

Impact of the implementation of MPI point-to-point communications on the performance of two general sparse solvers¹

Patrick R. Amestoy²
Iain S. Duff³
Jean-Yves L'Excellent⁴
Xiaoye S. Li⁵

Technical Report TR/PA/03/14

March 27, 2003

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT

We examine the send and receive mechanisms of MPI and show how to implement message passing robustly so that performance is not significantly affected by changes to the MPI system. We discuss this within the context of two different parallel algorithms for sparse Gaussian elimination: a multifrontal solver (MUMPS), and a supernodal one (SuperLU). The performance of our initial strategies based on simple MPI point-to-point communication primitives is very sensitive to the MPI system, particularly the way MPI buffers are used. Using nonblocking communication primitives improves the performance and robustness, but at the cost of increased code complexity.

Keywords: Sparse linear systems, high performance computing, MUMPS, SuperLU, multifrontal Gaussian elimination, supernodal elimination, distributed memory code, message passing.

AMS(MOS) subject classifications: 65F05, 65F35, 65F50.

¹Current reports available at <http://www.cerfacs.fr/algos/reports/index.html>. An earlier version of this report also appeared as technical reports at ENSEEIHT-IRIT, Toulouse (RT/APO/01/4), INRIA, Lyon (RR-4372), and LBNL, California (LBNL-48968). The project was

supported by the France-Berkeley Fund and utilized resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

²Patrick.Amestoy@enseeiht.fr. ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse, France. Most of the work of this author was performed while he was on a sabbatical visit to NERSC.

³duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England. Supported in part by EPSRC Grant GR/R46441.

⁴Christof.Voemel@cerfacs.fr. Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, UMR CNRS-ENS Lyon-INRIA 5668, 46 allée d'Italie, F-69364 Lyon Cedex 07, France.

⁵xiaoye@nersc.gov. NERSC, Lawrence Berkeley National Lab, MS 50F, 1 Cyclotron Rd., Berkeley, CA 94720. The research of this author was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020 and NSF Grant No. ACI-9813362.

Contents

1	Introduction	1
2	MPI point-to-point communication	3
3	MPI tuning for MUMPS	4
3.1	Introducing immediate receives during factorization	5
3.2	Performance analysis	8
4	MPI tuning for SuperLU	9
5	Conclusions	12

1 Introduction

This paper discusses our understanding and experience of using MPI [7] for message passing in the context of the implementation of sparse direct solvers on multiprocessor machines. In particular, we study ways of making the message passing much more robust with respect to MPI system releases or the use of MPI internal buffers and consider in detail the kind of algorithmic changes that are required to enable this robustness. We feel that the lessons that we have learned are very useful to share with the community who use MPI for message passing in addition to those writing sophisticated numerical packages for distributed memory machines.

The direct solution of sparse linear systems using Gaussian elimination has a clear advantage over iterative methods in terms of numerical robustness and it remains the method-of-choice for many ill-conditioned systems. However, it is very challenging to implement such methods efficiently even on a single processor. One of the main reasons is because of fill-in in the matrix factorization. Moreover, numerical pivoting will involve dynamically tracking the fill-ins that are generated in a somewhat unpredictable way. Handling highly irregular data access and computation is further compounded by sophisticated computer architectures with several layers of memory hierarchy. We consider two different algorithms for sparse Gaussian elimination on distributed memory architectures where communication is by message passing using MPI. One is a multifrontal solver called MUMPS [1, 3], the other is a supernodal solver called SuperLU [9]. MUMPS and SuperLU use different algorithms and are representative of a wide range of sparse direct solvers. Both algorithms can be described by a computational graph [8] whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, MUMPS, this graph is a tree, the so called assembly tree. Some steps of Gaussian elimination are performed on a dense frontal matrix at each node of the assembly tree and the remaining Schur complement (or contribution block) is passed for assembly at the parent node. Furthermore, each node of this tree can be a source of dynamic task creation and scheduling. In the case of the supernodal code, SuperLU, the distributed memory version uses a right-looking formulation which, having computed the factorization of a block of columns then immediately sends the data to update the block columns in the trailing submatrix. The dependency graph of SuperLU is more complex than the assembly tree of the multifrontal method but it is statically built and analysed prior to factorization. With regard to parallelization, MUMPS therefore adopts a fully asynchronous approach, whereas SuperLU uses a loosely synchronous approach. Despite the differences between the two approaches, we found that their parallel performance was influenced by the MPI implementation in a somewhat similar way. A detailed quantitative comparison of the two solvers has appeared elsewhere [4]. In this paper, we only focus on the factorization phase for which the performance is affected most by the MPI implementation.

In both solvers, the initial parallel strategies were based on simple MPI point-to-point communication primitives. With such approaches, the parallel performance of both codes is rather sensitive to the MPI implementation; the use of MPI internal buffers in particular. We modified our codes to use more sophisticated nonblocking versions of MPI communication. This significantly improved the performance robustness (independence from the MPI buffering mechanism) and scalability, but at the cost of increased code complexity.

Throughout this paper, we will use a set of test problems to illustrate the performance of our algorithms. Our test matrices come from the forthcoming Rutherford-Boeing Sparse Matrix Collection [6] ¹, the industrial partners of the PARASOL Project², and the EECS Department of UC Berkeley³. The PARASOL test matrices are available from Parallab, Bergen, Norway⁴. We also use some problems generated by the 11-point discretization of a Laplacian on cubic grids. In that case, we vary the dimensions of the grids according to the number of processors so that the amount of work per processor is approximately constant.

<i>Real Unsymmetric Assembled (RUA)</i>				
Matrix name	Order	Nb. of entries	StrSym ^(*)	Origin
BBMAT	38744	1771722	0.54	Rutherford-Boeing (CFD)
ECL32	51993	380415	0.93	EECS Department of UC Berkeley
INVEXTR1	30412	1793881	0.97	PARASOL (Polyflow S.A.)
MIXTANK	29957	1995041	1.00	PARASOL (Polyflow S.A.)
<i>Real Symmetric Assembled (RSA)</i>				
Matrix name	Order	Nb. of entries	Origin	
BMWCR1_1	148770	5396386	PARASOL (MSC.Software)	
BMW3_2	227362	5757996	PARASOL (MSC.Software)	
CRANKSG2	63838	7106348	PARASOL (MSC.Software)	
SHIP_003	121728	4103881	PARASOL (Det Norske Veritas)	

Table 1: Test matrices. ^(*) StrSym is the number of nonzeros matched by nonzeros in symmetric locations divided by the total number of entries (so that a structurally symmetric matrix has value 1.0).

Note that **SuperLU** cannot exploit the symmetry and is unable to produce an \mathbf{LDL}^T factorization. So symmetric matrices are only used for performance results with **MUMPS**.

The results presented in this paper have been obtained on the Cray T3E-900 (512 DEC EV-5 processors, 256 Mbytes of memory per processor, 900 peak Megaflop rate per processor) from NERSC at Lawrence Berkeley National Laboratory. The peak interprocessor communication bandwidth is 300 Mbytes/s, and the latency is 4 microseconds. Although we mainly report results on this machine, our codes are designed to be portable to any system supporting MPI and indeed have been run on many other systems. Similar performance gains have been obtained on other computers such as an IBM SP. This is the case because the buffering ideas are common to all MPI implementations even if the precise implementation of nonblocking communications may vary.

The rest of the paper is organized as follows. In Section 2, we discuss the MPI point-to-point communication primitives used in the solvers. We discuss these aspects in detail for **MUMPS** in Section 3 and for **SuperLU** in Section 4. In Section 5, we present some general conclusions.

¹Web page <http://www.cse.clrc.ac.uk/nag/hb/>

²EU ESPRIT IV LTR Project 20160

³Matrix ECL32 is included in the Rutherford-Boeing Collection

⁴Web page <http://www.parallab.uib.no/parasol/>

2 MPI point-to-point communication

In both solvers, the factorization algorithms almost use only MPI point-to-point communication primitives, that is, one process sends a message and another process receives the message. The send and receive operations are either *blocking* (`mpi_send/mpi_recv`) or *nonblocking* (`mpi_isend/mpi_irecv`). In the blocking version, the send call blocks until the send buffer can be reclaimed, and the receive call blocks until the receive buffer contains the message. The nonblocking functions come in two parts: the posting functions that initiate the operations and the test-for-completion functions that complete the requested operations. This allows the possible overlap of message transmittal with computation, or the overlap of multiple message transmittals with one another.

However, the actual semantics of the primitives depend on the underlying protocols that implement them. This usually amounts to a trade-off between buffering/copying and synchronization. For example, if a protocol attempts to minimize buffering and copying of data, the semantics for blocking calls might be handshaking. But if a protocol attempts to minimize a process' amount of blocking time, it might use buffering semantics. Here is how the MPI standard defines `mpi_send` semantics [10, pp. 32]:

“... It does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer.
... The MPI implementation might block the sender or it might buffer the data. ”

Very often, an MPI implementor chooses to use two different protocols depending on the length of the message:

- *Short protocol (eager protocol)* for small messages.
The sender copies the data into the system buffer, and returns immediately without waiting for the matching receive.
- *Long protocol* for large messages.
The sender first sends a “request-to-send” message to the receiver, then waits for the receiver to send back a “ready-to-receive” message. The sender now transmits the message data directly into the receiver’s user space without buffering. This protocol requires handshaking of the sender and receiver, but the message transfer overhead is smaller than for the short protocol because we do not pay the extra cost of copying.

Whether a message is short or long is usually determined by its size relative to a threshold. For example, on the Cray T3E, the user may adjust this by setting an environment variable `MPI_BUFFER_MAX`. If a message length exceeds this value, the long protocol will be used; otherwise, the short protocol with an internal MPI buffer is used. Table 2 lists the default values of `MPI_BUFFER_MAX` with various versions of the MPI implementation in the Cray’s Message Passing Toolkit (MPT). Note that, on the T3E, the total amount of memory available for message buffering (maximum size for internal MPI buffers on one process) is controlled by `MPI_BUFFER_TOTAL` which is, by default, unlimited. The memory used for MPI internal buffers will then depend on the size and amount of short messages waiting to be received, and could be large depending on `MPI_BUFFER_MAX`. Similar MPI environment variables `MP_EAGER_LIMIT` and `MP_BUFFER_MEM` are available on

the IBM SP and correspond to respectively `MPI_BUFFER_MAX` and `MPI_BUFFER_TOTAL` on the T3E.

MPT version	Date	default <code>MPI_BUFFER_MAX</code>
1.3.0.3	September, 1999	unlimited
1.3.0.4	February, 2000	4 Kbytes
1.4.0.0	August, 2000	0

Table 2: MPI default message length under which the short protocol is used on the T3E.

We now describe some details of the mechanisms specific to the T3E, and that are independent of the send and the receive are actually performed (`mpi_send` or `mpi_isend`, `mpi_recv` or `mpi_irecv`). For the short protocol, the sender writes directly into the MPI receive buffer of the destination process. When the receiver actually issues a reception, only copying from the MPI buffer to the user space is involved. For the long protocol, the communication of the effective data only starts when the receive instruction is posted and no intermediate MPI receive buffer is involved. Doing an `mpi_send` the sender would then be blocked whereas with an `mpi_isend` only the working area effectively sent could not be reused until reception. Note also that with a threshold `MPI_BUFFER_MAX` set to “unlimited” (short protocol always used), we expect communications based on standard sends and receives to perform very similarly to asynchronous immediate communications. However, with immediate communications, performance should not depend on `MPI_BUFFER_MAX` buffering so that by setting `MPI_BUFFER_MAX` small, the memory associated with the MPI buffer and the copy from the MPI buffer to the user space can be avoided. We refer here to MPI internal buffers. There can also be a buffer defined by the solver which may be a separate staging area or may be directly in the working space of the user process. In the following, we will always prefix the term buffer by MPI when we mean the MPI internal buffers so that the use of the term buffer, without this prefix, will always refer to the user-defined buffer space.

3 MPI tuning for MUMPS

The initial version of MUMPS will be referred to as *Version 4.0* whereas the modified version will be referred to as *Version 4.1*.

In Version 4.0, the communications are asynchronous and are based on the use of an immediate send (`mpi_isend`). The receiver normally matches the asynchronous send with a test for the availability of the message, potentially followed by an effective reception of the message (`mpi_recv`). A problem with this mechanism occurs for large messages. In this case, independently of the time difference between the issue of the send and the issue of the receive, almost all the data to be exchanged will start to be sent only when the receive process actually issues a receive instruction providing the user space required for the communication to proceed. This can very significantly affect the potential algorithmic overlapping between computation and communication, and thus delay the effective reception of messages. However, if we can use an immediate receive (`mpi_irecv`), which can ideally be interpreted as having a separate “spawned” process implementing the reception, the reception can progress in parallel with the process that issued the `mpi_irecv`,

so that potentially the receive can have completed (that is the complete message is available in the user space of the process issuing the `mpi_irecv`) at the time when we test for the availability of the message. Note that, by doing so, when the MPI buffer is used (short protocol) we have also overlapped the copying from the MPI buffer to the user space.

We illustrate, in Table 3, the impact of the `MPI_BUFFER_MAX` threshold on the performance of our algorithm on a large matrix from our test set. The standard receive (`mpi_recv`) is used to match the immediate send `mpi_isend`. One first sees that, on our

MPI_BUFFER_MAX (in bytes)								
0	128	512	1K	4K (*)	64K	512K	2Mega	8Mega
37.7	37.0	37.4	38.3	37.6	32.8	28.3	26.4	26.4

Table 3: Influence of `MPI_BUFFER_MAX` on the time (in seconds) for the factorization of matrix `CRANKSG2` on 8 processors. `mpi_recv` is used to match `mpi_isend`. (*) default value with version MPT 1.3.0.4.

example, the protocol used for communication (short or long) strongly influences the factorization time. Secondly, with the Version 1.3.0.4 default value of `MPI_BUFFER_MAX` (4 Kbytes), the use of a standard receive (`mpi_recv`) to match an immediate asynchronous send (`mpi_isend`) does not lead to a good overlapping of communication with computation. As we explained before, matching the immediate sends (`mpi_isend`) with immediate receives (`mpi_irecv`) should address both issues (that is, independence with respect to the protocol used and overlapping of communication).

Although, in the context of MUMPS, the use of an immediate receive seems quite natural, we explain in Section 3.1, why it required more algorithmic developments than might have been expected. The main issue with using an immediate receive in our very asynchronous environment is that we cannot tell *a priori* which message we are receiving. That is, the `mpi_irecv` request must be sent to receive any type of message from any source. In our implementation, we restrict ourselves to a single `mpi_irecv` pending request since this limits the memory associated with reception buffers in the user space and allows the algorithm to respect a necessary order of messages originating from the same source.

In Section 3.2, we study in more detail the benefit coming from our main algorithmic modification (that is, the use of asynchronous immediate receives) and explain the performance gains.

3.1 Introducing immediate receives during factorization

In this section, we describe the modifications to MUMPS Version 4.0 required for the introduction of asynchronous immediate receives.

As we mentioned in the previous sections, in MUMPS Version 4.0, communications are asynchronous. They are based on immediate sends with explicit buffering in the user space. A Fortran module was designed for this purpose and is briefly described in [3]. In order to avoid the drawback of centralized scheduling, MUMPS uses distributed dynamic scheduling; all tasks ready to be activated by a process are stored in a pool of tasks local to the process. These tasks correspond to the initiation of new activities (assembly or factorization of a new frontal matrix at a node in the assembly tree) which may require

the participation of other processes (the decision to ask for help from other processes is done dynamically). Each process then executes Algorithm 1. Comments are in parentheses using small and slanted fonts.

Algorithm 1 *Dynamic scheduling*

```

While not all tasks in the dependency graph processed
  If [ local pool empty ] Then
    blocking receive for a message; process ( received message )
  Else If [ message available ( - mpi_iprobe - ) ] Then
    receive and process message
  Else
    extract work from the pool, and process it
  End If
End While

```

On each process, the arrival of a message will be the key point for the synchronization and scheduling of the work. These messages are of various types and may contain: (i) information that the destination process is required to help the sender – which has just extracted a new task from the pool, (ii) parts of the Schur complement that needs to be assembled on another process at the parent node in the assembly tree, (iii) factorized blocks required to perform updates on part of a frontal matrix, or (iv) various symbolic structural information.

Furthermore, it is important to note that message reception can also be forced in the following two situations:

1. *Insufficient space in send buffer:*

The message cannot be stored in the MUMPS local send buffer used for `mpi_isend`. To avoid deadlock (since all processes could potentially be in that same situation), the corresponding process tries to receive messages and process corresponding tasks until space becomes available in its local send buffer.

2. *Task ordering:*

A process may want to receive and treat an “expected” message required to process the current task. This message needs to be treated in priority in order to finish the current task and free the associated temporary working space.

Algorithm 2 *Dynamic scheduling with immediate receive*

```

While not all tasks in the dependency graph processed
  Blocking = false; ExpectedMessage = AnyMessage
  If [ local pool empty ] Blocking = true
    Try_to_receive_and_process_message ( Blocking, ExpectedMessage, SetIrecv )
  If [ no message received and pool not empty ] Then
    extract work from the pool and process it
  End If
End While

```

To modify the dynamic scheduling algorithm in the context of immediate receives, we introduce in Algorithm 2 the procedure *Try_to_receive_and_process_message* for which the parameters `Blocking` and `ExpectedMessage` indicate whether we want to wait for the arrival of a message of type `ExpectedMessage`. `ExpectedMessage` is used to characterize both the process source and the message label (or tag) of the expected message, with the convention that `AnyMessage` will correspond to receiving a message from any process source and of any label. The parameter `Blocking` then specifies whether we wait until the reception of the message of type `ExpectedMessage` or not. The parameter `SetIrecv` has been introduced to explicitly control the activation of immediate receive calls. Its use will be justified when we describe the algorithm *Try_to_receive_and_process_message*.

Algorithm 3 *Try_to_receive_and_process_message*(`Blocking`,`ExpectedMessage`,`SetIrecv`)

```

0. If [ Receive request pending ] Then
1.   MessRecv=false; RightMessage=true
2.   If [ Blocking ] Then
3.     Wait for the end of pending request; ( - mpi_wait - )
4.     MessRecv=true ( - message is in buffer - )
5.     If [ message received  $\neq$  ExpectedMessage ] Then
6.       RightMessage=false
7.       Blocking probe ( ExpectedMessage ) ( - mpi_probe - )
8.     End If
9.   Else If [ Message in buffer ] MessRecv=true
10.  End If
11.  If [ MessRecv ] Then
12.    If [ Not RightMessage ] SetIrecv = false
13.    Process ( message already in buffer, SetIrecv )
14.    If [ Not RightMessage ] Then
15.      SetIrecv = true
16.      If [ ExpectedMessage ready to be received ( - mpi_iprobe - ) ] Then
17.        Receive and process ( ExpectedMessage, SetIrecv )
18.      Else ( - expected message is already received and processed - )
19.      End If
20.    End If
21.  End If
22. Else If [ Blocking ] Then
23.   Blocking receive ( AnyMessage ); Process ( received message, SetIrecv )
24. Else If [ Message ready to be received ( - mpi_iprobe - ) ] Then
25.   Receive and process ( message ready, SetIrecv )
26. End If
27. If [ No receive request pending and SetIrecv ] Reactivate immediate receive

```

The procedure *Try_to_receive_and_process_message* is described by Algorithm 3. During dynamic scheduling (Algorithm 2) we always process messages on the basis of first message arrived first processed (`ExpectedMessage` is set to `AnyMessage`). This will also be the case for situation 1 where the objective is only to receive messages to avoid deadlock. While processing tasks of type task ordering, we want indeed to perform a blocking receive

on `ExpectedMessage`. Such cases are illustrated in [2, 3] and are due to the fact that, although the algorithm is asynchronous, we have to maintain a partial order between the tasks. By doing so we control the amount of temporary memory required. This also helps in freeing space in our send buffers and MPI internal buffers. Note that our asynchronous algorithm has been designed (thanks to a “good” ordering of the message sent) so that, when an “expected message” needs to be received, we can guarantee that this message has already been sent. A blocking receive on this message can thus safely be performed. The main difficulty introduced by the use of a blocking receive on a given message is that a “wrong” message might already be in our receive buffer because of an asynchronous pending receive request. The parameter `ExpectedMessage` has thus been introduced to characterize the expected message. Note that, if `ExpectedMessage` is true in a call to Algorithm 3, then `Blocking` must also be true. For the sake of clarity, two local variables have been introduced. `MessRecv` indicates that a message has been received during the pending receive request. `RightMessage` is true when the message received is the expected one (i.e. it has the same characteristics as `ExpectedMessage`).

Between lines 5 and 8 of the algorithm we are in the situation of having already received a message in our local buffer which is not the expected one. Since we know that the expected message has been sent we can do a blocking probe on the expected message (line 7) and force the current process to wait for the availability of the expected message before processing the current message in the buffer. This will enable us to perform a nonblocking probe on the expected message at line 16 and *conclude at line 18 that the message must have been processed if it is not ready to be received*. In fact, we must also guarantee that the expected message has not been stored in the receive buffer by an immediate receive request. Therefore, we must be sure that between lines 7 and 16 another immediate receive has not been issued. The only place which could cause the activation of an immediate receive is at line 13 where Algorithm 3 might be called recursively. A receive can be issued during the processing of almost any message giving rise to a situation 1 (insufficient space in the send buffer) or 2 (task ordering). To avoid such an occurrence, we suspend the activation of immediate receives at line 12 and only reactivate it again at line 15. At line 27, we must then test whether activation of an immediate receive is authorized.

There are two reasons why we expect the use of immediate receives to improve the performance. Firstly, the actual reception is overlapped with the computation (this is particularly important if the long protocol is used). Secondly, the space in the send buffer can be reused earlier, avoiding the idle time in the sending process that may occur in situation 1.

3.2 Performance analysis

On our largest test matrices we show, in Table 4, the impact of using immediate receive during the factorization phase. The default size of our send buffer is twice the size of the largest message. The results shown in this section were obtained with release 1.3.0.4 of the CRAY operating system for which the threshold under which messages are buffered (`MPI_BUFFER_MAX`) is 4 Kbytes.

One can see that usually relatively larger gains are obtained on a smaller number of processors. Symmetric matrices seem to benefit more from this modification. This comes from the fact that our parallel algorithms [3] involve a relatively larger number of messages

Matrix	Ordering	mpi_irecv	Number of processors				
			4	8	16	32	64
BMWCR1_1	MeTiS	OFF	—	—	24.7	20.4	11.4
		ON	—	—	22.7	16.6	9.6
BMW3_2	MeTiS	OFF	—	24.6	16.4	9.2	6.2
		ON	—	22.6	15.9	8.2	5.7
CRANKSG2	MeTiS	OFF	—	37.6	22.1	13.3	8.9
		ON	—	26.4	18.1	11.3	8.0
SHIP_003	MeTiS	OFF	—	—	37.3	24.5	17.9
		ON	—	—	30.8	21.1	15.7
BBMAT	AMD	OFF	46.0	25.7	19.9	17.2	12.9
		ON	45.2	24.7	18.0	15.2	12.5
ECL32	AMD	OFF	56.7	38.4	26.5	19.9	15.3
		ON	54.0	35.4	23.4	18.4	15.7
INVEXTR1	AMD	OFF	37.7	26.9	19.4	21.6	20.0
		ON	36.8	25.6	19.5	21.3	18.9
MIXTANK	AMD	OFF	57.3	36.7	25.4	23.2	17.1
		ON	52.9	33.5	24.1	19.7	16.9

Table 4: Influence of the use of `mpi_irecv` on the time (in seconds) for factorization of MUMPS using MPT version 1.3.0.4 (`MPI_BUFFER_MAX` is 4K bytes). — means not enough memory.

on symmetric matrices than on unsymmetric matrices that might saturate more the send buffer and the internal MPI buffers.

Furthermore, as one might expect from the previous discussion, we show (compare the results in Tables 3 and 5), that the new code based on immediate receives (`mpi_irecv`) is very much less sensitive to the use of the internal MPI buffer than the initial version based on standard receives (`mpi_recv`).

MPI_BUFFER_MAX (in bytes)								
0	128	512	1K	4K (*)	64K	512K	2Mega	8Mega
27.1	27.3	26.5	26.6	26.4	26.2	26.2	26.4	26.2

Table 5: Influence of `MPI_BUFFER_MAX` on the time (in seconds) for the factorization of matrix CRANKSG2 on 8 processors using MPT version 1.3.0.4. `mpi_irecv` is used to match `mpi_isend`. (*) default value on the CRAY T3E.

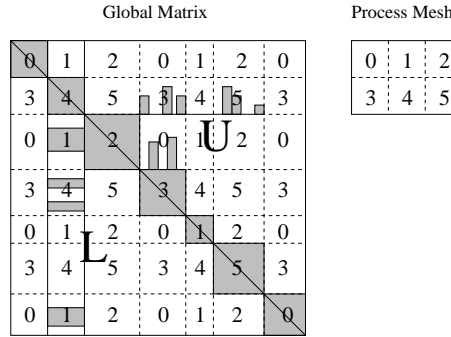
4 MPI tuning for SuperLU

We first describe the matrix-to-process mapping and the initial parallel factorization algorithm in SuperLU. We then explain how the algorithm can be improved by introducing nonblocking send and receive to better utilize the pipeline effect. This pipelined execution leads to more overlap between communication and computation, and shortens the critical path.

Our matrix partitioning is based on the notion of an *unsymmetric supernode* introduced in [5]. A supernode is a range ($r : s$) of columns of L with the triangular block just below the diagonal being full, and the same nonzero structure elsewhere (this is either full or

zero). This supernode partition is used as the block partition in *both* row and column dimensions. If there are N supernodes in an n -by- n matrix, there will be N^2 blocks of submatrices of non-uniform size. Figure 1 illustrates such a block partition. The off-diagonal blocks may be rectangular and may not be full. Some of the off-diagonal blocks may be entirely zero. The P processes are arranged as a 2D mesh of dimension $P_r \times P_c = P$. We use a block-cyclic layout, where block (I, J) (of L or U) is mapped onto the process at coordinate $((I - 1) \bmod P_r, (J - 1) \bmod P_c)$ of the process mesh.

Figure 1: The 2D block-cyclic layout and the data structure used in SuperLU.



Before we describe the algorithm, we first introduce the following notation for the process IDs that will be used to simplify the discussion. Matlab notation is used for integer ranges and submatrices.

- $PROC_c(K)$: the set of column processes that own block column K
For example, in Figure 1, $PROC_c(3) = PROC_c(6) = \{2, 5\}$.
- $PROC_r(K)$: the set of row processes that own block row K
For example, in Figure 1, $PROC_r(1) = PROC_r(3) = \{0, 1, 2\}$.
- P_K : the process in $PROC_c(K) \cap PROC_r(K)$
- me : the process rank as illustrated in Figure 1

The outer loop of the parallel factorization runs from block column 1 to N . There are three steps in the K -th iteration of the loop. In step (1), only processes $PROC_c(K)$ participate in factoring block column $L(K : N, K)$. In step (2), only processes $PROC_r(K)$ participate in factoring block row $U(K, K + 1 : N)$. If b is the block size of the K -th block column/row, then the rank- b update by $L(K + 1 : N, K)$ and $U(K, K + 1 : N)$ in step (3) represents most of the work and also exhibits more parallelism than the other two steps.

The actual implementation uses a *pipelined* organization so that processes $PROC_c(K + 1)$ will start step (1) of iteration $K + 1$ as soon as the rank- b update (step (3)) of iteration K to block column $K + 1$ finishes, before completing the update to the trailing matrix $A(K + 1 : N, K + 2 : N)$ owned by $PROC_c(K + 1)$. We found that the block factorization tasks are usually on the critical path, whereas the update tasks in step (3) are often overlapped with the other tasks. There is a lack of parallelism for the factorization tasks

in steps (1) and (2), because only one set of column processes or row processes participate in these tasks. Our pipelined mechanism alleviates this problem. For instance, on 64 processors of the Cray T3E, we observed speedups of between 10% and 40% over the non-pipelined implementation.

In an earlier version of the code, we used MPI’s standard send and receive operations `mpi_send` and `mpi_recv` for the message transfer tasks. We saw idle time (longer send) during the sending of $L(:, K + 1)$ for process P_{K+1} on the critical path. This could happen if the sender and receiver are required to handshake before proceeding, as is the case with large messages using the *long protocol*. That is, process P_{K+1} posts `mpi_send` long before processes $PROC_r(K)$ post the matching `mpi_recv`, because $PROC_r(K)$ are still busy with the updates for the K -th iteration. The sender P_{K+1} must then be blocked to wait for `mpi_recv`. For this code, we observed performance differences between setting `MPI_BUFFER_MAX` to 4 Kbytes and to “unlimited”, meaning that all messages use the short protocol. Table 6 shows the timing differences. For this experiment, the matrices are obtained from the 11-point discretization of the Laplacian operator on 3D cubic grids. The grid sizes are increased with increasing number of processors so that the number of operations per processor is roughly constant. The most dramatic difference is on 2 processors, where the smaller buffer results in a 74% speed loss. This is because on 2 processors, more messages are larger than `MPI_BUFFER_MAX`, and their transfers have to use the long protocol. With more processors, the average message size becomes smaller, and there is more chance to use the short protocol.

As a comparison, in the same table, we give the timings of the improved code after introducing `mpi_isend` and `mpi_irecv` (marked “new”). The detailed algorithmic change will be described soon. We see that the performance of the new code is not significantly influenced by the buffer size and is comparable to the old code using the value “unlimited”.

	Nprocs	1x1	1x2	2x2	2x4	4x4	4x8	8x8	8x16
	Grid size	29	33	36	41	46	51	57	64
<code>mpi_send/</code> <code>mpi_recv</code> (old)	<u><code>MPI_BUFFER_MAX</code></u>								
	unlimited	55.9	62.3	53.3	61.5	62.7	65.7	76.2	80.7
	4 Kbytes	55.9	108.2	92.4	102.5	104.2	101.6	119.3	111.0
<code>mpi_isend/</code> <code>mpi_irecv</code> (new)	<u><code>MPI_BUFFER_MAX</code></u>								
	unlimited	55.9	61.8	53.4	61.3	62.9	66.0	76.8	75.9
	4 Kbytes	55.9	62.8	53.8	62.4	64.2	68.5	78.3	77.0

Table 6: Times in seconds for two versions of SuperLU’s factorization for the cubic grid problems with `MPI_BUFFER_MAX` set to unlimited and to 4Kbytes. MPT version 1.3.0.4 is used.

It is very unpleasant that the performance of our code depends on the use of MPI system buffers. The cure for this problem is to use the nonblocking send and receive primitives, `mpi_isend` and `mpi_irecv`. This requires reorganizing the pipeline structure of the code. The basic ideas are as follows.

- For the sender, we simply replace `mpi_send` by `mpi_isend` when we send the L block.
- For the receiver, we will post `mpi_irecv` much earlier than we actually need the data. For example, for processes $PROC_r(K)$, we could post receive $L(:, K + 1)$ before

updating the trailing matrix at the K -th iteration. That is, as soon as we have received a message using `mpi_wait`, we will post the `mpi_irecv` for the next message, before performing the local computation with the just-arrived message.

To implement this scheme, we need to provide user-level buffer space to accommodate the messages in transit. Since for each process, there is only one outstanding message to be received, we only need one extra buffer. Algorithm 4 sketches the modified pipelined algorithm using `mpi_isend` and `mpi_irecv`. The main modification is in step (3). In the new algorithm, the original step (3) is split into two substeps (3.1) and (3.2). Step (3.1) implements a look-ahead scheme. Here, we only update the $(K + 1)$ -st block column, then immediately factorize this column and post send and receive of the factorized column for the $(K + 1)$ -st iteration of the loop. This message transfer will overlap with the rest of the trailing submatrix update appearing in step (3.2). In step (1), the processes wait for the posted send and receive to complete. In particular, `mpi_wait` in line 9 is matched with the posted `mpi_isend` in line 23 (and 3); `mpi_wait` in line 11 is matched with the posted `mpi_irecv` in line 25 (and 5).

In Table 7, we show the performance gain with the new code, when the default value for `MPI_BUFFER_MAX` is 0 Bytes in the latest MPT release (no short protocol). Clearly, the amount of gain is matrix dependent, and mainly depends on the message size. With an increasing number of processors, the message size is usually smaller, and the performance gain is less dramatic than on a smaller number of processors. The peak performance gain occurs on 4 processors where the new code is almost twice as fast as the old code.

Matrix	Ordering	mpi_isend/ mpi_irecv	Number of processors				
			2x2	2x4	4x4	4x8	8x8
BBMAT	AMD	OFF	113.1	57.2	31.7	17.2	11.3
		ON	64.7	36.6	21.3	12.8	9.2
ECL32	AMD	OFF	194.7	97.3	51.1	27.3	17.1
		ON	106.8	56.7	31.2	18.3	12.3
INVEXTR1	AMD	OFF	88.1	44.2	23.9	12.9	8.3
		ON	49.7	30.0	16.5	10.1	7.1
MIXTANK	AMD	OFF	131.2	65.8	34.5	18.4	11.0
		ON	70.8	38.2	21.1	11.9	7.9

Table 7: Influence of the use of `mpi_isend/mpi_irecv` on the time (in seconds) for the factorization in SuperLU. MPT version 1.4.0.0 is used, where the default `MPI_BUFFER_MAX` is 0 Bytes.

5 Conclusions

In this paper, we have studied in detail how we can design a message passing code that is robust against changes in MPI release and MPI buffering mechanisms and have indicated, for two sparse direct codes, the algorithmic changes necessary to achieve this. Our main technique is to use `mpi_irecv` as well as `mpi_isend` but we show that, whether the underlying algorithm is asynchronous or essentially synchronous, significant changes are required to use these protocols. Although we mainly based our study on a CRAY T3E, similar results have been obtained on more recent platforms such as the IBM SP, as we finally illustrate in Table 8.

Algorithm 4 *Modified SuperLU factorization algorithm with nonblocking send and receive.*

```

/* — Set up the initial stage for the pipeline — */
1. if [  $me \in PROC_c(1)$  ] then
2.   Factorize block column  $L(1 : N, 1)$ 
3.   Post send  $L(1 : N, 1)$  to the processes in my row who need it ( $- mpi\_isend -$ )
4. else
5.   Post receive  $L(1 : N, 1)$  from one process in  $PROC_c(1)$  (if I need it) ( $- mpi\_irecv -$ )
6. endif

/* — Main pipeline loop — */
7. for block  $K = 1$  to  $N$  do
8.   (1) if [  $me \in PROC_c(K)$  ] then
9.     Wait for the posted send of  $L(K : N, K)$  to complete ( $- mpi\_wait -$ )
10.  else
11.    Wait for the posted receive of  $L(K : N, K)$  to complete ( $- mpi\_wait -$ )
12.  endif
13.  (2) if [  $me \in PROC_r(K)$  ] then
14.    Factorize block row  $U(K, K + 1 : N)$ 
15.    Send  $U(K, K + 1 : N)$  to processes in my column who need it
16.  else
17.    Receive  $U(K, K + 1 : N)$  from one process in  $PROC_r(K)$  (if I need it)
18.  endif
19.  (3.1) if [  $K + 1 \leq N$  ] then
20.    /* — Factor-ahead scheme — */
21.    if [  $me \in PROC_c(K + 1)$  ] then
22.      Update  $(K + 1)$ -st column  $A(:, K + 1) \leftarrow A(:, K + 1) - L(:, K) \cdot U(K, K + 1)$ 
23.      Factorize block column  $L(:, K + 1)$ 
24.      Post send  $L(:, K + 1)$  to the processes in my row who need it ( $- mpi\_isend -$ )
25.    else
26.      Post receive  $L(:, K + 1)$  from one process in  $PROC_c(K + 1)$  ( $- mpi\_irecv -$ )
27.    endif
28.  (3.2) for  $J = K + 2$  to  $N$  do
29.    for  $I = K + 1$  to  $N$  do
30.      if [  $me \in PROC_r(I)$  and  $me \in PROC_c(J)$ 
31.        and  $L(I, K) \neq 0$  and  $U(K, J) \neq 0$  ] then
32.        Update trailing submatrix  $A(I, J) \leftarrow A(I, J) - L(I, K) \cdot U(K, J)$ 
33.      endif
34.    end for
35.  end for
36. end for

```

Nprocs	1	2	4	8	16	32	64	128
Grid size	29	33	36	41	46	51	57	64
MUMPS (LDL^T factorization)								
ISEND + RECV	5.5	7.9	6.9	8.9	10.1	14.2	22.1	31.9
ISEND + IRECV	5.5	7.9	6.9	8.0	9.3	13.3	20.7	31.6
SuperLU (LU factorization)								
SEND + RECV	13.7	23.3	20.7	23.2	25.2	26.0	38.2	48.7
ISEND + IRECV	13.7	16.0	14.3	17.4	19.3	21.8	34.5	41.8

Table 8: Factorization time in seconds for the cubic grid problems on the IBM SP at NERSC. Results for MUMPS are with the symmetric code, so SuperLU performs twice more operations.

Our application of solving large sparse systems is a significant one and inter alia we have shown the complexity of this problem when implementing such codes on distributed memory computers. We hope that our findings will be of interest both for those concerned with the efficient use of MPI and for the sparse matrix community.

Acknowledgments

We want to thank James Demmel, Jacko Koster and Rich Vuduc for very helpful discussions. We are grateful to Chiara Puglisi for her comments on an early version of this report and her help with the presentation.

References

- [1] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
- [2] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Parallélisation de la factorisation LU de matrices creuses non-symétriques pour des architectures à mémoire distribuée. *Calculateurs Parallèles Réseaux et Systèmes Répartis*, 10(5):509–520, 1998.
- [3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.
- [4] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, 2001.
- [5] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.
- [6] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS.
- [7] Message Passing Interface Forum. <http://www-unix.mcs.anl.gov/mpi/index.html>.
- [8] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.
- [9] X. S. Li and J. W. Demmel. SuperLU-DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. Technical Report LBNL-49388, Lawrence Berkeley National Laboratory, 2003. To appear in ACM Trans. Mathematical Software, 2003.
- [10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.