Parallel Sparse LU Factorization on Second-class Message Passing Platforms*

Kai Shen
kshen@cs.rochester.edu
Department of Computer Science, University of Rochester

ABSTRACT

Several message passing-based parallel solvers have been developed for general (non-symmetric) sparse LU factorization with partial pivoting. Due to the fine-grain synchronization and large communication volume between computing nodes for this application, existing solvers are mostly intended to run on tightly-coupled parallel computing platforms with high message passing performance (*e.g.*, 1–10 μs in message latency and 100–1000 Mbytes/sec in message throughput). In order to utilize platforms with slower message passing, this paper investigates techniques that can significantly reduce the application's communication needs. In particular, we propose batch pivoting to make pivot selections in groups through speculative factorization, and thus substantially decrease the inter-processor synchronization granularity. We experimented with an MPI-based implementation on several message passing platforms. While the speculative batch pivoting provides no performance benefit and even slightly weakens the numerical stability on an IBM Regatta multiprocessor with fast message passing, it improves the performance of our test matrices by 28–292% on an Ethernet-connected 16-node PC cluster. We also evaluated several other communication reduction techniques and showed that they are not as effective as our proposed approach.

1. INTRODUCTION

The solution of sparse linear systems [9] is a computational bottleneck in many scientific computing problems. Direct methods for solving non-symmetric linear systems often employ partial pivoting to maintain numerical stability. At each step of the LU factorization, the pivoting process performs row exchanges so that the diagonal element has the largest absolute value among all elements of the corresponding pivot column. Parallel sparse LU factorization has been extensively studied in the past. Several solvers (*e.g.*, SuperLU [8, 18], WSMP [16], and PARDISO [23]) were developed specifically to run on shared memory machines and inter-processor communications in them take place through access to the shared memory. Some other solvers (*e.g.*, van der Stappen *et al.* [28], S^+ [24, 25], MUMPS [1], and SuperLU_DIST [19]) employ explicit message passing which allows them to run on non-cache-coherent distributed memory computing platforms.

Despite the apparent portability of message passing-based parallel code, existing solvers were mostly intended to run on tightly-coupled parallel computing platforms with high message passing performance, such as the Cray T3E and the IBM SP2. This is because parallel sparse LU factorization with partial pivoting requires fine-grain synchronization and large communication volume between computing nodes. In particular, since different rows of the matrix may reside on different processors, the column-by-column pivot selection and accompanied row exchanges result in significant message passing overhead. This paper focuses on improving the performance of sparse LU factorization on parallel platforms with relatively poor message passing performance (we call them "second-class" platforms). These platforms may be constrained by the communication capabilities of the network hardware as in PC clusters. The message passing speed may also be slowed down by software overhead such as TCP/IP processing.

The key to improve the performance of parallel sparse LU factorization on second-class message passing platforms is to reduce the inter-processor synchronization granularity and communication volume. In this paper, we examine the message passing overhead in parallel sparse LU factorization with two-dimensional data mapping and investigate techniques to reduce such overhead. Our main finding is that such an objective can be achieved with a small amount of extra computation and slightly weakened numerical stability. Although such tradeoffs would not be worthwhile when running on systems with high message passing performance, these techniques can be very beneficial for second-class message passing platforms.

In particular, we propose a novel technique called *speculative batch pivoting*, under which large elements for a group of columns at all processors are collected at one processor and then the pivot selections for these columns are made together through speculative factorization. These pivot selections are accepted if the chosen pivots pass a numerical stability test. Otherwise, the scheme would fall back to the conventional column-by-column pivot selection for this group of columns. Speculative batch pivoting substan-

*This work was supported in part by NSF grants CCR-0306473, ITR/IIS-0312925, and an NSF CAREER Award CCF-0448413.

ICS'05, June 20-22, Boston, MA, USA.

tially decreases the inter-processor synchronization granularity compared with the conventional approach. This reduction is made at the cost of increased computation (*i.e.*, the cost of speculative factorization).

The rest of this paper is organized as follows. Section 2 introduces some background knowledge on parallel sparse LU factorization. Section 3 assesses the existing application performance on parallel computing platforms with different message passing performance. Section 4 describes techniques that can improve the application performance on second-class message passing platforms. Section 5 presents experimental results of performance and numerical stability on several different message passing platforms. We also provide a direct comparison between our approach and another message passing-based solver. Section 6 discusses the related work and Section 7 concludes the paper.

2. BACKGROUND

LU factorization with partial pivoting decomposes a non-symmetric sparse matrix A into two matrices L and U, such that $PA = LU$, where L is a unit lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix containing the pivoting information. Combined with the forward and backward substitution, the result of LU factorization can be used to solve linear system $Ax = b$. In this section, we describe some key components in parallel sparse LU factorization.

Static symbolic factorization. In sparse LU factorization, some zero elements may become nonzeros at runtime due to factorization and pivoting. Predicting these elements (called *fill-ins*) can help avoid costly data structure variations during the factorization. The static symbolic factorization [15] can identify the worst case fill-ins without knowing numerical values of elements. The basic idea is to statically consider all possible pivoting choices at each step of the LU factorization and space is allocated for all possible nonzero entries. In addition to providing space requirement prediction, static symbolic factorization can also help identify dense components in the sparse matrix for further optimizations.

Since static symbolic factorization considers all possible pivoting choices at each factorization step, it might overestimate the fill-ins which leads to unnecessary space consumption and extra computation on zero elements. Our past experience [24, 25] shows that the static symbolic factorization does not produce too many fill-ins for most matrices. For the exceptions, our previous work proposed several space optimization techniques to address the problem of fill-in overestimation [17].

L/U supernode partitioning. After the fill-in pattern of a matrix is predicted, the matrix can be partitioned using a supernodal approach to identify dense components for better caching performance. In [18], a non-symmetric supernode is defined as a group of consecutive columns, in which the corresponding L part has a dense lower triangular block on the diagonal and the same nonzero pattern below the diagonal. Based on this definition, the L part of each column block only contains dense subrows. Here by "subrow", we mean the contiguous part of a row within a supernode. After an L supernode partitioning has been performed on a sparse matrix A, the same partitioning is applied to the rows of A to further break each supernode into submatrices. Since coarse-grain partitioning can produce large submatrices which do not fit into the cache, an upper bound on the supernode size is usually enforced in the partitioning.

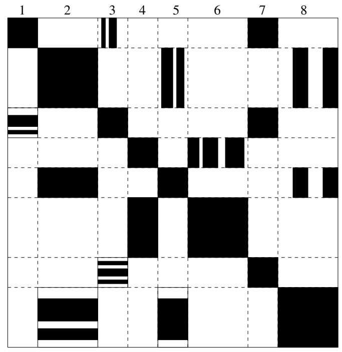

Figure 1: Example of a partitioned sparse matrix.

After the L/U supernode partitioning, each diagonal submatrix is dense, and each nonzero off-diagonal submatrix in the L part contains only dense subrows, and furthermore each nonzero submatrix in the U part of A contains only dense subcolumns. This is the key to maximize the use of BLAS-3 subroutines, which is known to provide high caching performance. Figure 1 illustrates an example of a partitioned sparse matrix and the black areas depict dense submatrices, subrows, and subcolumns.

Data mapping. After symbolic factorization and matrix partitioning, a partitioned sparse matrix A has $N \times N$ submatrix blocks. For example, the matrix in Figure 1 has 8×8 submatrices. For notational differentiation, we use capital letter symbols to represent block-level entities while we use lowercase letter symbols for element-level entities. For example, we use $a_{i,j}$ to represent A's element in row i and column j while $A_{I,J}$ denotes the submatrix in A with row block index I and column block index J. We also let $L_{I,J}$ and $U_{I,J}$ denote a submatrix in the lower and upper triangular part of matrix A respectively.

For block-oriented matrix computation, one-dimensional (1D) column block cyclic mapping and two-dimensional (2D) block cyclic mapping are commonly used. In 1D column block cyclic mapping, a column block of A is assigned to one processor. In 2D mapping, processors are viewed as a 2D grid, and a column block is assigned to a column of processors. Our investigation in this paper focuses on 2D data mapping because it has been shown that 2D sparse LU factorization is substantially more scalable than 1D data mapping [12]. In this scheme, p available processors are viewed as a two dimensional grid: $p = p_r \times p_c$. Then block $A_{I,J}$ is assigned to processor $P_{I \bmod p_r,\ J \bmod p_c}$. Note that each matrix column is scattered across multiple processors in 2D data mapping and therefore pivoting and row exchanges may involve significant inter-processor synchronization and communication.

Program partitioning. The factorization of supernode partitioned sparse matrix proceeds in steps. Step K ($1 \le K \le N$) contains three types of tasks: $Factor(K)$, $SwapScale(K)$, and $Update(K)$.

- Task $Factor(K)$ factorizes all the columns in the Kth column block and its function includes finding the piv-

```
for K = 1 to N
    Perform task Factor(K) if this processor owns a
        portion of column block K;
    Perform task SwapScale(K);
    Perform task Update(K);
endfor
```

Figure 2: Partitioned sparse LU factorization with partial pivoting at each participating processor.

oting sequence associated with those columns and updating the lower triangular portion (the L part) of column block K. The pivoting sequence is held until the factorization of the Kth column block is completed. Then the pivoting sequence is applied to the rest of the matrix. This is called "delayed pivoting".

- Task $SwapScale(K)$ does "row exchanges" which applies the pivoting sequence derived by $Factor(K)$ to submatrices $A_{K:N,\ K+1:N}$. It also does "scaling" that uses the factorized diagonal submatrix $L_{K,K}$ to scale row block U_K.
- Task $Update(K)$ uses factorized off-diagonal column block K $(L_{K+1,K}, \cdots, L_{N,K})$ and row block K $(U_{K,K+1}, \cdots, U_{K,N})$ to modify submatrices $A_{K+1:N,\ K+1:N}$

Figure 2 outlines the partitioned LU factorization algorithm with partial pivoting at each participating processor.

3. PERFORMANCE ON DIFFERENT MESSAGE PASSING PLATFORMS

We assess the existing parallel sparse LU solver performance on three different message passing platforms supporting MPI:

- *PC cluster*: A cluster of PCs connected by 1 Gbps Ethernet. Each machine in the cluster has a 2.8 Ghz Pentium-4 processor, whose double-precision BLAS-3 GEMM performance peaks at 1382.3 MFLOFS. The BLAS/LAPACK package on PC is built on the source release from `http://www.netlib.org/lapack/` using the GNU C/Fortran compilers with the optimization flags "`-funroll-all-loops -O3`". The PC cluster runs MPICH [21] with the TCP/IP-based p4 communication device.
- *Regatta/MPICH*: An IBM p690 "Regatta" multiprocessor with 32 1.3 Ghz Power-4 processor, whose peak double-precision BLAS-3 GEMM performance is 970.9 MFLOPS. The BLAS/LAPACK package on Regatta is built using the IBM C/Fortran compilers with the optimization flag "`-O`". This platform also runs MPICH with the p4 communication device.
- *Regatta/shmem*: The IBM Regatta using shared memory-based message passing. Although MPICH provides a shared memory-based communication device, it does not yet support the IBM multiprocessor running AIX. For the purpose of our experimentation, we use a modified version of MPICH that uses a shared memory region to pass messages between MPI processes.

Figure 3 depicts the message passing performance of the three parallel platforms using an MPI-based ping-pong microbenchmark. We use the single-trip latency to measure the

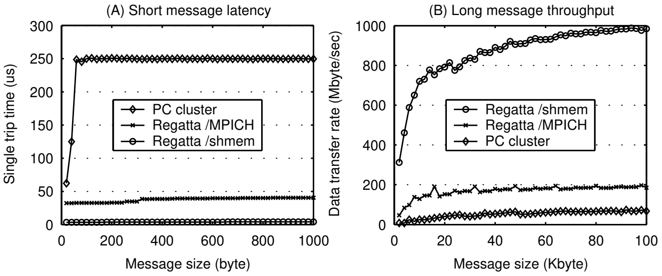

Figure 3: Message passing performance of three parallel computing platforms using an MPI-based ping-pong microbenchmark. Note that we use different metrics for short messages (latency) and long messages (throughput).

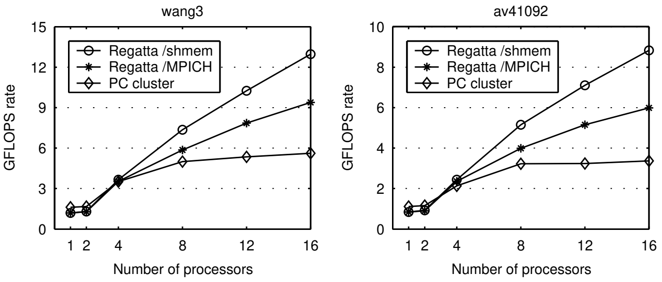

Figure 4: The performance of S^+ for solving two matrices (wang3 and av41092) on three message passing platforms.

short message performance and data transfer rate to measure the long message performance because the message size does not play a dominant role in the overall performance for short messages. Results in Figure 3 show that the short message latency for the three platforms are around $4\,\mu s$, $35\,\mu s$, and $250\,\mu s$ respectively while the long message throughputs are around 980 Mbytes/sec, 190 Mbytes/sec, and 67 Mbytes/sec respectively. Compared with Regatta/shmem, the relatively poor performance of Regatta/MPICH is mainly the result of extra software overhead such as the TCP/IP processing. The message passing performance of the PC cluster is further slowed down by the hardware capability of the Ethernet.

We use the S^+ solver [24, 25] to demonstrate the performance of parallel sparse LU factorization on platforms with different message passing performance. S^+ uses static symbolic factorization, L/U supernode partitioning, and 2D data mapping described in Section 2. In addition, it employs a new scheduling scheme that asynchronously exploits the parallelism available in the application. The comparison of S^+ with some other solvers was provided in [5, 25].

Figure 4 illustrates the S^+ performance for solving two matrices on the three message passing platforms. Detailed statistics about these two matrices and others in our test collection are provided later in Section 5.1. Results in Figure 4 show significant impact of the platform message passing speed on the application performance and such impact grows more substantial at larger scales (*i.e.*, with more processors). Despite the better BLAS performance of the PC processor, the performance on Regatta/shmem is about three times that on the PC cluster at 16 processors.

```
for each column k in column block K
   Find largest local element a_{i,k} (i ≥ k) in the
      column as local pivot candidate;
   Gather all local pivot candidate rows at PE;
   if (I am PE) then
      Select the pivot as the globally largest;
   endif
   Broadcast the pivot row from PE to all processors;
   Swap row if the chosen pivot is local;
   Use the pivot row to update the local portion of
      the column block K;
endfor
```

Figure 5: Illustration of key steps in $Factor(K)$. PE can be any designated processor, such as the one owning the diagonal submatrix.

4. IMPROVING THE PERFORMANCE ON SECOND-CLASS PLATFORMS

The key to support efficient parallel sparse LU factorization on platforms with slow message passing is to reduce the inter-processor synchronization granularity and communication volume. At each step of a solver with 2D data mapping (*e.g.*, step K in Figure 2), there are primarily three types of message passing between processors:

1. Within $Factor(K)$, the pivot selection for each column requires the gathering of local maximums from participating processors at a designated processor (called PE) and the broadcast of final selection back to them. Row swaps within the column block K is then performed if necessary. Note that such communication occurs in a column-by-column fashion because the column block needs to be updated between the pivoting of any two consecutive columns. Figure 5 illustrates the key steps in $Factor(K)$.

2. Within $SwapScale(K)$, "row exchanges" are performed to apply the pivoting sequence derived by $Factor(K)$ to submatrices $A_{K:N,\ K+1:N}$.

3. Before $Update(K)$ can be performed, the factorized off-diagonal column block K ($L_{K+1,K}, \cdots, L_{N,K}$) and row block K ($U_{K,K+1}, \cdots, U_{K,N}$) must be broadcast to participating processors.

It is difficult to reduce the column and row block broadcast for $Update()$ (type 3 communication) without changing the semantics of the LU factorization. However, the other two types of communications can be reduced by using different pivoting schemes. In Section 4.1, we describe the previously proposed threshold pivoting that decreases the number of "row exchanges" in $SwapScale()$. Sections 4.2 and 4.3 present techniques that can lower the synchronization granularity in $Factor()$ through batch pivoting.

4.1 Threshold Pivoting

Threshold pivoting was originally proposed for reducing fill-ins in sparse matrix factorization [9]. It allows the pivot choice to be other than the largest element in the pivot column, as long as it is within a certain fraction ($u \leq 1$) of the largest element. In other words, after the pivoting at column k, the following inequality holds:

$$|a_{k,k}| \geq u \cdot \max_{i>k} \{|a_{i,k}|\}. \tag{1}$$

A smaller u would allow more freedom in the pivot selection, however it might also lead to weakened numerical stability. Several prior studies have empirically examined the appropriate choice for the threshold parameter such that the numerical stability is still acceptable [6, 27]. In particular, Duff recommends to use $u = 0.1$ after analyzing results from these studies [9].

With more freedom in the pivot selection, there is more likelihood that we are able to choose a pivot element residing on the same processor that contains the original diagonal element, and consequently the row exchange for this pivoting step can be performed locally. In this way threshold pivoting can reduce the inter-processor communication volume on row exchanges. This idea was proposed previously for dense LU factorization by Malard [20].

4.2 Large Diagonal Batch Pivoting

Among the three types of message passing (listed in the beginning of this section) for 2D parallel sparse LU factorization, the pivoting in $Factor(k)$ incurs less communication volume compared with the other two types. However, it requires much more frequent inter-processor synchronization since the pivoting selection is performed in a column-by-column fashion while the other types of message passing occur on a once-per-block (or once-per-supernode) basis. In this section and the next, we investigate techniques that allow the pivoting to be performed together for groups of columns (ahead of the numerical updates) such that each group requires only a single round of message passing. Lowering the message frequency would produce significant performance benefit on platforms with long message latency.

Duff and Koster investigated row and column permutations such that entries with large absolute values are moved to the diagonal of sparse matrices [10, 11]. They suggest that putting large entries in diagonal ahead of the numerical factorization allows pivoting down the diagonal to be more stable. The large diagonal permutation was adopted in SuperLU_DIST [19] by Li and Demmel. It allows a priori determination of data structures and communication patterns in parallel execution.

Motivated by these results, we employ large diagonal row permutations for the purpose of pivoting in groups, and thus reducing the inter-processor synchronization frequency in $Factor()$. The objective is to select pivots for a group of columns (*e.g.*, those belonging to one column block or supernode) ahead of the numerical updates, such that for each column k in the group:

$$|a_{k,k}| \geq \max_{i>k} \{|a_{i,k}|\}. \tag{2}$$

Below we describe an approach (we call *large diagonal batch pivoting*) that follows this idea. First each participating processor determines the local pivot candidates for all columns in a column block. Then the pivot candidate sets from all p_r participants are gathered at a designated processor (called PE) and the globally largest element for each column is selected as its pivot. Subsequently the pivots for all columns in the column block are broadcast to participating processors in one message. Batch pivoting requires a single gather-broadcast synchronization for the whole column block. In comparison, the conventional approach requires one gather-broadcast per column.

Except in the case of diagonal dominance, having large elements in the diagonal cannot guarantee the numerical

```
for each column k in column block K
    Find largest local element a_{i,k} (i ≥ k) in the column as local pivot candidate;
endfor
Gather all local pivot candidate rows for all columns in block K at PE;
if (I am PE) then
    Select the pivot for each column as the globally largest;
    Assemble the new diagonal submatrix using the selected pivot rows for all columns in block K;
    Perform factorization on the new diagonal submatrix without any additional row swaps;
    for each column k in column block K
        if (|a_{k,k}| after the factorization / |a_{k,k}| before the factorization ≤ ε) then prepare for abort;
    endfor
endif
Broadcast the selected pivot rows for all columns in block K (or the abort signal) to all processors;
if (abort) then
    Roll back all changes made and call the original Factor(K) with column-by-column pivoting;
else
    for each column k in column block K
        Swap row if the chosen pivot is local;
        Use the pivot row k to update the local portion of the column block K;
    endfor
endif
```

Figure 6: Illustration of $Factor(K)$ using large diagonal batch pivoting. PE is a designated processor.

stability of LU factorization. For example, the factorization of the following block with large elements on the diagonal is numerically unstable without additional row swaps:

$$\begin{pmatrix} 12 & 0 & 8 & 0 \\ 0 & 12 & 8 & 0 \\ 9 & 9 & 12 & 1 \\ 0 & 0 & 1 & 12 \end{pmatrix}$$

The reason for this is that a large diagonal element may become very small or even zero due to updates as the LU factorization proceeds. To address this problem, we conduct a test on the stability of the large diagonal batch pivoting before accepting pivots produced by it. Our approach is to assemble the new diagonal submatrix with large diagonal pivots and perform factorization on the submatrix without any additional row swaps. We then check whether the absolute value of each factorized diagonal element is larger than an error threshold, specified as a small constant (ϵ) times the largest element in the corresponding column before the factorization (which also happens to be the diagonal element). If this inequality does not hold for any column, we consider the large diagonal batch pivoting as unstable and we will fall back to the original column-by-column pivoting to maintain the numerical stability. Figure 6 illustrates the key steps in $Factor(K)$ under this approach.

Note that the large diagonal batch pivoting can be combined with threshold pivoting, in which case we add a threshold parameter u into the right hand size of the inequality (2). This change allows more freedom in pivot selections and consequently we can choose more pivots such that interprocessor row swaps are not required.

4.3 Speculative Batch Pivoting

When the large diagonal batch pivoting can maintain the desired numerical stability for most or all column blocks, the approach can significantly reduce the application synchronization overhead. If it often fails the stability test, however, it must fall back to the original column-by-column pivoting and therefore it merely adds overhead for the application. In order for any batch pivoting scheme to be successful, there must be a high likelihood that its pivot selections would result in numerically stable factorization.

We attempt to achieve a high level of numerical stability by determining the pivots for a group of columns through speculative factorization. The first part of this approach is the same as that of the large diagonal batch pivoting. Each participating processor determines the local pivot candidates for the group of columns and then the pivot candidate sets from all p_r participants are gathered at a designated processor (called PE). If there are C_K columns in the column block K, then all candidate pivot rows would form a submatrix with $C_K \cdot p_r$ rows and C_K columns at PE. We then perform full numerical factorization on such a submatrix and determine the pivots for each of the C_K columns one by one. For each column, the element with the largest absolute value is chosen as the pivot. This approach is different from the large diagonal batch pivoting because elements in subsequent columns may be updated as the numerical factorization proceeds column-by-column. We call this approach batch pivoting through speculative factorization, or *speculative batch pivoting* in short.

The pivot selection factorization at PE is performed with only data that has already been gathered at PE. Therefore no additional message passing is required for the pivot selections. The factorization incurs some extra computation overhead. However, such cost is negligible compared with the saving on the communication overhead when running on second-class message passing platforms.

The pivot sequence chosen by the speculative batch pivoting is likely to be numerically more stable than that of the large diagonal batch pivoting because it considers numerical updates during the course of LU factorization. However, it still cannot guarantee the numerical stability because some rows are excluded in the factorization at PE (only local pivot candidates are gathered at PE). This limitation is hard to avoid since gathering all rows at PE would be too expensive in terms of communication volume and the computation cost. To address the potential numerical instability, we

Matrix	Order	$\|A\|$	Floating point ops	Application domain
olafu	16146	1015156	9446.3 million	Structure problem
ex11	16614	1096948	33555.6 million	Finite element modeling
e40r0100	17281	553562	7133.6 million	Fluid dynamics
shermanACb	18510	145149	26154.6 million	Circuit and device simulation
raefsky4	19779	1328611	26887.9 million	Container modeling
af23560	23560	484256	23272.0 million	Navier-Stokes Solver
wang3	26064	177168	171900.5 million	Circuit and device simulation
av41092	41092	1683902	130474.5 million	Partial differential equation

Table 1: Test matrices and their statistics.

examine the produced pivots before accepting them. During the pivot selection factorization at PE, we check whether the absolute value of each factorized diagonal element is larger than a specified error threshold. The threshold is specified as a small constant (ϵ) times the largest element in the corresponding column before the factorization. If this inequality does not hold for any column, we consider the speculative batch pivoting as unstable and we will fall back to the original column-by-column pivoting to maintain the numerical stability. The structural framework of $Factor(K)$ under this approach is similar to that of the large diagonal batch pivoting illustrated in Figure 6. We do not show it specifically in another figure to save space.

5. EXPERIMENTAL EVALUATION

We have implemented the communication reduction techniques described in the previous section using MPI. The implemented code (S^+ version 1.1) can be downloaded from the web [26]. The main objective of our evaluation is to demonstrate the effectiveness of these techniques on parallel computing platforms with difference message passing performance. Section 5.1 describes the evaluation setting in terms of the application parameters, platform specifications, and properties of test matrices. Sections 5.2 and 5.3 present the overall performance and numerical stability of the implemented code respectively. Section 5.4 shows direct effects of individual techniques described in this paper. Finally, Section 5.5 provides a direct comparison of our approach with another message passing-based solver SuperLU_DIST [19].

5.1 Evaluation Setting

Application parameters. The parallel sparse LU factorization code in our evaluation uses 2D data mapping. p available processors are viewed as a two dimensional grid $p = p_r \times p_c$ such that $p_r \leq p_c$ and they are as close as possible. For example, 16 processors are organized into a 4-row 4-column grid while 8 processors are arranged into a 2-row 4-column grid. In our code, the threshold pivoting parameter $u = 0.1$. For the two batch pivoting schemes, the numerical test error threshold parameter $\epsilon = 0.001$. We specify that a supernode can contain at most 25 columns. Note that many supernodes cannot reach this size limit since only consecutive columns/rows with the same (or similar) nonzero patterns can be merged into a supernode. All our experiments use double precision numerical computation.

Our experiments compare the performance of several different versions of the application:

#1. *ORI*: the original S^+ [24, 25] without any techniques described in Section 4.

#2. *TP*: the original version with threshold pivoting.

#3. *TP+LD*: the original version with threshold pivoting and large diagonal batch pivoting.

#4. *TP+SBP*: the original version with threshold pivoting and speculative batching pivoting.

Platform specifications. The evaluations are performed on three MPI platforms: a PC cluster, an IBM Regatta running MPICH p4 device, and the IBM Regatta using shared memory message passing. The specifications of these platforms were given earlier in Section 3.

Statistics of test matrices. Table 1 shows the statistics of the test matrices used in our experimentation. All matrices can be found at Davis' UF sparse matrix collection [7]. Column 2 in the table lists the number of columns/rows for each matrix and Column 3 shows the number of nonzeros in the original matrices. In column 4, we show the number of floating point operations required for factorizing each of these matrices when the ORI version of our code is employed. Note that due to the overestimation of the symbolic factorization, the floating point operation count includes some on zero elements. Therefore the FLOPS numbers shown in this paper are not directly comparable to those produced by solvers that do not employ the static symbolic factorization (*e.g.*, SuperLU). However, this should not affect our objective of evaluating communication reduction techniques on platforms with difference message passing performance.

5.2 Overall Performance

We examine the overall GFLOPS performance of all test matrices with up to 16 processors. Figures 7, 8, and 9 illustrate such performance on Regatta/shmem, Regatta/MPICH, and the PC cluster respectively.

Due to the high message passing performance on Regatta/shmem, results in Figure 7 show very little benefit for any of the communication reduction techniques (threshold pivoting, large diagonal batch pivoting, or speculative batch pivoting). Moreover, we even found performance degradation of threshold pivoting for matrix raefsky4 (up to 25%), and to a lesser extent for matrices shermanACb (up to 13%) and wang3 (up to 9%). Further measurement shows that such performance degradation is attributed to different amount of computation required for different pivoting schemes. More specifically, pivoting schemes that produce more nonzeros in the pivot rows would require more computation in subsequent updates. Although it is possible to control the number of nonzeros in the pivot rows with threshold pivoting, such control would require additional inter-processor communications. We plan to investigate such an issue in the future.

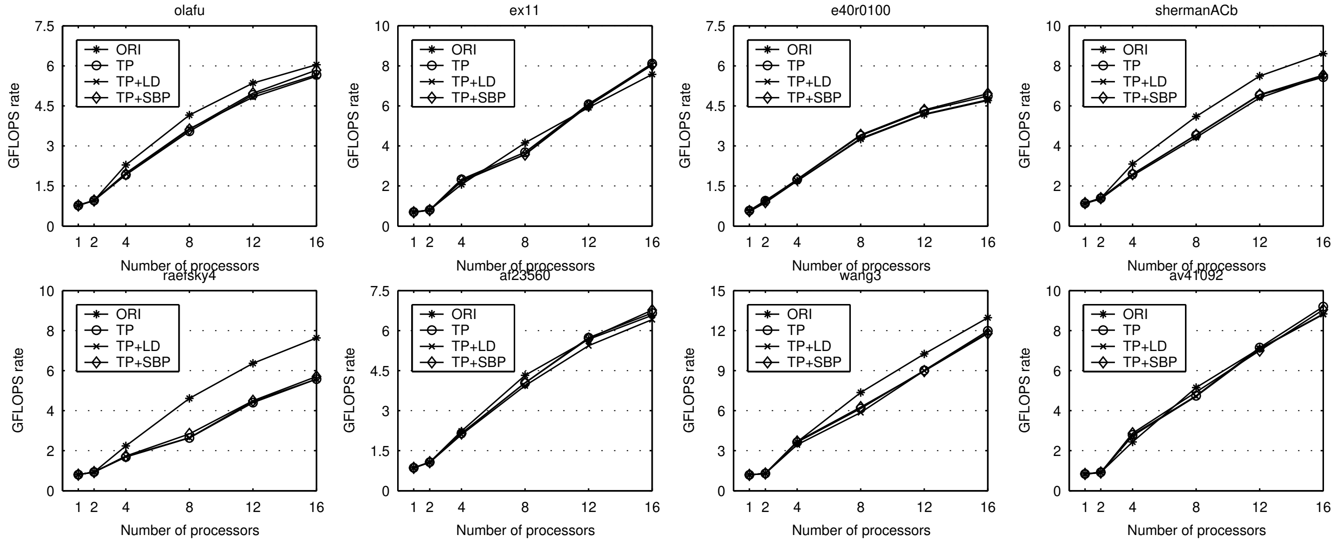

Figure 7: LU factorization performance on the IBM Regatta using a shared memory-based MPI runtime system. ORI stands for the original S^+; TP stands for *Threshold Pivoting*; LD stands for *Large Diagonal Batch Pivoting*; and SBP stands for *Speculative Batch Pivoting.*

Figure 8 shows the application GFLOPS performance on Regatta/MPICH, whose message passing performance is between those of Regatta/shmem and the PC cluster. The result for threshold pivoting is similar to that on Regatta/shmem — no benefit and even performance degradation on some matrices. By comparing TP+SBP and TP, we find the effectiveness of speculative batch pivoting is quite significant for some test matrices. Particularly for olafu, e40r0100, and af23560 at 16 processors, the speculative batch pivoting improves the application performance by 47%, 48%, and 33% respectively. In contrast, the large diagonal batch pivoting is not very effective in enhancing the performance. This is because many batch pivotings under LD do not pass the numeric stability test and must fall back to column-by-column pivoting.

Figure 9 illustrates the results on the PC cluster, which supports much slower message passing compared with Regatta/shmem (up to 60 times longer message latency and around 1/15 of its message throughout). By comparing TP+SBP and TP, the speculative batch pivoting shows substantial performance benefit for all test matrices — ranging from 28% for ex11 to 292% for olafu. In comparison, the improvement for LD is relatively small, again due to its inferior numerical stability and more frequent employment of the column-by-column pivoting. We observe some small performance benefit of threshold pivoting. With the exception of ex11 (25% at 16 processors), the benefit of threshold pivoting is below 13% for all other matrices. We also notice poor scalability (at 8 and more processors) for the four smaller matrices: olafu, ex11, e40r0100, and shermanACb. However, the larger matrices exhibit scalable performance for up to 16 processors.

5.3 Numerical Stability

We examine the numerical errors of our communication reduction techniques. We calculate numerical errors in the following fashion. After the LU factorization for A, one can derive the solution $\tilde{x}$ of linear system $Ax = b$ for any right-hand side b using the forward and backward substitution.

Matrix	ORI	TP	TP+LD	TP+SBP
olafu	5.6 E-11	1.7 E-10	6.2 E-09	4.3 E-10
ex11	3.2 E-13	4.0 E-11	1.8 E-09	5.6 E-11
e40r0100	4.2 E-13	1.7 E-09	1.8 E-07	7.8 E-10
shermanACb	2.0 E-09	2.0 E-09	2.0 E-09	2.0 E-09
raefsky4	7.1 E-08	2.0 E-02	1.5 E-01	8.4 E-04
af23560	1.4 E-12	2.2 E-10	3.6 E-08	1.7 E-10
wang3	5.0 E-11	3.0 E-10	1.9 E-06	9.3 E-11
av41092	1.4 E-10	7.1 E-07	5.3 E-03	1.8 E-05

Table 2: Numerical errors at 16 processors.

We then define the numerical error of the solution as

$$\max_{1\le i\le n} \frac{|(A\tilde{x})_i - b_i|}{\sum_{1\le j\le n} |A_{i,j}\cdot \tilde{x}_j| + |b_i|},$$

where $\cdot_i$ indicates the ith element in the vector. This definition of numerical error measures the "componentwise relative backward error" of the computed solution as described in [3, 22]. We choose all unit unknowns in our error calculation, or $b = A \cdot (1.0\ 1.0\ \cdots\ 1.0)^T$.

Table 2 lists the numerical errors of ORI, TP, TP+LD, and TP+SBP at 16 processors. Results show various levels of increases on numerical errors by each communication reduction scheme. Among them, TP+LD incurs the most amount of error increase for our test matrices. Particularly for matrices raefsky4 and av41092, the absolute errors are 1.5E-01 and 5.3E-03 respectively for TP+LD. In comparison, TP+SBP still maintains a high degree of numerical stability and no matrix exhibits an error larger than 8.4E-04. More importantly, the speculative batch pivoting incurs no obvious additional error over threshold pivoting.

5.4 Direct Effects of Individual Techniques

We examine the direct effects of individual techniques in this section. The main effect of threshold pivoting is to reduce the inter-processor row exchanges and therefore lower the communication volume. Figure 10 shows the number of

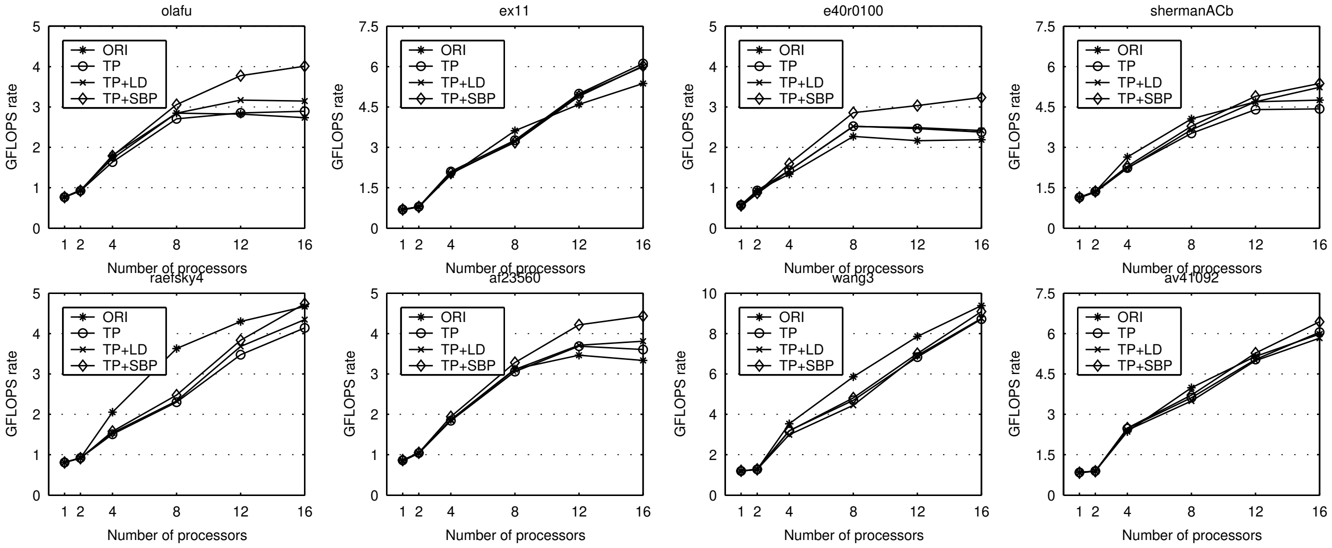

Figure 8: LU factorization performance on the IBM Regatta using MPICH with the p4 device.

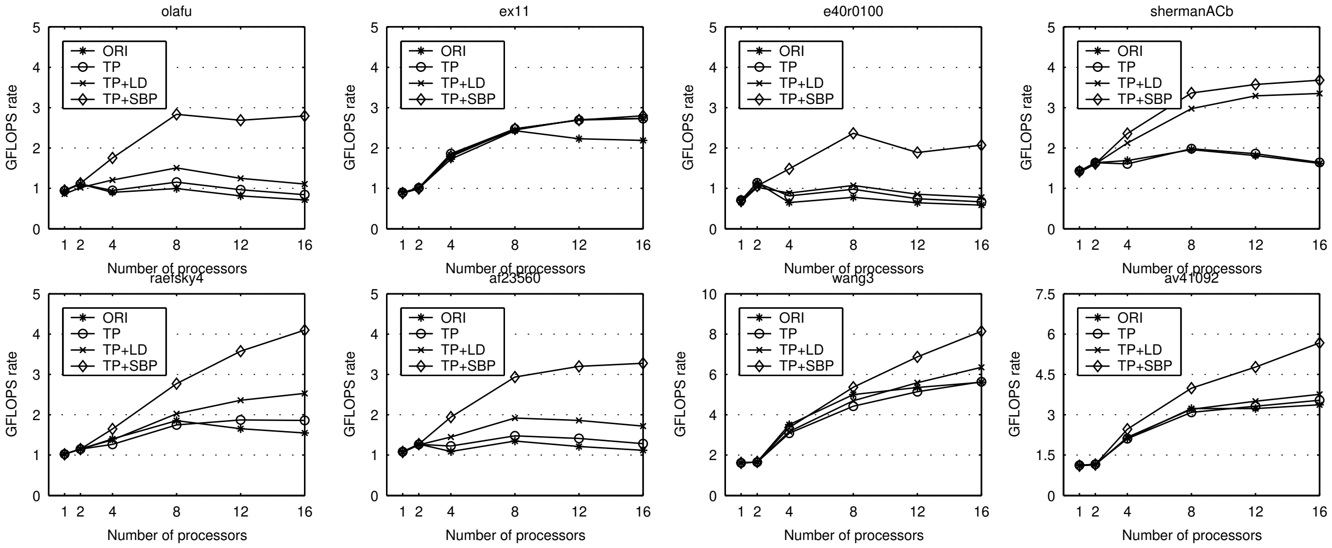

Figure 9: LU factorization performance on the PC cluster.

inter-processor row exchanges for different solvers. We observe that the threshold pivoting reduces the inter-processor row exchanges by 39–95%. Other solvers (TP+LD and TP+SBP) inherit such a reduction by incorporating the threshold pivoting. Despite such communication reduction, results in Section 5.2 showed that the performance enhancement of threshold pivoting is very modest. We believe this is because the message latency is more critical than the message throughput for the performance of our application on second-class message passing platforms. Consequently, lowering the communication volume is not as beneficial as reducing the synchronization count.

Figure 11 shows the number of gather-broadcast synchronizations during pivot selection for different solvers. Both ORI and TP require one synchronization for each column in the matrix. TP+LD and TP+SBP may reduce the number of synchronizations by performing batch pivoting. However, they must fall back to column-by-column pivoting when the batch pivoting cannot produce desired numerical stability. Results in Figure 11 show significant message reduction for TP+LD (32–78%) and more reduction for TP+SBP (37–93%). This explains the superior performance of TP+SBP on the PC cluster and on Regatta/MPICH.

5.5 Comparison with SuperLU_DIST

We provide a direct comparison between SuperLU_DIST [19] and the TP+SBP approach. SuperLU_DIST permutes large elements to the diagonal before the numerical factorization. These diagonal elements are also pre-determined pivots and no further pivot selections will be performed during the factorization (called *static pivoting*). Static pivoting allows fast factorization for several reasons. First, it permits the accurate prediction of fill-ins ahead of the factorization, which would result in much smaller space and computation costs compared with symbolic factorization that considers all possible pivot choices. Second, static pivoting eliminates the

Matrix	Factorization time		Solve time		Numerical errors	
	TP+SBP	SuperLU_DIST	TP+SBP	SuperLU_DIST	TP+SBP	SuperLU_DIST
ex11	11.985 sec	2.920 sec	0.193 sec	0.597 sec (2 IR steps)	5.6E-11	7.8E-08
raefsky4	6.557 sec	3.490 sec	0.157 sec	0.347 sec (1 IR steps)	8.4E-04	2.5E-07
wang3	21.121 sec	9.700 sec	0.400 sec	1.270 sec (2 IR steps)	9.3E-11	1.6E-16
av41092	23.002 sec	18.180 sec	0.753 sec	1.550 sec (1 IR steps)	1.8E-05	**7.8E-01**

Table 3: Comparison with SuperLU_DIST on factorization time, solve time, and numerical stability (at 16 processors on the PC cluster). The solve time of SuperLU_DIST includes the time of iterative refinements.

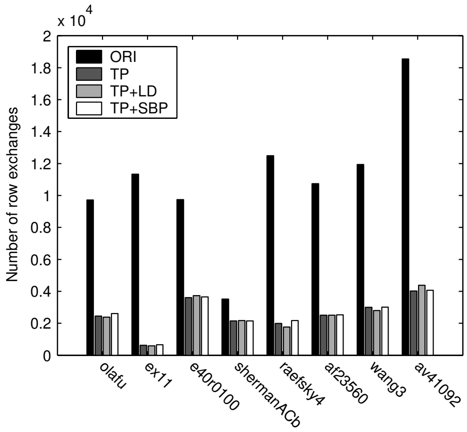

Figure 10: The number of inter-processor row exchanges at 16 processors.

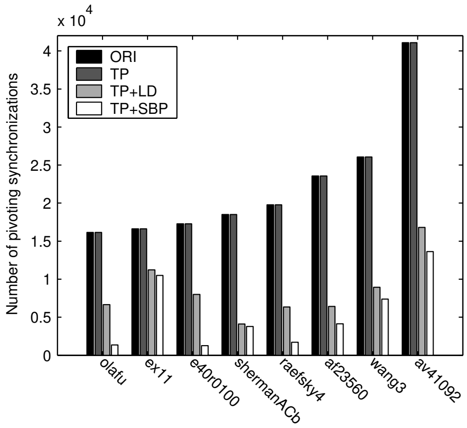

Figure 11: The number of gather-broadcast synchronizations at 16 processors.

need for pivoting-related inter-processor communications (*e.g.*, pivot selection and row exchanges) during the factorization. This would be particularly beneficial for second-class message passing platforms where the communication performance is relatively poor. Unlike our approach, however, static pivoting cannot guarantee the numerical stability of the LU factorization. Therefore, SuperLU_DIST employs post-solve iterative refinement to improve the stability.

Table 3 shows the performance of our approach and SuperLU_DIST on factorization time, solve time, and numerical stability for the four largest matrices (in terms of the floating point operation count) in our test collection. Results are at 16 processors on the PC cluster. Note that the solve time of SuperLU_DIST includes the time of iterative refinements. Thanks to its static pivoting, the factorization time of SuperLU_DIST is significantly faster than TP+SBP. However, SuperLU_DIST requires two or three times more solve time due to the additional iterative refinement. The solve time is particularly important for problem sets with the same coefficient matrix A but different right-hand side b since the factorization only needs to be performed once for all problems with the same coefficient matrix. This is also one fundamental performance tradeoff between direct solvers and iterative solvers.

Results in Table 3 show that SuperLU_DIST's post-solve iterative refinement can achieve a high level of numerical stability for most matrices. However, the numerical error is substantial for av41092. After relaxing the default stop condition of the iterative refinement in SuperLU_DIST, the numerical error of av41092 arrives at 1.9E-11 after 7 steps of iterative refinement. However, the solve time also reaches 6.332 seconds at this setting.

6. RELATED WORK

Parallel sparse LU factorization has been extensively studied in the past. Solvers such as SuperLU [8, 18], WSMP [16], and PARDISO [23] are designed to run on shared memory parallel computers. Several others, including van der Stappen *et al.* [28], S^+ [24, 25], MUMPS [1], and SuperLU_DIST [19], employ message passing so they can run on distributed memory machines. Except for SuperLU_DIST, which employs static pivoting, most existing solvers are only intended to run on tightly coupled parallel computers with high message passing performance.

Malard employed threshold pivoting to reduce inter-processor row interchanges for dense LU factorization [20]. Duff and Koster [10, 11] and Li and Demmel [19] have explored permuting large entries to the diagonal as a way to reduce the need of pivoting during numerical factorization. Built on these results, our work is the first to quantitatively assess the effectiveness of these techniques on platforms with different message passing performance.

Gallivan *et al.* proposed a matrix reordering technique to exploit large-grain parallelism in solving parallel sparse linear systems [13]. Although larger-grain parallelism typically results in less frequent inter-processor communications, their work only targets work-stealing style solvers on shared memory multiprocessors. Previous studies explored broadcasting/multicasting strategies for distributing pivot columns or rows while achieving load balance [14, 20]. In comparison, our work focuses on the performance on second-class message passing platforms where reducing the communication overhead is more critical than maintaining computational load balance.

Many earlier studies examined application-level techniques to address performance issues in underlying computing platforms. For instance, Amestoy *et al.* studied the impact of the MPI buffering implementation on the performance of sparse matrix solvers [2]. Chakrabarti and Yelick investigated application-controlled consistency mechanisms to minimize synchronization and communication overhead for solving the Gröbner basis problem on distributed memory machines [4]. Our work addresses a different platform-related problem — improving the performance of parallel sparse LU factorization on platforms with relatively poor message passing performance.

7. CONCLUSION

Functional portability of MPI-based message passing applications does not guarantee their *performance portability.* In other words, applications optimized to run on a particular platform may not perform well on other MPI platforms. This paper investigates techniques that can improve the performance of parallel sparse LU factorization on systems with relatively poor message passing performance. In particular, we propose speculative batch pivoting which can enhance the performance of our test matrices by 28–292% on an Ethernet-connected 16-node PC cluster. Our experimental results also show that this technique does not significantly affect the numerical stability of the LU factorization.

Our work is an early step toward building adaptive applications which can adjust themselves according to the characteristics and constraints of the underlying computing platforms. We believe application-level adaptation is particularly effective when tradeoffs can be made among multiple metrics for the application. For parallel sparse LU factorization, our work finds that substantial reduction in message passing overhead can be attained at the cost of some extra computation and slightly weakened numerical stability.

Software Availability

Techniques described in this paper have been incorporated into a parallel sparse linear system solver (S^+ version 1.1). S^+ can be downloaded from the web [26].

8. 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 J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[2] P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and X. S. Li. Impact of the Implementation of MPI Point-to-Point Communications on the Performance of Two General Sparse Solvers. *Parallel Computing*, 29:833–849, 2003.

[3] M. Arioli, J. W. Demmel, and I. S. Duff. Solving Sparse Linear Systems with Sparse Backward Error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, 1989.

[4] S. Chakrabarti and K. Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Proc. of the 4th ACM Symp. on Principles and Practice of Parallel Programming*, pages 169–178, San Diego, CA, June 1993.

[5] M. Cosnard and L. Grigori. Using Postordering and Static Symbolic Factorization for Parallel Sparse LU. In *Proc. of the Int'l Parallel and Distributed Processing Symp.*, Cancun, Mexico, May 2000.

[6] A. R. Curtis and J. K. Reid. The Solution of Large Sparse Unsymmetric Systems of Linear Equations. *J. Inst. Maths. Applics.*, 8:344–353, 1971.

[7] T. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[8] J. W. Demmel, S. Eisenstat, J. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986.

[10] I. S. Duff and J. Koster. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[11] I. S. Duff and J. Koster. On Algorithms for Permuting Large Entries to the Diagonal of A Sparse Matrix. *SIAM J. Matrix Anal. Appl.*, 20(4):973–996, 2001.

[12] C. Fu, X. Jiao, and T. Yang. A Comparison of 1-D and 2-D Data Mapping for Sparse LU Factorization on Distributed Memory Machines. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[13] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff. The Parallel Solution of Nonsymmetric Sparse Linear Systems Using the H* Reordering and An Associated Factorization. In *Proc. of the 8th ACM Conf. on Supercomputing*, pages 419–430, Manchester, UK, July 1994.

[14] G. Geist and C. Romine. Parallel LU Factorization on Message Passing Architecture. *SIAM J. Sci. Stat. Comput.*, 9(4):639–649, July 1988.

[15] A. George and E. Ng. Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6):877–898, November 1987.

[16] A. Gupta. WSMP: Watson Sparse Matrix Package (Part-II: Direction Solution of General Sparse Systems). Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, 2000.

[17] B. Jiang, S. Richman, K. Shen, and T. Yang. Efficient Sparse LU Factorization with Lazy Space Allocation. In *Proc. of the 9th SIAM Conf. on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999.

[18] X. S. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, Computer Science Division, EECS, UC Berkeley, 1996.

[19] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Software*, 29(2):110–140, June 2003.

[20] J. Malard. Threshold Pivoting for Dense LU Factorization on Distributed Memory Multiprocessors. In *Proc. the ACM/IEEE Conf. on Supercomputing*, pages 600–607, Albuquerque, NM, November 1991.

[21] MPICH – A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[22] W. Oettli and W. Prager. Compatibility of Approximate Solution of Linear Equations with Given Error Bounds for Coefficients and Righ Hand Sides. *Num. Math.*, 6:405–409, 1964.

[23] O. Schenk and K. Gärtner. Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, April 2004.

[24] K. Shen, X. Jiao, and T. Yang. Elimination Forest Guided 2D Sparse LU Factorization. In *Proc. of the 10th ACM Symp. on Parallel Algorithms and Architectures*, pages 5–15, Puerto Vallarta, Mexico, June 1998.

[25] K. Shen, T. Yang, and X. Jiao. S+: Efficient 2D Sparse LU Factorization on Parallel Machines. *SIAM J. Matrix Anal. Appl.*, 22(1):282–305, 2000.

[26] The S^+ Project Web Site. http://www.cs.rochester.edu/u/kshen/research/s+.

[27] J. A. Tomlin. Pivoting for Size and Sparsity in Linear Programming Inversion Routines. *J. Inst. Maths. Applics.*, 10:289–295, 1972.

[28] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.