Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

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

Permalink
https://escholarship.org/uc/item/9812h6z5

Authors
Amestoy, Patrick R.
Duff, Iain S.
L'Excellent, Jean-Yves
et al.

Publication Date
2001-10-10

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Impact of the Implementation of MPI Point-to-Point Communications on the Performance of Two General Sparse Solvers[*]

Patrick R. Amestoy[†] Iain S. Duff[‡] Jean-Yves L'Excellent[§] and Xiaoye S. Li[¶]

October 19, 2001

Abstract

We examine the send and receive mechanisms of `MPI` and 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 more sophisticated nonblocking communication primitives improves the performance robustness and scalability, but at the cost of increased code complexity.

[*]The project is supported by the France-Berkeley Fund. This project also 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.

[†]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.

[‡]I.Duff@rl.ac.uk, CERFACS, 42 Ave G Coriolis, F-31527 Toulouse Cedex 1, France. Also at RAL, Chilton, Oxfordshire, England OX11 0QX. Supported in part by EPSRC Grant GR/M78502.

[§]jylexcel@ens-lyon.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.

[¶]xiaoye@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

This paper discusses our understanding and experience of using `MPI` 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 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 never mind on multiprocessor machines. 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 somewhat unpredictable way. Handling highly irregular data access and computation is further compounded by sophisticated computer architectures with several layers of memory hierarchy. We have parallelized two different algorithms to perform sparse Gaussian elimination for distributed memory architectures where communication is by message passing using `MPI`. One is a multifrontal solver called `MUMPS` [3, 1], the other is a supernodal solver called `SuperLU` [9]. `MUMPS` and `SuperLU` use different algorithms but are representative of a wide range of sparse direct solvers. With regard to parallelization, `MUMPS` adopts a fully asynchronous approach, whereas `SuperLU` uses a loosely synchronous approach. Despite the differences, we found that their parallel performance was influenced by the `MPI` implementation in somewhat similar way. A detailed quantitative comparison of the two solvers has appeared elsewhere [4]. In this paper, we focus only on the factorization phase in 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 very sensitive to the `MPI` implementation, the use of the `MPI` internal buffer in particular. We then modified our codes to use more sophisticated nonblocking versions of `MPI` communication. This significantly improved the performance robustness (independent of the `MPI` buffering mechanism) and scalability, but at the cost of the increased code complexity.

The rest of the paper is organized as follows. Section 2 gives the characteristics of our parallel algorithms and presents the test environment used. In Section 3 we discuss the `MPI` point-to-point communication primitives used in the solvers. We discuss these aspects in detail for `MUMPS` in Section 4 and for `SuperLU` in Section 5. In Section 6, we present some general conclusions.

2 Parallel algorithms and test environment

The multifrontal algorithm used in `MUMPS` and the supernodal algorithm used in `SuperLU` are two representative algorithms to perform sparse Gaussian elimination. In this section, we briefly describe the main characteristics of the algorithms and highlight the major differences between them.

Both algorithms can be described by a computational graph [7] whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, `MUMPS`, this graph is a tree. Some steps of Gaussian elimination are performed on a dense frontal matrix at each node of the tree and the Schur complement (or contribution block) that remains is passed for assembly at the parent node. 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.

There are some common aspects in both solvers. Firstly, in the preprocessing phase, we first use row or column permutations to permute large entries onto the diagonal (the code for this is `MC64` from HSL [8]). For `MUMPS`, it reduces the number of off-diagonal pivots and the number of delayed pivots and may also increase the structural symmetry. For `SuperLU`, it should reduce the need for small diagonal perturbations and the number of iterative refinement steps. After this step, a symmetric ordering (e.g., minimum degree or nested dissection based on the graph of $A + A^T$) is used to preserve sparsity. Secondly, in the postprocessing phase, iterative refinement can be invoked to improve the accuracy of the solution. `MUMPS` rarely needs it; `SuperLU` sometimes needs it, and one step of refinement often suffices.

The main differences between the two codes lie in the factorization phase. These are outlined below:

- `MUMPS`
  - multifrontal based on elimination tree [10] of $A + A^T$
  - partial threshold pivoting
  - partial static mapping using elimination tree (1D for the frontal matrices and 2D for the root)
  - asynchronous, dynamic distributed scheduling (in part because of the fact that numerical pivoting causes a delay in pivot selection with consequent modification of data structures during the numerical factorization).
- `SuperLU`
  - supernodal fan-out based on elimination DAGs [7]
  - static pivoting with possible half-precision perturbations on the diagonal
  - static 2D irregular block-cyclic mapping using supernode structure
  - loosely synchronous scheduling with pipelining.

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 Project[2], and the EECS

[1]Web page `http://www.cse.clrc.ac.uk/Activity/SparseMatrices/`

[2]EU ESPRIT IV LTR Project 20160

Department of UC Berkeley[3]. The PARASOL test matrices are available from Parallab, Bergen, Norway[4].

Real Unsymmetric Assembled (RUA)				
Matrix name	Order	No. 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.)

Real Symmetric Assembled (RSA)			
Matrix name	Order	No. of entries	Origin
BMWCRA_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`. Matrices MIXTANK and INVEXTR1 have been modified because of underflow values in the matrix files. To keep the same sparsity pattern, we have replaced all entries with exponents smaller than -300 by numbers with the same mantissa but with exponents of -300. For each linear system, the right-hand side vector is generated so that the true solution is a vector of all ones.

All 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 only study the results on this one machine, our codes are designed to be portable to any system supporting `MPI` and indeed have been run on many other systems. For the purposes of this exercise, it suffices to study the performance on this one machine since the buffering ideas are common to all `MPI` implementations even if the quantification of the effects, as well as the precise implementation of nonblocking communications may vary.

3 MPI point-to-point communication

In both solvers, the factorization algorithms almost only use `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

[3]Matrix ECL32 is included in the Rutherford-Boeing Collection
[4]Web page `http://www.parallab.uib.no/parasol/`

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 [11, 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 message might be copied directly into the matching receiver buffer, or it might be copied into a temporary system buffer. In the first case, the send call will not complete until a matching receive call occurs. In the second case, the send call may return ahead of the matching receive call, allowing a single-threaded process to continue with its computation. The `MPI` implementation may make either of these choices. It 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. The additional copying usually increases the message transfer overhead. However, in many asynchronous algorithms the effect of this overhead may be greatly reduced by overlapping the computation with the communication.

- *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.

Although with the short protocol, there is an extra cost associated with copying, if the algorithm is well designed, this cost can be offset by the overlapped computations. 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 processor) 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`.

Note that we refer here to `MPI` internal buffers. There can also be a buffer defined by the user 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.

MPT version	Year	default `MPI_BUFFER_MAX`
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 Cray T3E.

In this paragraph, we recall what is specific to a CRAY T3E implementation of `MPI` communications that will be relevant to the analysis of the performance of our sparse solvers. We focus on the differences between standard blocking communications (`mpi_send` and `mpi_recv`) and nonblocking communications (`mpi_isend` and `mpi_irecv`). On the CRAY T3E, only `MPI` receive buffers (no `MPI` send buffer) are used to implement the short protocol. If the message length is smaller than `MPI_BUFFER_MAX` (see default value in Table 2), then the sender writes directly in the `MPI` receive buffer of the destination process. Note that this will occur independently of the way the send is actually performed (`mpi_send` or `mpi_isend`) and the receiver performs the reception (`mpi_recv` or `mpi_irecv`). When the receive process actually issues a reception, only copying from the `MPI` buffer to the user space will be involved. For large messages (larger than `MPI_BUFFER_MAX`) then no `MPI` buffer is used. The communication of the effective data will only start when the receiver posts the receive instruction (`mpi_recv` or `mpi_irecv`). Note that with a threshold `MPI_BUFFER_MAX` set to "unlimited", we would expect communications based on standard sends and receives (`mpi_send` and `mpi_recv`) to perform very similarly to asynchronous immediate communications (`mpi_isend` and `mpi_irecv`). However, with immediate communications, performance should not depend on `MPI` buffering and, if `MPI_BUFFER_MAX` is set to 0, the memory associated with the `MPI` buffer and the copy from the `MPI` buffer to the user space could be avoided.

4 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 fully asynchronous and are based on 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 communication overlapping).

Although, in the context of `MUMPS`, the use of an immediate receive seems quite natural, we explain in Section 4.1, 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 avoid some possible added complications by restricting ourselves to a single `mpi_irecv` pending request. In Section 4.2, 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. We study the performance obtained on large symmetric and unsymmetric matrices and illustrate, using `vampir` traces, the gains obtained on one of our largest problems.

4.1 Introducing immediate receives during factorization

In this section, we describe the modifications required for our introduction of asynchronous immediate receives in `MUMPS` Version 4.0. For the sake of clarity, we first describe how we modify the reception of messages involved during dynamic scheduling. We then show how to modify this solution to handle all type of receptions involved in the `MUMPS` code.

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 user space. A Fortran module was designed for this purpose and is briefly described in [3]. On the destination process, the reception of the messages will be the key point for the synchronization and scheduling of the work. In fact, message reception can be invoked in the following three situations :

1. *Dynamic scheduling*:
   Blocking and non-blocking receives are used to drive the scheduling of the tasks on each process.

2. *Task ordering:*
   A process may have to receive and treat a "late" message to be able to finish its current task.

3. *Insufficient space in send buffer:*
   To avoid deadlock, the corresponding process tries to receive messages until space becomes available in its local send buffer.

In order to avoid the drawback of centralised 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. Each process then executes Algorithm 1.

Algorithm 1 *Dynamic scheduling*

```
While  not all nodes processed
         If  [ local pool empty ] Then
             blocking receive for a message; process the 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
```

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 parameter `blocking` indicates whether we want to wait for the arrival of a message or not.

The procedure *Try_to_receive_and_process_message* is described by Algorithm 3. We differentiate between the cases of a receive request pending and of the blocking wait for a message. Finally, we always receive all short messages related to dynamic scheduling (one integer holding the updated load of the other processes) that are ready to be received before reactivating an immediate receive request. This information is used so that tasks generated during the computation can dynamically be affected to the less loaded processes, where the load of a process is defined as the total number of operations ready to be performed on

Algorithm 2 *Dynamic scheduling with immediate receive*

```
While  not all nodes processed
       blocking = false
       If [ local pool empty ] blocking = true
       Try_to_receive_and_process_message ( blocking )
       If [ no message received and pool not empty ] Then
          extract work from the pool and process it
       End If
End While
```

this process. In fact, since a maximum of one `mpi_irecv` request is pending at a given time, part of the benefit of issuing `mpi_irecv` might be lost if one does not force, at this point of the algorithm, the reception of these short and trivial to process messages (in fact, they are really used to emulate an `mpi_put` in a portable way). If we take the example of a large message following a single dynamic scheduling message then, until the small message is processed, it is not possible to have the `mpi_irecv` active on the large message. The start of the `mpi_irecv` on the large message is thus postponed which might cause a delay in the sending process because of situation 3. A delay in the sender could then cascade causing a blocking receive because of a "late message" as in situation 2. An immediate receive request will thus be issued each time a message is received and processed.

Algorithm 3 *Try_to_receive_and_process_message (blocking)*

```
If [ Receive request pending ] Then
   If [ blocking ] Then
      Wait for the end of pending receive request (– mpi_wait –)
      Process message
   Else If [ Message in buffer (– mpi_test –) ] Then
           Process message
   End If
Else
   If [ blocking ] Then
      Blocking receive for any message ( – mpi_recv –)
      Process message
   Else If [ Message ready to be received ( – mpi_iprobe –) ] Then
           Receive message in buffer (– mpi_recv – )
           Process message
   End If
End If
If [ No receive request pending ] Then
   process all ready-to-be-received short messages related to the load of other processes
   Reactivate an immediate receive request (– mpi_irecv – )
End If
```

Actually, Algorithm 3 should also be designed to handle messages corresponding to situation 2 (task ordering). During task ordering, we need to perform a blocking receive on a so called "late message". Such cases are illustrated in [2, 3] and are due to the fact that, although the algorithm is asynchronous, we still have to maintain a partial order between the tasks. Our asynchronous algorithm has been designed so that, when a "late 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. Note that in Algorithm 3, the parameter `blocking` only specifies that we are blocked until the reception of any message.

The main difficulty introduced by the use of a blocking receive on a given message in Algorithm 3 is that a "wrong" message might already be in our receive buffer because of an asynchronous pending receive request. Algorithm 4 shows how we have modified Algorithm 3 to solve this problem. An additional parameter `LateMessage` has been introduced to characterise the expected message. Combined with `blocking` set to true, `LateMessage` indicates the type of message (process source and message label) that is expected. Setting `LateMessage` to "any message" will enable us to perform a blocking receive on any message as required by the dynamic scheduling Algorithm 2.

For the sake of clarity, two new local variables have been introduced in Algorithm 4. `MessRecv` indicates that a message has been received during the pending receive request. `RightMessage` is true when the message received is the expected one (that is has the same characteristics as `LateMessage`). Comments are in parentheses using small and slanted fonts. Note that, if `LateMessage` is true in a call to Algorithm 4, then `blocking` must also be true.

Between lines 5 and 8 of the algorithm we are, in the case mentioned before, 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 late message before processing the current message in the buffer. This will enable us to perform a non-blocking 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 be 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 4 might be called recursively. A receive can be issued during the processing of almost any message giving rise to a situation 2 (task ordering) or 3 (insufficient space in the send buffer). 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 31, we must then test whether activation of an immediate receive is authorised.

One final minor problem introduced by the use of immediate receive is that it must now be the responsibility of the sending process to decide if the receive buffer of the destination process is large enough to process it. In Version 4.0 of the code, the destination process always checked the size of the message to be received before receiving it. The maximum size of the receive buffer can only be *estimated* during the analysis phase because of the delay in selecting pivots caused by numerical pivoting for stability.

Using immediate receive, we explain in the following why one can expect better

Algorithm 4 *Try_to_receive_and_process_message(* blocking, LateMessage *)*

```
0. MessRecv=false; RightMessage=true
1. If [ Receive request pending ] Then
2.     If [ blocking ] Then
3.         Wait for the end of pending request; (– mpi_wait – )
4.         MessRecv=true ( – message is in buffer – )
5.         If [ The message received $\neq$ LateMessage ] Then
6.             RightMessage=false
7.             Blocking probe for expected message ( – mpi_probe – )
8.         End If
9.     Else If [ Message in buffer ] MessRecv=true
10.    End If
11.    If [ MessRecv ] Then
12.        If [ Not RightMessage ] Suppress activation of immediate receive
13.        Process the message already in buffer
14.        If [ Not RightMessage ] Then
15.            Re-authorise activation of immediate receive
16.            If [ LateMessage ready to be received (– mpi_iprobe – ) ] Then
17.                Receive and process it
18.            Else (– expected message is already received and processed – )
19.            End If
20.        End If
21.    End If
22. Else
23.    If [ blocking ] Then
24.        Blocking receive for any message ( – mpi_recv –)
25.        Process message
26.    Else If [ Message ready to be received ( – mpi_iprobe –) ] Then
27.            Receive message in buffer (– mpi_recv – )
28.            Process message
29.    End If
30. End If
31. If [ No receive request pending and Immediate receive authorised ] Then
32.    Receive and process all ready-to-be-received messages related to dynamic scheduling
33.    Reactivate an immediate receive request (– mpi_irecv – )
34. End If
```

overlapping of communication and computation. In this context, a message is said to be large if it is larger than `MPI_BUFFER_MAX`. For large messages, we see two reasons for obtaining an improvement. (For short messages we do not expect much improvement.) First, with no immediate receive, if a large message is to be received then the message might actually finish being transferred/sent only when the receiver actually performs the reception. Second, using immediate receive, the space in the send buffer becomes free earlier. Less idle time in the sending process, as in situation 3, might be expected if the send buffer is not saturated.

4.2 Performance analysis

Matrix	Ordering	`mpi_irecv`	Number of processors				
			4	8	16	32	64
BMWCRA_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 our largest test matrices we show, in Table 4, the impact of using immediate receive during the factorization phase. `MUMPS` with the same tuning of machine dependent parameters has been used to get all the results (with and without immediate receive) reported in Table 4. 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. Node parallelism involves a relatively larger number of messages on symmetric matrices than on unsymmetric matrices that might saturate more the send buffer and the internal `MPI` buffers.

In Table 5, we show the maximum size of the messages and the average volume of

Matrix	Ordering	Number of processors				
		4	8	16	32	64
Maximum message size						
BMWCRA_1	MeTiS	—	—	7.3	2.7	2.2
BMW3_2	MeTiS	—	10.0	2.2	1.2	1.3
CRANKSG2	MeTiS	—	7.0	4.0	2.1	1.6
SHIP_003	MeTiS	—	—	5.6	2.6	2.1
BBMAT	AMD	4.8	3.0	2.6	2.5	2.4
ECL32	AMD	12.9	6.1	3.4	3.4	3.2
INVEXTR1	AMD	7.2	5.1	3.4	2.8	2.5
MIXTANK	AMD	16.0	7.3	4.3	3.9	3.9
Average Communication volume						
BMWCRA_1	MeTiS	—	—	34	33	18
BMW3_2	MeTiS	—	25	25	17	9
CRANKSG2	MeTiS	—	31	30	23	18
SHIP_003	MeTiS	—	—	65	60	34
BBMAT	AMD	35	35	33	32	19
ECL32	AMD	64	63	56	45	26
INVEXTR1	AMD	60	43	33	28	15
MIXTANK	AMD	77	115	109	93	51

Table 5: Maximum message size (in Mbytes) and average volume of communication per processor (in Mbytes) during factorization. – means not enough memory.

communication. One can see that, because of node level parallelism, the maximum size of the messages generally decreases when increasing the number of processors [1]. It explains why, for a fixed problem, larger relative gains are obtained in Table 4 on a smaller number of processors. We also see that the total volume of messages can also be a good indicator of the gain that can be expected from the use of immediate receives.

To further analyse the gain due to the use of immediate receives, we show in Figures 1 and 2 the execution traces for the factorization of matrix CRANKSG2 (using 8 processors of the CRAY T3E). Messages have been suppressed to see better the proportion of execution time used by `MPI` communications. One can see that `MPI` takes significantly more time when immediate receive is off than when it is on. The summary chart of the same traces in Figure 3 shows that using immediate receives reduces the time spent in `MPI` calls by almost a factor of three.

To conclude this study, we show (compare the results in Tables 3 and 6), as one might expect from the previous discussion, 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`).

Figure 1: Immediate receive OFF; Trace of the factorization phase of matrix CRANKSG2 using 8 processors of the CRAY T3E with `MPI_BUFFER_MAX` set to 4Kbytes. Black areas correspond to the time spent in `MPI`.

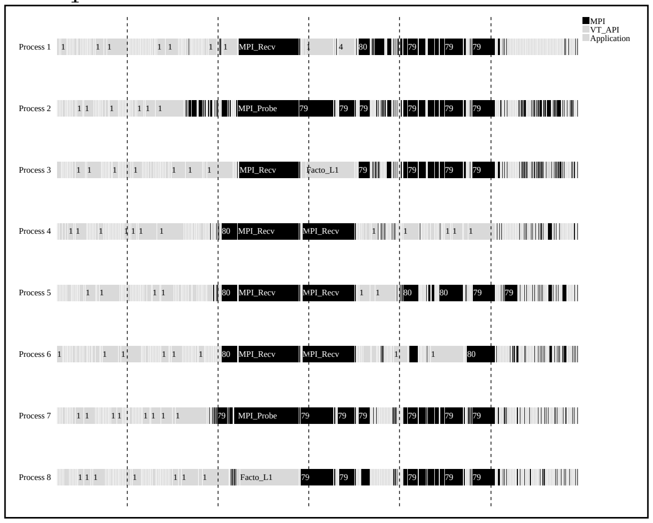


Figure 2: Immediate receive ON; Trace of the factorization phase of matrix CRANKSG2 using 8 processors of the CRAY T3E with `MPI_BUFFER_MAX` set to 4Kbytes. Black areas correspond to the time spent in `MPI`.

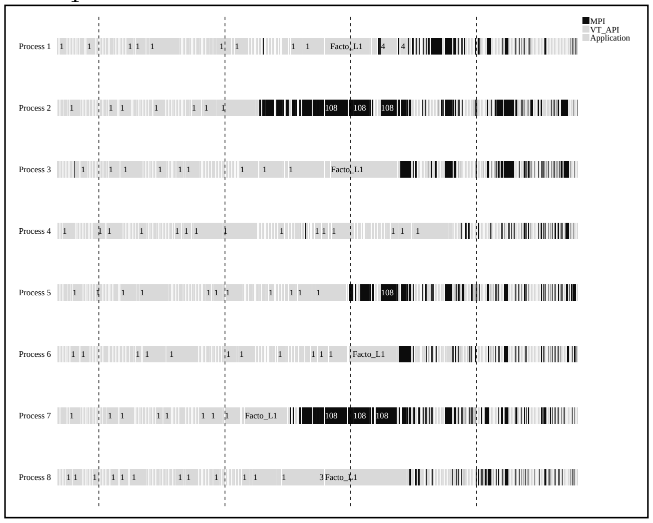
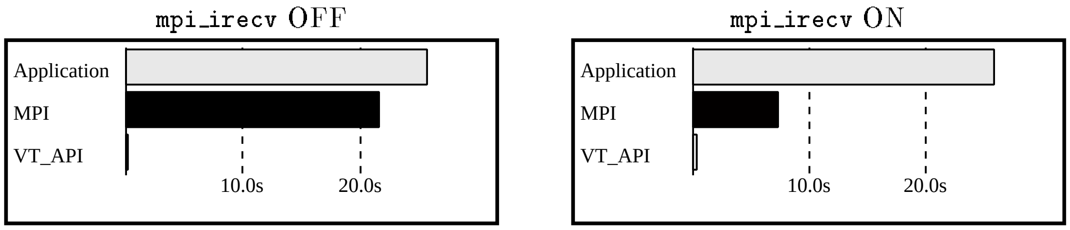


Figure 3: Summary chart on the use of immediate receive during the `MUMPS` factorization phase (Matrix CRANKSG2, ND ordering, 8 processors with `MPI_BUFFER_MAX` set to 4Kbytes).



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 6: 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.

5 MPI tuning for SuperLU

5.1 Static partition

In order to understand the parallel factorization algorithm used in `SuperLU`, we first explain how we partition the matrix into blocks of submatrices, and how the blocks are assigned to the processes.

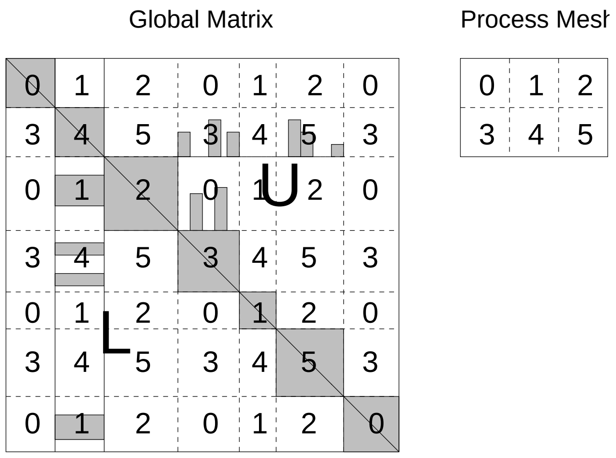
Our matrix partitioning is based on the notion of an *unsymmetric supernode* introduced in [5]. The supernode is defined over the matrix factor L. A supernode is a range $(r : s)$ of columns of L with the triangular block just below the diagonal being full, and the same nonzero structure elsewhere (this 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 non-uniform size. Figure 4 illustrates such a block partition. The off-diagonal blocks may be rectangular and may not be full. Furthermore, the columns in a block of U do not necessarily have the same row structure. We call a dense subvector in a block of U a *segment*. The P processes are also arranged as a 2D mesh of dimension $P_r \times P_c = P$. By block-cyclic layout, we mean 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. During the factorization, block $L(I, J)$ is only needed by the processes on the process row $((I-1) \bmod P_r)$. Similarly, block $U(I, J)$ is only needed by the processes on the process column $((J-1) \bmod P_c)$. This partitioning and mapping can be controlled by the user. First, the user can set the *maximum block size* parameter. The symbolic factorization algorithm identifies supernodes, and chops the large supernodes into smaller ones if their sizes exceed this parameter. The supernodes may be smaller than this parameter due to sparsity and the blocks are then formed along the supernode boundaries. Second, the user can set the shape of the process grid, such as 2×3 or 3×2. Better performance is obtained when we keep the process row dimension slightly smaller than the process column dimension.

5.2 Pipelining and nonblocking send and receive

In this subsection, we first describe in detail how the parallel factorization algorithm utilizes the pipeline effect. Then we discuss how to improve the performance robustness by introducing immediate sends and receives. The following notation will be used in Figures 5 and 6, and throughout the discussion. Matlab notation is used for integer ranges and submatrices.

- Process IDs

Figure 4: The 2D block-cyclic layout and the data structure used in `SuperLU`.



- $PROC_c(K)$: the set of column processes that own block column K
  For example, in Figure 4, $PROC_c(3) = PROC_c(6) = \{2, 5\}$.
- $PROC_r(K)$: the set of row processes that own block row K
  For example, in Figure 4, $PROC_r(1) = PROC_r(3) = \{0, 1, 2\}$.
- P_{K+1} : the process in $PROC_c(K+1) \cap PROC_r(K+1)$
- me : the process rank as illustrated in Figure 4

• Tasks labelled in Figure 6

- F (...) : Factorize a block column or a block row[5]
- S (...) : Send a block column or a block row
- R (...) : Receive a block column or a block row
- U$^{(k)}$(...) : Update the trailing submatrix using $L(:, K)$ and $U(K, :)$

Figure 5 outlines the parallel sparse LU factorization algorithm. 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 $U(K, K+1 : N)$. 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, where b is the block size of the K-th block column/row.

For ease of understanding, the algorithm presented in Figure 5 has been simplified. 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 $PROC_c(K+1)$. Figure 6 illustrates this idea using Steps K and $K+1$ of the algorithm. In the figure, we show the activities of the four process

[5]There is also communication involved in this task, but it is negligible, and so is omitted in the discussion.

groups along the time line. The path marked with the dashed line represents the critical path, that is, the parallel runtime could be reduced only if the critical path is shortened. The block factorization tasks "F (...)" are usually on the critical path, whereas the update tasks "U (...)" are often overlapped with the other tasks. There is lack of parallelism for the "F (...)" tasks in Steps (1) and (2), because only one set of column processes or row processes participate in these tasks. This pipelining 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.

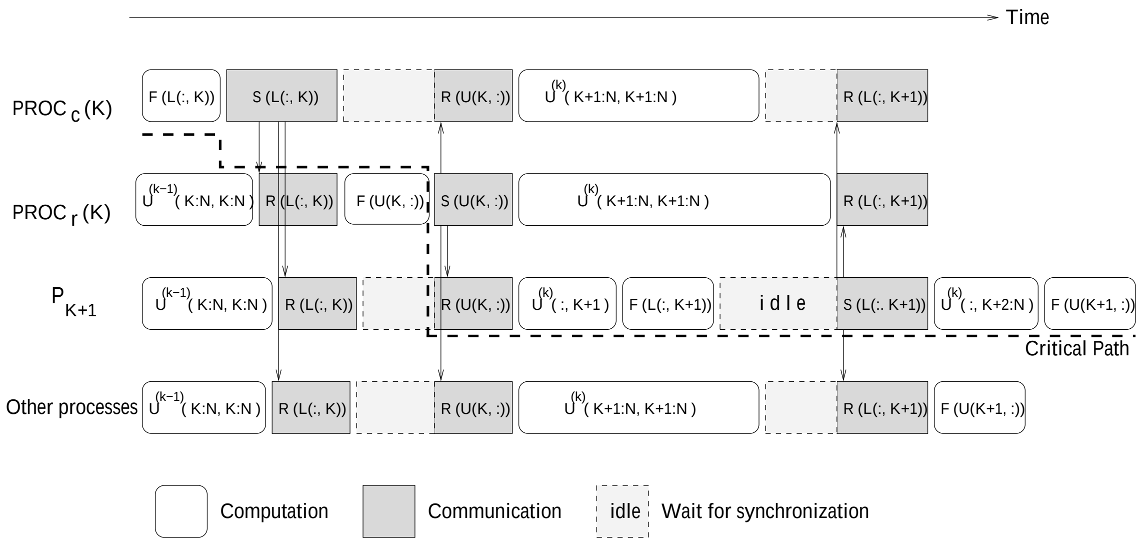
Figure 5: Outline of the parallel factorization algorithm used in `SuperLU`.

```
for block $K = 1$ to $N$ do
    (1) if [ $me \in PROC_c(K)$ ] then
            Factorize block column $L(K : N, K)$
            Send $L(K : N, K)$ to the processes in my row who need it
        else
            Receive $L(K : N, K)$ from one process in $PROC_c(K)$ (if I need it)
        endif
    (2) if [ $me \in PROC_r(K)$ ] then
            Factorize block row $U(K, K+1 : N)$
            Send $U(K, K+1 : N)$ to processes in my column who need it
        else
            Receive $U(K, K+1 : N)$ from one process in $PROC_r(K)$ (if I need it)
        endif
    (3) for $J = K + 1$ to $N$ do
            for $I = K + 1$ to $N$ do
                if [ $me \in PROC_r(I)$ and $me \in PROC_c(J)$
                    and $L(I, K) \neq 0$ and $U(K, J) \neq 0$ ] then
                    Update trailing submatrix $A(I, J) \leftarrow A(I, J) - L(I, K) \cdot U(K, J)$
                endif
            end for
        end for
end for
```

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 "S (...)" and "R (...)". In Figure 6, we see idle time (longer send) during the sending of "S ($L(:, K+1)$)" 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 *long protocol.* That is, process P_{K+1} posts `mpi_send` long before processes $PROC_r(K)$ post the matching `mpi_recv`, and the sender must be blocked to wait for `mpi_recv`.

We have observed big differences in performance between setting `MPI_BUFFER_MAX` to "unlimited" and to 4 Kbytes. Here, "unlimited" means that all messages use the short protocol. Table 7 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 table shows that the most dramatic difference is on 2 processors, where the smaller buffer results in 74% speed loss. This

Figure 6: Illustration of the pipeline at Steps K and $K+1$ during the SuperLU factorization.



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.

For comparison, in Table 8, we also give the timings of the improved code after introducing mpi_isend and mpi_irecv. The detailed algorithmic change will be described in the following. We see that the performance of the new code is not constrained by the MPI buffer size. In fact, the performance is comparable with MPI_BUFFER_MAX set to either 4 KB or "unlimited". Compared to Table 7, the performance is also comparable to the old code using the value "unlimited".

Nprocs	1	2	4	8	16	32	64	128
Grid size	29	33	36	41	46	51	57	64
<u>MPI_BUFFER_MAX</u>								
unlimited	57.0	62.3	53.3	61.5	62.7	65.7	76.2	80.7
4 Kbytes	57.9	108.2	92.4	102.5	104.2	101.6	119.3	111.0

Table 7: SuperLU factorization time in seconds 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 primitive, mpi_isend and mpi_irecv as follows. 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. This could eliminate the idle time during the send "S $(L(:, K+1))$" shown in Figure 6.

- For the receiver, we will post mpi_irecv much earlier than we actually need the data.

Nprocs	1	2	4	8	16	32	64	128
Grid size	29	33	36	41	46	51	57	64
<u>`MPI_BUFFER_MAX`</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 8: `SuperLU` factorization time in seconds for the cubic grid problems with `MPI_BUFFER_MAX` set to unlimited and to 4Kbytes. MPT version 1.3.0.4 is used. `mpi_isend` and `mpi_irecv` are used.

> For example, for processes $PROC_r(K)$ in Figure 6, we could post "R $(L(:, K+1))$" before "U $(A(K+1:N, K+1:N))$". 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 idea, 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. Figure 7 sketches the modified pipelining algorithm using `mpi_isend` and `mpi_irecv`. Comparing the loop bodies of Figures 5 and 7, we see that Step (2) remains the same. The main difference 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 $(K+1)$-th block column, then immediately factorize this column and post send and receive of the factorized column for the $(K+1)$-th 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 9, we illustrate 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). Table 10 shows the maximum size of all the messages during the factorization. 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 decreasing, 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 faster than the old code.

6 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 the algorithmic changes necessary in two sparse direct codes 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.

Our application of solving large sparse systems is a significant one and inter alia we

Figure 7: 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 [ $K+1 \le N$ ] then
             /* — Factor-ahead scheme — */
20.          if [ $me \in PROC_c(K+1)$ ] then
21.              Update $(K+1)$-th column $A(:,K+1) \leftarrow A(:,K+1) - L(:,K) \cdot U(K,K+1)$
22.              Factorize block column $L(:,K+1)$
23.              Post send $L(:,K+1)$ to the processes in my row who need it (– mpi_isend – )
24.          else
25.              Post receive $L(:,K+1)$ from one process in $PROC_c(K+1)$ (– mpi_irecv – )
26.          endif
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 $L(I,K) \ne 0$ and $U(K,J) \ne 0$ ] 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
```

Matrix	Ordering	`mpi_isend/`	Number of processors				
		`mpi_irecv`	4	8	16	32	64
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 9: Influence of the use of `mpi_isend/mpi_irecv` on the time (in seconds) for factorization of `SuperLU`. MPT version 1.4.0.0 (latest) is used, where the default `MPI_BUFFER_MAX` is 0 Byte.

Matrix	Ordering	Number of processors				
		4	8	16	32	64
BBMAT	AMD	0.19	0.18	0.09	0.09	0.05
ECL32	AMD	0.32	0.32	0.16	0.16	0.09
INVEXTR1	AMD	0.24	0.24	0.12	0.12	0.07
MIXTANK	AMD	0.32	0.33	0.17	0.16	0.09

Table 10: Maximum message size (in Mbytes) during `SuperLU` factorization.

show 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.*, pages 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*, 2001. to appear.

[5] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph 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, Toulouse.

[7] J. R. Gilbert and J. W. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.

[8] HSL. A collection of Fortran codes for large scale scientific computation, 2000. http://www.cse.clrc.ac.uk/Activity/HSL.

[9] X. S. Li and J. W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24, 1999.

[10] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[11] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.