figure 1 summarizes message transfer paths [5]. Such a convention helps ensure safety and security in a multi-user environment, as a message transfer to and from user memory requires the cooperation of both the source and destination nodes. This convention may similarly be expected on other cluster architectures using user-level communications.

However, this mode of message transfer can potentially degrade communication performance seriously, with for example 8KB messages (a typical size for a vector) being transferred at a rate of only 30 MB/s, whereas the AP3000's hardware inter-node bandwidth is 200 MB/s [3].

A method to improve performance for larger messages is the *protocol method* [5], which involves breaking the messages into large (eg. 32 KB) chunks and pipelining the transfers of chunks over the three stages. This is possible because the SBUS can perform read and write transfers simultaneously without loss of bandwidth. In this way, 1 MB or larger messages can be transferred at the rate of 85 MB/s. This method is already incorporated into VPPLib and MPI message send and receive calls on the AP3000; the details of its effective implementation are however quite subtle.

# 3 Improving Communication Pattern Performance

The idea of the protocol method can be extended to *communication patterns*, where the pipelining can be performed over the several messages involving any one node in the pattern, and/or the copies to special memory can be amortized over different messages.

For example, in blocked parallel matrix factorizations using the storage blocking method [1, 7], the horizontal panel (contained in a processor column) (sub-matrix) is communicated using a pipelined row broadcast [1], and the vertical panel (contained in a processor row) is communicated via a tree column broadcast. Using the algorithmic blocking method, [7] where the panels are distributed across all processors (see Figure 2(a)), these operations tend to occur on long vectors instead, and/or are replaced with a panel multi-broadcast or spread operation [7]. QR and LDLT factorizations may also use row or column reductions [7, 6]. Point-to-point messages are use for explicit transposition (QR and LDLT) and row/column interchanges (LU, LDLT). In many dense linear algebra libraries such as ScaLAPACK, a *buffered send* is required to fulfill the required communication semantics, that is sender must return from the send call regardless of the state of the receiver; this is generally achieved by buffering the message on the sender.

In the dense linear algebra context, the data to be communicated are in the form of $M \times N$ (row-major) matrices, with a row stride of $\texttt{Lda} \geq N$. If $\texttt{Lda} > N$ and $M > 1$, the message is not *contiguous*; it is important that extra memory copying be avoided where possible for *non-contiguous* messages.

The main cost in parallel matrix factorization for moderate-large matrices on the AP3000 is in communication volume overheads [6]. We are primarily interested in improving the performance of such applications; however, these patterns occur in many other diverse applications, which could also benefit from this work.

This idea's main potential in enhancing performance is for moderate-sized messages, that is in messages too small for the (full) benefit of the protocol method. For smaller messages, performance might be improved by looking at alternate memory copy routines than Solaris `memcpy()` (which we have found to be very efficient on moderate-large data sizes to and from special memory).

## 3.1 Communication Patterns to be Optimized

A binary tree broadcast is a well-known pattern; over $P$ cells, the root node must send $\lg_2 P$ messages; hence the cost of the broadcast relative to a single message is $\lg_2 P$.

The binary tree reduce can be thought of as the reverse of this operation, where each node receiving a message adds to that message its corresponding contribution, before sending that message up to its parent node in the tree. Thus, the root node receives the sumnation of all cell's contributions. Like the tree broadcast, its relative cost is $\lg_2 P$; however, as the memory operation is a (vector) add, rather than a vector copy, it is generally slower than the broadcast.

The pipelined or *increasing ring* broadcast [1] can be used as a faster alternative to the tree broadcast in computations. A pipelined row broadcast can be used in computation having at least $Q$ stages, where the broadcast source changes in a round robin fashion in each stage, and no communications need flow in the opposite direction. As illustrated in Figure 2(b), a 'bubble' of size $Q$ is introduced; hence the cost of $Q$ pipelined broadcast, relative to $Q$ normal message sends is 2, and is thus faster than a tree broadcast if $Q > 2$.

For multi-broadcast or *spread* operations (see [4] and the references within), we must consider matrices block-cyclically distributed [1, 7] across a $P \times Q$ processor grid. Here, all processors have an (often roughly equal) portion of the data to be replicated. Figure 2(a) shows a row 'spread' of $A$ (having 8 block columns) across a $3 \times 3$ grid, storing the result in the column-replicated matrix `AsM`. The pattern can be achieved by $Q$ individual broadcasts (including the

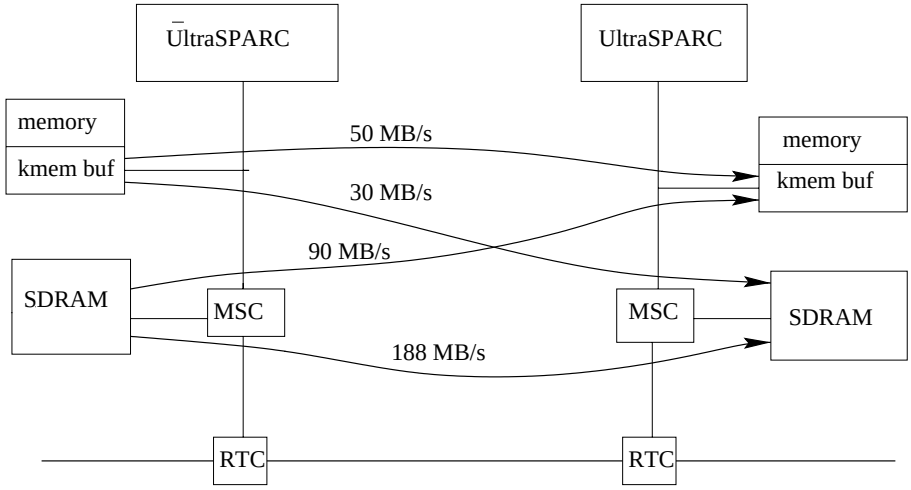



**Figure 1. AP3000 inter-node communication configuration**

pipelined broadcast, which is a potentially competitive way of implementing this function). However, it can also be achieved efficiently using a series of $Q-1$ ring-shift operations [4]. Here, where each cell initially sends its own contribution rightwards; it then receives $Q-1$ messages from the left, copying the message into the corresponding part of `AsM` before passing it rightwards. Relative to $Q-1$ normal message sends, the cost of the ring shift is between 1 and 2, depending on whether the cells in a target platform can be simultaneously sending and receiving messages at full speed.

The ring multi-reduce [4] can be implemented very similarly to the ring multi-broadcast; each cell initially sends rightwards its contribution in `AsM` belonging to the cell on its left; as that message is passed around, each intermediate cells adds the corresponding contribution. Thus after $Q-1$ messages, the message arriving in each cell is added to that cell's contribution in `AsM`, which is then stored in that cell's portion of $A$.

Note that all of these patterns, except for the *increasing ring* broadcast, are supported by MPI [2].

## 4 Implementation

This section describes how the above patterns can be implemented efficiently. The first consideration is however the performance characteristics of the platform's (in this case the AP3000's) special memory, which has considerable impact on design decisions. These patterns can be divided into two classes: 'synchronous' patterns, where it can be assumed that all cells participating in the pattern do so at the same time, and 'non-synchronous' patterns. Each has slightly different design considerations.

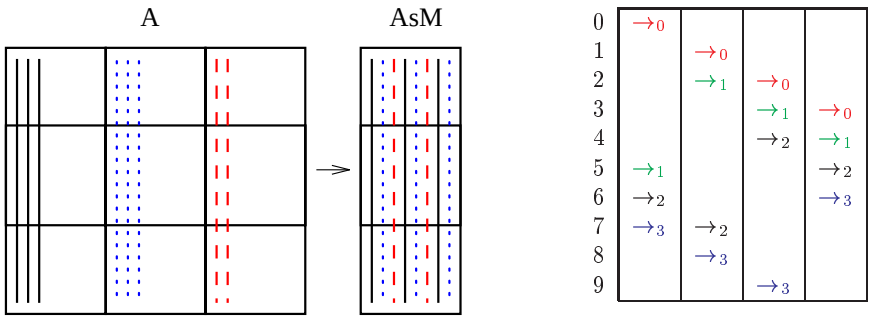
### 4.1 Memory Performance of the AP3000 Special Memory

As depicted in Figure 1, direct user level transfers can only occur to and from the SDRAM or KMEM special memories, with the optimal path beings from SDRAM to KMEM or SDRAM for large messages. Thus, total message transfer must include the efficient copying of data to/from SDRAM or KMEM; to support the protocol method, this copying occurs in chunks of size $c = 32$KB. While both SDRAM and KMEM are cacheable, in the message transfer context, this is irrelevant as either the chunks will not be in cache, or if they are, must be flushed on send or invalidated upon receive. Thus, finding an efficient memory copy and add routines for the case when the chunk is uncached is important.

We discovered that the Solaris memory copy `memcpy()` routine performed consistently in this situation, achieving 100–180 MB/s for large contiguous messages between SDRAM, KMEM and normal memory. It achieved such high consistent performance by using the UltraSPARC block load and store instructions which bypass cache.

The AP3000's SBUS has high latency; this has a strong impact on any scheme to optimize strided memory accesses to/from SDRAM, as between rows there is potential to 'lose' the SBUS, which could result this latency being incurred on each row. Even for contiguous messages of size $< c$, KMEM was used to send messages, being faster here than SDRAM.

For the DBLAS library, a general matrix copy routine `MatCopy()` was developed for the UltraSPARC [8]. This is highly software pipelined to hide top-level cache misses; thus, if the data was in level 2 cache, it would out-perform




(a) Row 'spread' on a $3 \times 3$ grid

(b) (multiple) pipelined b/c across a $1 \times 4$ grid

**Figure 2. Multi-broadcast communication patterns**

`memcpy()` by a factor of 2 or more. However, for uncached data, it ran at 120 MB/s for normal memory, and up to a factor of 3 slower if SDRAM was used. A highly pipelined general matrix add routine `MatAdd()` was also developed; it had similar performance characteristics.

For contiguous messages or messages with long rows (eg. $\geq$8KB), the optimal method to copy a matrix to/from a chunk of special memory was to use `memcpy()` to copy each row individually. In however the case of a small (eg. $<$8KB) message send, `MatCopy()` would be used as it is faster, as the source is in normal memory and likely to be in cache.

For messages with short rows, a moderately software-pipelined memory copy routine was found to be optimal. Due to the high S-BUS latencies, this only achieved good performance if the source or destination was KMEM.

For the memory add, no corresponding vendor-supplied routine exists; however, a minimally pipelined routine was found to significantly faster than `MatAdd()` in the same circumstances `memcpy()` out-performed `MatCopy()`. It should be noted that in either case, performance was degraded by a factor of 2 if the destination matrix was from SDRAM, so that KMEM is the preferred type of memory for this operation, even if used for a subsequent send.

### 4.2 'Synchronous' Patterns: (Multi-) Broadcast and Reduce

The AP3000 SDRAM and KMEM memory is a precious resource that must be shared between all parallel tasks currently executing on an AP3000 node. As it can be assumed that all cells participating in these patterns do so at the same time, it is possible and desirable to use a fixed number of special memory chunks in these routines, regardless of the total message size. Thus, the algorithms for these routines consist of an outer loop iterating over the number of chunks, and an inner loop iterating over the number of messages (that the current cell is involved in).

A second principle was to initiate sends (denoted as an '`ISend()`' call in MPI terminology [2]) as early as the chunk had the correct value, and to defer the corresponding call to wait for completion ('`WaitSend()`') only when the chunk had to be reused (or deallocated). Similarly, the receive would be initiated ('`IRecv()`') on a (KMEM) chunk as soon as it was safe to start overwriting the chunk. By thus putting the minimal constraint on when the message transfer of a chunk actually occurred maximized the chance that the transfer could be overlapped by the copy to/from other chunks.

The tree broadcast over $P$ cells was thus implemented requiring $n_c \leq 4$ (send) chunks to be allocated; the root node would copy each chunk into SDRAM and post their `ISend()` as soon as each copy was complete. Other nodes posted `IRecv()` calls $n_c - 1$ chunks in advance, and copied the chunks into the destination memory only after sending that (KMEM) chunk onto its child nodes. This not only reduces the amount of memory copies by a factor of $\lg_2 P/2$ but also maximizes the chance of the copies being overlapped with the sending of the same chunk and the receipt of new chunks.

The tree reduce required $n_c \leq 2$ chunks for receiving and $n_c$ chunks for sending. All non-root cells first copy their contribution to the respective send chunk. There are less opportunities for overlapping here as intermediate nodes must add each (KMEM) chunk it receives to its send chunk; only when that completes can it be sent to its parent cell. The root cell adds each received chunk directly into the destination matrix.

The ring multi-broadcast and reduce are implemented using a single routine. It should be noted from Figure 2(a) that strided messaging is needed whenever there are $> P$ block columns. In either case, $n_c \leq 4$ send and $n_c$ receive

chunks were required. Here, there are $P-1$ messages involved with each chunk; thus this pattern has more scope for reducing memory copies and overlapping than does the tree broadcast or reduce, provided $P > 2$. For the first message in the pattern, each cell would copy the next chunk as it sent the current chunk to the cell on the left. In the case of the ring multi-broadcast, when a cell received a chunk, it would immediately post an `ISend()` and begin copying that chunk into its destination matrix; for the multi-reduce, as for the reduce, the `ISend()` had to be deferred till the corresponding add into its (KMEM) send chunk was completed (except for the $P-1$th message, where the received chunk would be copied directly into the destination matrix).

### 4.3 'Non-Synchronous' Patterns: Buffered Point-to-Point Send and Pipelined Broadcast

Consider the context of an application calling ScaLAPACK with MPI-BLACS (BLACS implemented using MPI [1]) on the AP3000. The layered design of the BLACS, MPI and VPPLib software has the result that data to be sent in a message is copied several times, for different reasons:

1. to or from special memory, which can be accessed by the MSC
2. (for a send call) to a local buffer, so that the send call is locally blocking (guaranteed to return promptly, leaving the sending user's message space free for reuse, whether or not the receiver has issued a receive call)
3. to pack/unpack non-contiguous data (eg. matrices)

The BLACS codes use a single copying step for purposes 2 and 3 above, but the copying to and from the AP3000 special memory is performed at a lower level, where the software is specific to the AP3000 hardware. The routines described here aimed to do a single copy for all three purposes. This required that enough special memory be used to buffer the entire message. Furthermore, for a matrix with short rows, copies to/from special memory could now occur in relatively small contiguous portions (eg. 32 double words).

We found that for a block-size (eg the number of rows of a column-major matrix being sent) of less than 2400 bytes, it was better to use KMEM for sending messages as well as for receiving them; for larger blocksizes SDRAM was used for sending, and KMEM for receiving. For block-sizes less than 400 bytes, the moderately pipelined routine was used for copying data to and from special memory, whereas `memcpy()` was used for larger blocksizes.

For the ring broadcast, each node except the first and last receives the message and sends it on to the next node. For these nodes, the one routine

- receives the message into special memory (SDRAM was found to be fastest)
- sends it on to the next node
- copies it into user memory

In this way the amount of copying of the message is further halved, since these nodes each do a receive and a send, but only one copy between user and special memory.

## 5 Performance

This section describes the performance of these communication patterns on an U170 based AP3000 of 8 nodes. The 'synchronous patterns' could be compared using a multi-spread or reduce benchmark; a ping-pong benchmark was used for the point-to-point send.

The following sections gives results for the communication bandwidth, as bandwidth improvement is objective of our techniques. It was found that these techniques gave only a marginal improvement in latency, primarily due to having less software overheads, eg. procedure calls.

### 5.1 (Multi) Broadcast and Reduce

As previously mentioned, the multi-broadcast (Figure 2(a)) can be implemented in terms of a series of (tree) broadcasts. Thus, it forms a suitable benchmark for a tree broadcasts, as root-to-leaf latencies dominate the overall time, as the source of the broadcasts changes in a round-robin fashion at each internal stage. Similar comments apply for the reduce.

Table 1 gives the performance in MB/s of the various 'synchronous patterns' of Section 4.2. The multi- broadcast (or reduce) operation was repeated sufficiently to get an accurate timing, with the nodes being synchronized before a timing was taken. Note that this benchmark includes a memory copy step for each cell to initialize its portion of the workspace (see Figure 2(a)).

Columns 'A' use normal VPPLib `I_Send/Recv()` calls to perform the corresponding patterns. Buffers of normal memory are used to reduce packing/unpacking overheads for the case of multiple messages [8]; thus these routines are already optimized in some sense and present a performance target for our optimized routines (columns 'B'). $1000 \times Q$ matrices do not require strided communication; thus for $P = 2$, there is little scope for the optimized routines for performance improvement. The $4000 \times 32$ matrices represent messages large enough for full overlapping to be achieved via the *protocol method*; similarly, for $P = 2$, the optimized routines only have the advantage of reducing some of the buffer packing overheads. Thus, for the middle

| | $Q=2$ | | | | $Q=4$ | | | | $Q=8$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | tree | | ring | | tree | | ring | | tree | | ring | |
| $M \times N$ | A | B | A | B | A | B | A | B | A | B | A | B |
| $1000 \times Q$ | 26 | 23 | 36 | 36 | 11 | 14 | 19 | 20 | 6 | 10 | 15 | 15 |
| $1000 \times 2Q$ | 24 | 28 | 33 | 39 | 10 | 15 | 19 | 23 | 10 | 13 | 15 | 21 |
| $1000 \times 32$ | 40 | 34 | 37 | 45 | 15 | 20 | 24 | 29 | 11 | 14 | 15 | 23 |
| $4000 \times 32$ | 28 | 32 | 44 | 41 | 19 | 22 | 25 | 28 | 16 | 15 | 21 | 24 |
| $1000 \times Q$ | 31 | 21 | 33 | 32 | 12 | 11 | 17 | 18 | 7 | 7 | 14 | 14 |
| $1000 \times 2Q$ | 20 | 23 | 29 | 33 | 11 | 12 | 17 | 19 | 8 | 8 | 15 | 17 |
| $1000 \times 32$ | 27 | 33 | 29 | 31 | 15 | 18 | 20 | 20 | 9 | 9 | 15 | 19 |
| $4000 \times 32$ | 24 | 32 | 31 | 29 | 17 | 14 | 24 | 20 | 11 | 10 | 21 | 18 |

**Table 1. Row Multi-broadcast (top rows) and reduce (bottom rows) benchmark in MB/s for $M \times N$ double precision matrices on a $1 \times Q$ AP3000**

two sizes and for $P > 2$, there is most scope for improvement of performance by the optimized routines.

We see from Table 1 that the optimized communication patterns generally do achieve better performance, sometimes as much as by 50%, but that overall it is not as great as might have been expected. This is partly because of the nature of AP3000's special memory, whose copy performance is best on large contiguous messages, and partly because the full overlapping of the protocol method is harder to achieve for complex communication patterns than for individual point-to-point transfers. This is particularly the case for the reduce operations, where adding to special memory is particularly problematic, and the order of its sub-operations are more tightly constrained than for the broadcast.

A comparison of our pipelined broadcast with that of MPI BLACS indicated only marginal improvement for small messages ($<$ 16KB), but showed a clear advanatage for large messages. This was most marked for $Q = 2$ (100% faster), but the difference decreased with $Q$, being 27% faster at $Q = 8$.

### 5.2 Blocking Point-to-Point Send

As expected, our routines performed better than the corresponding BLACS routines. Both were tried using blocksizes of 256 bytes and 4000 bytes, and for contiguous messages. The graph in Fig. 3 shows the speeds for various message sizes. We also compared the performance of `MPI_ISend()` for contiguous data, which is the target performance of our routine. Its performance was generally comparable to that of our routine, although its graph was irregular: for some message sizes it performed significantly worse.

Preliminary results for the pipelined broadcast indicate similar speeds were attained.

## 6 Conclusions

On message-passing platforms where the safety of user-level transfers requires buffering the message (in 'special' memory) at both the send and receive, there is twofold scope for the improvement of communication patterns: firstly, to reduce extra packing overheads in the case of strided messages, and secondly, to overlap the buffering with the transmission of the different messages involved in the pattern. We have shown how algorithms for such patterns can be designed in order to achieve such improvements.

On the AP3000, peculiarities in the performance of the 'special' memory has made difficult achieving the full potential of our optimized algorithms for these communication patterns. However, substantial performance improvements were observed for the tree broadcast, the ring multi-broadcast, the pipelined broadcasts and the buffered point-to-point send. In general, the simpler the pattern the better the improvement, as it is easier here to get full overlapping of the message buffering and transmission in this situation.

We expect however that the improvements are great enough for a substantial speedup for applications such as dense linear algebra, which intensively use these patterns. Future work includes integrating our routines into and AP3000 implementation of the BLACS, so that ScaLAPACK applications may use these routines. Another avenue would be to investigate other contemporary message-passing platforms for the suitability of these techniques.

## References

[1] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D.W. Walker, and R.C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR and

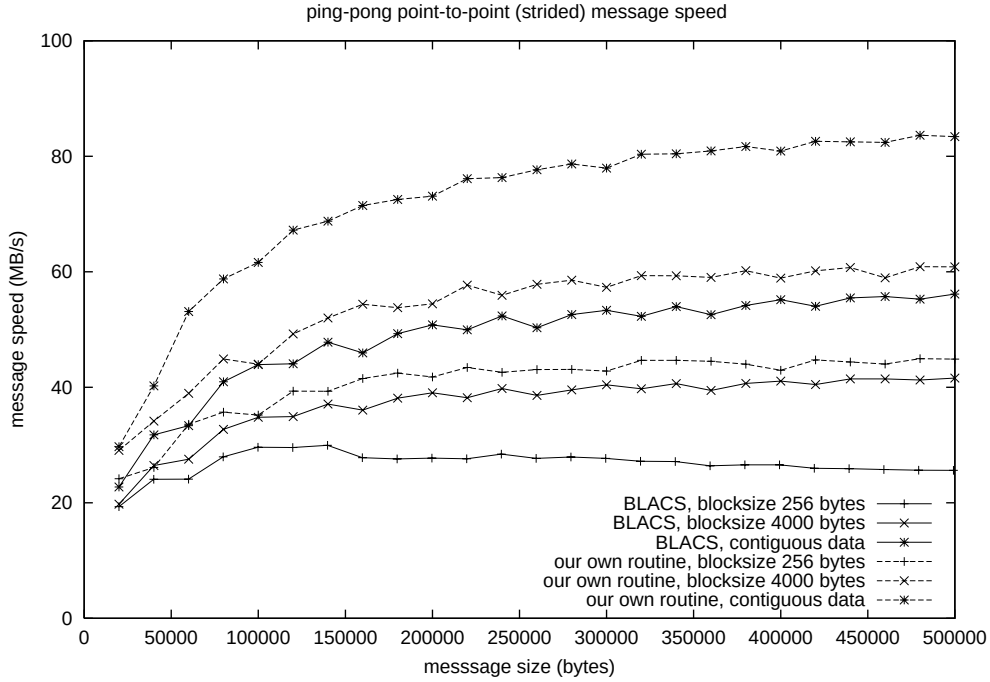



**Figure 3. Message speeds, BLACS and our routine, various blocksizes**

Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

[2] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical Report TCS-94-230, University of Tennessee Knoxville, April 1994.

[3] H. Ishihata, M. Takahashi, and H. Sato. Hardware of the AP3000 Parallel Server. *Fujitsu Scientific and Technical Journal*, 33(1):24–29, 1997.

[4] P. Mitra, D. Payne, R. van de Geijn L. Shuler, and J. Watts. Fast Collective Communication Libraries, Please. In *Proceedings of the Intel Supercomputing Users' Group*, 1995.

[5] David Sitsky and Paul Mackerras. A high-performance message passing Library for the Fujitsu AP3000. In *Proceedings of the Eighth Parallel Computing Workshop*, pages 245–251, Singapore, September 1998. National University of Singapore. paper P1-E.

[6] P. E. Strazdins. A Dense Complex Symmetric Indefinite Solver for the Fujitsu AP3000. Technical Report TR-CS-99-01, Computer Science Dept, Australian National University, May 1999.

[7] P.E. Strazdins. Lookahead and Algorithmic Blocking Techniques Compared for Parallel Matrix Factorization. In *PDCN'98: 10th International Conference on Parallel and Distributed Computing and Systems*, pages 291–297, Las Vegas, September 1998. IASTED.

[8] Peter E. Strazdins. Transporting Distributed BLAS to the Fujitsu AP3000 and VPP-300. In *Proceedings of the Eighth Parallel Computing Workshop*, pages 69–76, Singapore, September 1998. School of Computing, National University of Singapore. paper P1-E.