Performance and tuning of two distributed memory sparse solvers

Patrick R. Amestoy\textsuperscript{2}, Iain S. Duff\textsuperscript{3}, Jean-Yves L'Excellent\textsuperscript{4}, and Xiaoye S. Li\textsuperscript{5}

Technical Report TR/PA/00/91


CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT
We examine the performance of two codes using direct methods to solve large sparse linear equations on distributed memory computers. The code SuperLU is a right-looking supernodal code, written in C, while MUMPS is written in Fortran 90 and uses a multifrontal technique. Both codes use MPI for message-passing.

In this short paper, we consider the influence of the MPI buffer size and send and receive primitives on both codes and compare the performance of the factorization and solution phases of both codes on test examples from the PARASOL Project on a CRAY T3E-900.

Keywords: sparse linear systems, distributed memory codes, multifrontal, supernodal, direct methods, comparison of codes.

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

\textsuperscript{1}Current reports available at http://www.cerfacs.fr/algor/algorreports.html. Also appeared as Report RAL-TR-2001-004 from Rutherford Appleton Laboratory. The project was supported by the France-Berkeley Fund.

\textsuperscript{2}amestoy@enseeiht.fr. ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse, France. Much of the work done while a visitor at NERSC.

\textsuperscript{3}duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England.

\textsuperscript{4}excelle@enseeiht.fr. NAG Ltd, Wilkinson House, Oxford OX2 8DR, England.

\textsuperscript{5}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 .................................................. 1

2 Algorithmic improvements and tuning .................. 3
   2.1 Impact of the MPI internal buffer size on the performance of our solvers .. 3
   2.2 Using asynchronous immediate receives to improve the performance .... 4

3 Performance analysis ........................................ 5

4 Concluding remarks .......................................... 8
1 Introduction

This work was performed in the context of a France-Berkeley funded project between NERSC located in Berkeley (USA) and CERFACS-ENSEEIHT located in Toulouse (France). We consider the direct solution of sparse linear equations on distributed memory computers using two codes, MUMPS (Amestoy, Duff, L’Excellent and Koster 1999, Amestoy, Duff and L’Excellent 2000a) and SuperLU (Li and Demmel 1999) The first uses a multifrontal approach with dynamic pivoting for stability while the second is based on a supernodal technique with static pivoting. We discuss both the tuning and performance analysis of the two sparse solvers on the 512 processor Cray T3E from NERSC, Lawrence Berkeley National Laboratory.

Both approaches can be described by a computational tree whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, MUMPS, at each node some steps of Gaussian elimination are performed on a dense frontal matrix and the Schur complement 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 corresponding to a node of the tree, then immediately sends the data to update the block columns corresponding to ancestors in the tree. In both approaches, a pivot order is defined by the analysis and symbolic factorization stages. In the case of MUMPS, the modulus of the prospective pivot is compared to the largest modulus of an entry in the row and it is only accepted if this is greater than a threshold value, typically a value between 0.001 and 0.1 (our default value is 0.01). Note that, even though MUMPS can choose pivots from off the diagonal, the largest entry in the column might not be available for pivoting at this stage because all entries in its row may not be fully summed. If a prospective pivot fails the test and cannot be used within the partial factorization at a node, all that happens is that it is kept in the Schur complement and is passed to the parent node. Eventually all of the columns will be available for pivoting, at the root if not before, so that a pivot can be chosen from that column. Thus the numerical factorization can respect the threshold criterion but at a cost of increasing the size of the frontal matrices and potentially causing more work and fill-in than were forecast. For the SuperLU approach, a static pivoting strategy is used and we keep to the pivotal sequence chosen in the analysis. The magnitude of the potential pivot is tested against a threshold of $e^{1/2}||A||$, where $e$ is the machine precision and $||A||$ is the 1-norm of $A$. If it is less than this value it is immediately set to this value (with the same sign) and the modified entry is used as pivot. This corresponds to a half-precision perturbation to the original matrix. The result is that the factor is not exact and iterative refinement may then be needed. Note that, after iterative refinement, we obtain an accurate solution in all the cases that we tested. If problems were still to occur then extended precision BLAS could be used.

For both solvers it can be very beneficial (in different ways) to precede the ordering by performing an unsymmetric permutation to place large entries on the diagonal and then scaling the matrix so that the diagonals are all of modulus one and the off-diagonals have modulus less than or equal to one. We use the MC64 (Duff and Koster 1999) code of HSL (HSL 2000) to perform this preordering and scaling (Amestoy, Duff, L’Excellent and Li 2000b).

Both approaches use Level 3 BLAS to perform the elimination operations. However, in MUMPS the frontal matrices are always square. It is shown in Amestoy and Puglisi (2000) how one can detect and exploit sparsity within the frontal matrices but the present implementation takes no advantage of this sparsity and all the counts measured assume the frontal matrix is dense. In
SuperLU, advantage is taken of sparsity in the blocks and usually the dense matrix blocks are smaller than those used in MUMPS. In addition, SuperLU uses a more sophisticated data structure to keep track of the irregularity in sparsity.

The parallelism within MUMPS is at two levels. The first uses the structure of the assembly tree, exploiting the fact that nodes which are not ancestors or descendants are independent. The initial parallelism from this source (tree parallelism) is the number of leaf nodes but this reduces to one at the root. The second level is in the subdivision of the elimination operations through blocking of the frontal matrix. This blocking, giving rise to node parallelism, is either one-dimensional or two-dimensional (at the root node only). During the analysis phase each tree node is statically assigned a processor a priori. The subassignment of blocks of the frontal matrix is then done dynamically during factorization. SuperLU also uses two levels of parallelism although more advantage is taken of the node parallelism through blocking of the supernodes. Because the pivotal order is fully determined at the analysis phase, the assignment of blocks to processors can be done statically a priori before the factorization commences. A 2D block-cyclic layout is used and the execution can be pipelined since the sequence is predetermined.

<table>
<thead>
<tr>
<th>Real Unsymmetric Assembled (RUA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix name</td>
</tr>
<tr>
<td>BBMAT</td>
</tr>
<tr>
<td>ECL32</td>
</tr>
<tr>
<td>INVESTR1</td>
</tr>
<tr>
<td>MIXTANK</td>
</tr>
<tr>
<td>TWOTONE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Real Symmetric Assembled (RSA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix name</td>
</tr>
<tr>
<td>CRANKSG2</td>
</tr>
</tbody>
</table>

Table 1.1: Test matrices. \(^{(*)}\) StrSym is the number of nonzeros matched by nonzeros in symmetric locations divided by the total number of entries (that is, a symmetric matrix has value 1.0).

Our test matrices come from the forthcoming Rutherford-Boeing Sparse Matrix Collection\(^1\) (Duff, Grimes and Lewis 1997), the industrial partners of the PARASOL Project\(^2\), and the EECS Department of UC Berkeley\(^3\). Since SuperLU cannot exploit symmetry, most of our test matrices are unsymmetric. The relatively large symmetric matrix CRANKSG2 and the symmetric version of MUMPS will be used only in Section 2 to illustrate the impact of the size of the MPI buffers on the performance of MUMPS.

We first describe in Section 2 how we have tuned and improved our algorithms. We then compare in Section 3 the performance of the two solvers on the Cray T3E-900 (512 DEC EV5 processors, 256 Mbytes of memory per processor, 900 peak Megaflop rate per processor) from NERSC on a set of large unsymmetric matrices from real applications. We present some concluding remarks in Section 4.

---

\(^1\) Web page http://www.cse.clrc.ac.uk/Activity/SparseMatrices/
\(^2\) EU ESPRIT IV LTR Project 20160. Matrices are on Web page http://www.parallab.uib.no/parasol/
\(^3\) Matrix ECL32 included in the Rutherford-Boeing Collection
2 Algorithmic improvements and tuning

Porting a code to a new platform always provides a good opportunity for improving its performance not only on the target platform but also on previously implemented environments. For both solvers, the first phase in the optimisation on a new platform consists in adjusting a set of machine dependent parameters to fit the target machine. These parameters are used to balance the parallel machine’s speed of computation and communication, and the algorithm’s degree of parallelism. In the case of MUMPS, the porting of the code to the 512 processor CRAY T3E-900 gave us the opportunity to study the behaviour of the code on a larger number of processors than used in our previous work (Amestoy et al. 1999, Amestoy et al. 2000a). From our set of machine dependent parameters we chose appropriate parameters to address this issue. Other algorithmic modifications were motivated by having more processors available to us than formerly. The dynamic scheduling approach used in MUMPS was modified (see Amestoy et al. (2000b)) to better control the dynamic distribution of the tasks to the processors. When porting SuperLU to an IBM SP2, we found that there is a rather big performance gap between Level 2.5 and Level 3 BLAS. This motivated us to refine the numerical kernel to always use Level 3 BLAS. Depending on the matrices, this Level 3 BLAS kernel improves the uniprocessor factorization speed by about 20% to 40% on an IBM SP2. A performance gain was also observed on the Cray T3E. It is clear that the extra operations are well offset by the benefit of the more efficient Level 3 BLAS routines.

Furthermore, even on the same machine, we found that MPI programming environment changes (for example the default internal buffer size) may result in a dramatic performance difference. This enabled us to identify possible enhancements to the SuperLU code to make its performance more robust. We further discuss this issue in the next two subsections.

2.1 Impact of the MPI internal buffer size on the performance of our solvers

Currently the message transfer in SuperLU is performed using MPI standard send and receive operations, mpi_send and mpi_recv. 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 communication may be overlapped with computation. This is exactly what we observed from the SuperLU performance.

- **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 determined by the size of the MPI system buffer. For example, on the Cray T3E, the user may determine the size of the system buffer by setting an environment variable **MPI_BUFFER_MAX**. If a message length exceeds this value, the long protocol
will be used. On the Cray T3E, the current default value of the MPI internal buffer is 4 Kbytes and in an earlier MPI implementation (Release 1.3.0.3) the buffer size was unlimited.

<table>
<thead>
<tr>
<th>$N_{\text{procs}}$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>29</td>
<td>33</td>
<td>36</td>
<td>41</td>
<td>46</td>
<td>51</td>
<td>57</td>
<td>64</td>
</tr>
<tr>
<td>unlimited</td>
<td>57.0</td>
<td>62.3</td>
<td>53.3</td>
<td>61.5</td>
<td>62.7</td>
<td>65.7</td>
<td>76.1</td>
<td>80.7</td>
</tr>
<tr>
<td>4 Kbytes</td>
<td>57.0</td>
<td>108.2</td>
<td>92.4</td>
<td>102.5</td>
<td>104.2</td>
<td>101.6</td>
<td>119.3</td>
<td>111.0</td>
</tr>
</tbody>
</table>

Table 2.1: SuperLU factorization time (seconds). Cubic grid problems of dimension $N \times N \times N$ (11-point discretization of the Laplacian operator). $\text{MPI\_BUFFER\_MAX}$ is set to unlimited and to 4Kbytes. ($\text{mpi\_recv}$ is used to match $\text{mpi\_send}$.)

We illustrate, in Tables 2.1 and 2.2, the impact of the size of the MPI buffer on the performance of our codes. One first sees that, in both cases, the size of the MPI buffer strongly influences the factorization time. Secondly, with the default size of the MPI buffer (4 Kbytes), the use of a standard receive ($\text{mpi\_recv}$) to match a send ($\text{mpi\_send}$ or even $\text{mpi\_isend}$) does not lead to a good overlapping of communication with computation.

<table>
<thead>
<tr>
<th>Size (in Bytes) of the MPI buffer</th>
<th>0</th>
<th>128</th>
<th>512</th>
<th>1K</th>
<th>4K</th>
<th>64K</th>
<th>512K</th>
<th>2Mega</th>
<th>8Mega</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>37.7</td>
<td>37.0</td>
<td>37.4</td>
<td>38.3</td>
<td>37.6</td>
<td>32.8</td>
<td>28.3</td>
<td>26.4</td>
<td>26.4</td>
</tr>
</tbody>
</table>

Table 2.2: MUMPS factorization time (seconds). Matrix cranks4 on 8 processors of the CRAy T3E. ($\text{mpi\_recv}$ is used to match $\text{mpi\_isend}$). Number of operations is $4.2 \times 10^{10}$.)

2.2 Using asynchronous immediate receives to improve the performance

It is somewhat unpleasant that the performance of our codes depend on the MPI system buffer size. However, if we can match immediate sends ($\text{mpi\_isend}$) with immediate receives ($\text{mpi\_irecv}$) we can hope to address both issues (that is, independence with respect to MPI buffer size and communication overlapping).

In the MUMPS solver, the communications are fully asynchronous and are based on an immediate send ($\text{mpi\_isend}$). 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 ($\text{mpi\_recv}$). A problem with this mechanism occurs when messages are much larger than the MPI internal buffer size. 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 and provides user space for the communication to proceed. This is independent of the type of send used ($\text{mpi\_send}$ in SuperLU or $\text{mpi\_isend}$ in MUMPS). This can very significantly affect the potential algorithmic overlapping between computation and communication. However, if we can use an immediate receive ($\text{mpi\_irecv}$), which can be interpreted as having a separate “spawned” process implementing the reception, the reception can proceed in parallel with the process that issued
the \texttt{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 \texttt{mpi\_irecv}) at the time when we test for
the availability of the message. Note that by doing so we have also overlapped the copying from
the MPI buffer to the user space.

Although, in the context of \texttt{MUMPS}, the use of an immediate receive seems quite natural,
we explain in Amestoy et al. (2000b) why it has required more algorithmic developments than
might have been expected. The main issue with using an immediate receive in our asynchronous
algorithmic context is that we cannot tell \textit{a priori} which message we are receiving. That is,
the \texttt{mpi\_irecv} request must be sent to receive any type of message from any source. In our
implementation, we have avoided some possible added complications by restricting ourselves to
a single \texttt{mpi\_irecv} pending request. We show (compare the results in Tables 2.2 and 2.3) that,

<table>
<thead>
<tr>
<th>Size (in Bytes) of the MPI buffer</th>
<th>0</th>
<th>128</th>
<th>512</th>
<th>1K</th>
<th>4K</th>
<th>64K</th>
<th>512K</th>
<th>2Mega</th>
<th>8Mega</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>27.1</td>
<td>27.3</td>
<td>26.5</td>
<td>26.6</td>
<td>26.4</td>
<td>26.2</td>
<td>26.2</td>
<td>26.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: \texttt{MUMPS} factorization time (in seconds) of matrix \texttt{CRANKSG2} using 8
processors of the CRAY T3E. \texttt{mpi\_irecv} is used to match \texttt{mpi\_isend}.

as one might expect, the new code based on immediate receives (\texttt{mpi\_irecv}) is very much less
sensitive to the size of the internal MPI buffer than the initial version based on standard receives
(\texttt{mpi\_recv}).

For the \texttt{SuperLU} solver, we plan to change the communications pattern used so that the code
performance is less dependent on the underlying MPI implementation and is more portable.
Although this idea might not be hard to implement, we need to provide an extra buffer on the
receiving process to take care of one outstanding message. We still have to experiment with
the new scheme and see how sensitive the performance is to the size of the system buffer. We
plan to report on this later.

3 Performance analysis

In this section, we compare the performance and study the behaviour of the numerical phases
(factorization and solve) of the two solvers.

For the sake of clarity, we will only report results with the best (in terms of factorization
time) sparsity ordering for each approach. Both a minimum degree ordering (\texttt{AMD}) (Amestoy,
Davis and Duff 1996) based on \texttt{MC47} from HSL (HSL 2000) and a nested dissection (\texttt{ND}) ordering
are considered. (Sometimes we use the \texttt{ON-MES} ordering from \texttt{MES} (Karypis and Kumar
1998), and sometimes the nested dissection/\texttt{haloamd} ordering from \texttt{SCOTCH} (Pellegrini, Roman
and Amestoy 1999) depending on which performs better on each particular problem.) For the
matrix \texttt{TWOTONE} it is very beneficial to precede the ordering by an unsymmetric permutation
to place large entries on the diagonal. We use the \texttt{MC64} code of HSL to perform this preordering
and scaling (Duff and Koster 1999). When the best ordering for \texttt{MUMPS} is different from that for
\texttt{SuperLU}, results with both orderings are provided.

We see that \texttt{MUMPS} is usually faster than \texttt{SuperLU} and is significantly so on a small number of
processors. We believe there are two reasons. First, \texttt{MUMPS} handles symmetric and more regular
data structures better than SuperLU, because MUMPS uses Level 3 BLAS kernels on bigger blocks than those used within SuperLU. As a result, the Megaflop rate of MUMPS on one processor is on average about twice that of the SuperLU factorization. Note that, even on the matrix twotone, for which SuperLU performs three times fewer operations than MUMPS (see column 3), MUMPS is over 2.5 times faster than SuperLU on four processors. On a small number of processors, we also notice that SuperLU does not always fully benefit from the reduction in the number of operations due to the use of a nested dissection ordering (see BBMAT with SuperLU using 4 processors).

We see that the ordering very significantly influences the performance of the codes (see results with matrices BBMAT and ECL32). In particular, MUMPS generally outperforms SuperLU when nested dissection ordering is used even on a large number of processors. On the other hand, if we use the minimum degree ordering, SuperLU is generally faster than MUMPS on a large number of processors. We also see that, on most of our unsymmetric problems, neither solver provides enough parallelism to benefit from using more than 128 processors. The only exception is matrix ECL32 using the AMD ordering (requiring $64 \times 10^9$ flops for the factorization), for which SuperLU continues to decrease the factorization time up to 512 processors. Our lack of other large unsymmetric systems gives us few data points in this regime but one might expect that, independently of the ordering, the 2D distribution used in SuperLU should provide better scalability (and hence eventually better performance) on a large number of processors than the mixed 1D and 2D distribution used in MUMPS. To further analyse the scalability of our solvers we have also reported in Amestoy et al. (2000a) results obtained on large three dimensional regular grid problems.

We now focus on the time spent to obtain the solution. We apply enough steps of iterative refinement to ensure that the componentwise relative backward error ($Berr$) is less than $\sqrt{e} = 1.48 \times 10^{-8}$. Each iterative refinement involves not only a forward and a backward solve but also a matrix-vector product with the original matrix. With MUMPS, the user can provide the input matrix in a very general distributed format (Amestoy et al. 1999). This functionality was used

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Ord.</th>
<th>Flops $\times 10^9$</th>
<th>Solver</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBMAT</td>
<td>AMD</td>
<td>41.5</td>
<td>MUMPS</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>4.0</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.5</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.5</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td>ECL32</td>
<td>AMD</td>
<td>64.6</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>64.6</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.9</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20.7</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td>INVESTR</td>
<td>ND</td>
<td>8.1</td>
<td>MUMPS</td>
<td>31.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.9</td>
<td>SuperLU</td>
<td>68.2</td>
</tr>
<tr>
<td>MIXTANK</td>
<td>ND</td>
<td>13.2</td>
<td>MUMPS</td>
<td>40.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.9</td>
<td>SuperLU</td>
<td>88.1</td>
</tr>
<tr>
<td>TWOTONE</td>
<td>MC64</td>
<td>29.3</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>+AMD</td>
<td>8.0</td>
<td>SuperLU</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.1: Factorization time (in seconds). “—” indicates not enough memory. Flops corresponds to the number of operations involved during factorization.
to parallelize the matrix-vector products. With SuperLU, the parallelization of the matrix-vector product was easier because the input matrix is duplicated on all the processors.

In Table 3.2, we report both the time to perform one step of solution (use factorized matrix to solve $Ax = b$ and when necessary ($Berr$ greater than $\sqrt{\varepsilon}$) the time to improve the solution using iterative refinement (lines with “+ IR”). With SuperLU, one step of iterative refinement was always enough to reduce the backward error to $\sqrt{\varepsilon}$. With MUMPS, iterative refinement was only required on the matrix invext1 and the backward error was already so close to $\sqrt{\varepsilon}$ that on 4 and 8 processors iterative refinement was not required. We first observe that, on a small number of processors (less than 8), the solve phase is almost two orders of magnitude less costly than the factorization. On a large number of processors, because our solve phases are relatively less scalable than the factorization phases, the difference drops to one order of magnitude.

The performance reported in Table 3.2 shows that the regularity in the structure of the matrix factors generated by the factorization phase of MUMPS generally leads to a faster solve phase than that of SuperLU for up to 256 processors. On 512 processors, the solve phase of SuperLU is sometimes faster than that of MUMPS. The cost of iterative refinement can significantly increase the cost of obtaining a solution. With SuperLU, because of static pivoting, it is more likely that iterative refinement will be required to obtain an accurate solution on numerically difficult matrices. With MUMPS, the use of partial pivoting during the factorization reduces the number of matrices for which iterative refinement is required.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Order</th>
<th>Solver</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBMAT</td>
<td>AMD</td>
<td>MUMPS</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>— + (IR)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— + (IR)</td>
<td>—</td>
</tr>
<tr>
<td>ECL32</td>
<td>AMD</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td>INVEXT1</td>
<td>ND</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— + (IR)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— + (IR)</td>
<td>—</td>
</tr>
<tr>
<td>MIXTANK</td>
<td>ND</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td>TWOTONE</td>
<td>HC64</td>
<td>MUMPS</td>
<td>—</td>
</tr>
<tr>
<td>+AMD</td>
<td></td>
<td>SuperLU</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ND</td>
<td>— + (IR)</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.2: Solve time (in seconds). “— + (IR)” shows the time spent improving the initial solution using iterative refinement. “—” indicates not enough memory.
4 Concluding remarks

We have described some algorithmic tuning and performance analysis of two state-of-the-art distributed sparse direct solvers, SuperLU and MUMPS. Tuning that is planned for MUMPS includes using more global information in the scheduling algorithm and a closer integration of orderings based on nested dissection; for SuperLU, the plan is to exploit more parallelism from the dependency graph and to experiment on the use of immediate primitives to overlap communication and computation.

We started this exercise with the intention of comparing a wider range of sparse codes. However, we have observed that the task of conducting such a comparison is very complex. We do feel that the experience we have gained in this task will be useful in extending the comparisons in the future. In Amestoy et al. (2000b), we summarize the major characteristics of the parallel distributed sparse direct codes of which we are aware.

Acknowledgments
We want to thank James Demmel, Jacko Koster, Chiara Puglisi and Rich Vuduc for very helpful discussions.

References


