Parallel Sparse LU Factorization on Different Message Passing Platforms

Kai Shen

Department of Computer Science, University of Rochester
Rochester, NY 14627, USA

Abstract

Several message passing-based parallel solvers have been developed for general (non-symmetric) sparse LU factorization with partial pivoting. Existing solvers were mostly deployed and evaluated on parallel computing platforms with high message passing performance (*e.g.*, 1–10 μs in message latency and 100–1000 Mbytes/sec in message throughput) while little attention has been paid on slower platforms. This paper investigates techniques that are specifically beneficial for LU factorization on platforms with slow message passing. In the context of the S^+ distributed memory solver, we find that significant reduction in the application message passing overhead can be attained at the cost of extra computation and slightly weakened numerical stability. 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 on three different message passing platforms with different communication speeds. While the proposed techniques provide no performance benefit and even slightly weaken numerical stability on an IBM Regatta multiprocessor with fast message passing, they improve the performance of our test matrices by 15–460% on an Ethernet-connected 16-node PC cluster. Given the different tradeoffs of communication-reduction techniques on different message passing platforms, we also propose a sampling-based runtime application adaptation approach that automatically determines whether these techniques should be employed for a given platform and input matrix.

Key words: Applications and performance analysis, Parallel algorithms and implementations, Sparse LU factorization with partial pivoting, Message passing performance, Application adaptation

Email address: `kshen@cs.rochester.edu` (Kai Shen).
URL: `www.cs.rochester.edu/u/kshen` (Kai Shen).

1 Introduction

The solution of sparse linear systems [1] 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. Sparse LU factorization has been extensively studied in the past. Solvers like UMFPACK [2] run on serial computing platforms. Parallel solvers such as SuperLU_MT [3,4], WSMP [5], and PARDISO [6] were developed specifically for shared memory machines and inter-processor communications in them take place through access to the shared memory. Other solvers like van der Stappen *et al.* [7], S^+ [8,9], SPOOLES [10], MUMPS [11], and SuperLU_DIST [12] 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 deployed and evaluated on tightly-coupled parallel computing platforms with high message passing performance, *e.g.*, those with 1–10 μs in message latency and 100–1000 Mbytes/sec in message throughput. They may exhibit poor scalability on much slower message passing platforms that are constrained by the network hardware capabilities (such as Ethernet-connected PC clusters) or the software overhead (like TCP/IP processing). One primary reason for slow LU factorization performance on these platforms is that the application requires fine-grain synchronization and large communication volume between computing nodes. The key to improve the parallel application performance on platforms with slow message passing is to reduce the application inter-processor communications.

This paper studies communication-reduction techniques in the context of the S^+ solver [8,9], which uses static symbolic factorization, supernodal matrix partitioning, and two-dimensional data mapping. In such a solver, the column-by-column pivot selection and accompanied row exchanges result in significant message passing overhead when selected pivots do not reside on the same processors as corresponding diagonals. In this paper, 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 substantially 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) and slightly weakened numerical stability.

Communication-reduction techniques such as the speculative batch pivoting and previously proposed threshold pivoting [1] may not yield much performance improvement when running on systems with fast message passing. Considering their potential side effects (*e.g.*, weakened numerical stability), it might not be worthwhile to employ these techniques on such platforms. Therefore it is important to determine the appropriateness of these techniques according to the underlying message passing speed. Such decision might also depend on the input matrix whose nonzero patterns and numerical values affect the application communication patterns. In this paper, we propose a runtime sampling approach to estimate the benefit of communication-reduction techniques on the underlying message passing platform and input matrix. The decision on whether to employ these techniques is then made based on the estimated performance gain.

We acknowledge that our proposed pivoting techniques would not benefit solvers that already require no communications for pivoting. In particular, SuperLU_DIST [12] performs pre-factorization large diagonal permutation and then pivots down the diagonal entries without any row swapping during factorization. MUMPS [11,13] distributes data in a way that pivoting is within one processor and thus no message passing is required. Some other communication-reduction techniques might be needed to improve the performance and scalability of these solvers on platforms with slow message passing. These issues fall beyond the scope of this paper.

We organize the rest of this paper 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 platforms with slow message passing. Section 5 evaluates the performance and numerical stability of communication-reduction techniques on several different message passing platforms. We also provide a direct comparison of the new S^+ with the latest SuperLU_DIST and MUMPS solvers. Section 6 presents the design and evaluation of a sampling-based application adaptation scheme. Section 7 discusses related work and Section 8 concludes the paper.

2 Background on Parallel Sparse LU Factorization

LU factorization with partial pivoting decomposes a non-symmetric sparse matrix A into two matrices L and U, such that $PAQ = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. P and Q are permutation matrices containing the pivoting and column reordering information respectively. 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 and the S^+ solver in particular.

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 [14] 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. As a result, S^+ with static symbolic factorization has relatively slow uni-processor performance. On the other hand, it exhibits competitive parallel performance [8,9] due to the exploitation of dense data structures and the absence of runtime data structure variation.

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 [4], 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.

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

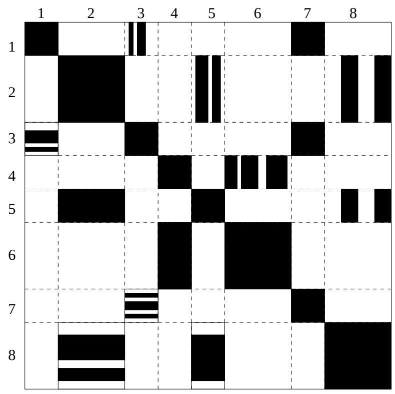

Fig. 1. Example of a partitioned sparse matrix.

only dense subcolumns. This is the key to maximize the use of BLAS-3 subroutines [15], 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 [16]. 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 ($1 \leq K \leq N$) contains three types of tasks: $Factor(K)$, $SwapScale(K)$, and $Update(K)$.

```
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
```

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

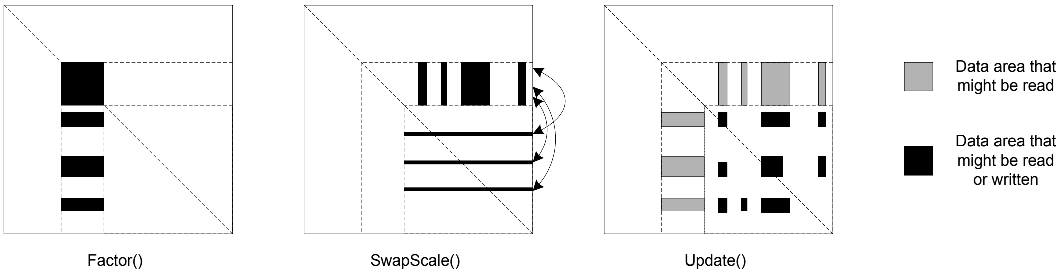

Fig. 3. Illustration of data areas involved in each step of the LU factorization.

- Task $Factor(K)$ factorizes all the columns in the Kth column block and its function includes finding the pivoting sequence associated with those columns and updating the lower triangular portion (the L part) of column block K. Note that the pivoting sequence is not applied to the rest of the matrix until later.
- 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 uses the factorized diagonal submatrix $L_{K,K}$ to scale row block K ($U_{K,K+1:N}$).
- Task $Update(K)$ uses factorized off-diagonal column block K ($L_{K+1:N,K}$) and row block K ($U_{K,K+1:N}$) 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. We also provide an illustration of the data areas involved in each step of $Factor()$, $SwapScale()$, and $Update()$ tasks (Figure 3).

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 4465.6 MFLOFS. The BLAS/LAPACK package on PC is built from ATLAS [17] 3.6.0 using the GNU C/Fortran

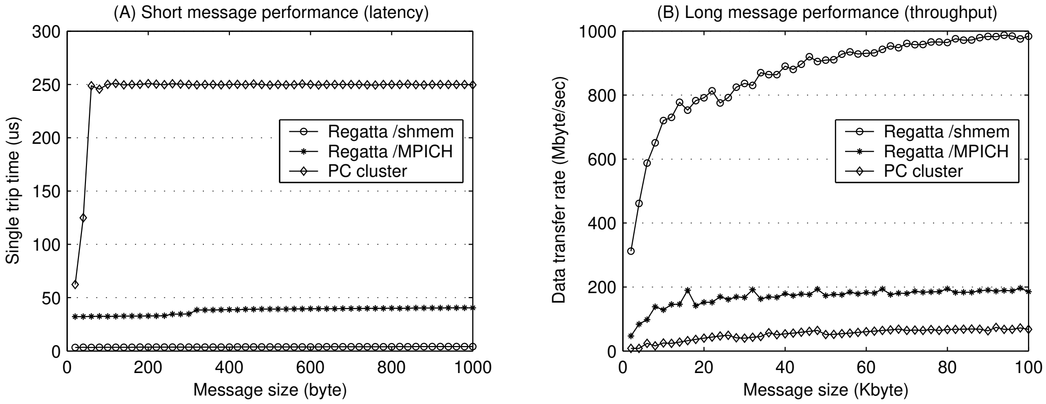

Fig. 4. Message passing performance of three parallel computing platforms using an MPI-based ping-pong microbenchmark. Note that we use different metrics for short messages (latency) and long messages (throughput).

compilers. The PC cluster runs MPICH [18] 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 3232.3 MFLOPS. The BLAS/LAPACK package on Regatta is built from ATLAS 3.6.0 using the IBM C/Fortran compilers. 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 4 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 short message performance and data transfer rate to measure the long message performance. Results in Figure 4 show that the short message latency for the three platforms are around 250 μs, 35 μs, and 4 μs respectively while the long message throughputs are around 67 Mbytes/sec, 190 Mbytes/sec, and 980 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 [8,9] 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. The comparison of S^+ with some other solvers was provided in [19,9]. Figure 5 illustrates the S^+ performance for solving two

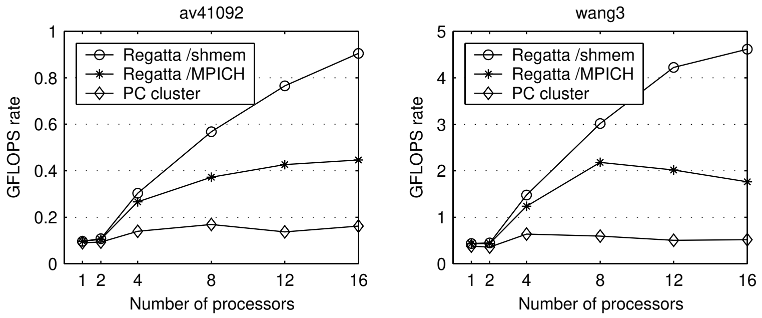

Fig. 5. The performance of S^+ for solving two matrices (av41092 and wang3) on three message passing platforms.

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 5 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 more than five times that on the PC cluster at 16 processors.

4 Communication-reduction Techniques

The key to support efficient parallel sparse LU factorization on platforms with slow message passing is to reduce the inter-processor communications. At each step of the S^+ 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 6 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 ($L_{K+1:N,K}$) and row block K ($U_{K,K+1:N}$) 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

```
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
```

Fig. 6. Illustration of key steps in $Factor(K)$. PE can be any designated processor, such as the one owning the diagonal submatrix.

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 [1]. 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 numerical stability is still acceptable [20–22]. In particular, Duff recommends to use $u = 0.1$ after analyzing results from these studies [1].

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 [23].

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 pivot 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 pivot selection 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 synchronization 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 [24,25]. 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 [12] 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 numerical stability of LU factorization. For example, the factorization of the following 4-by-4 block with large elements on the diagonal is numerically unstable without additional row swaps. More specifically, its third pivot becomes numerically zero after two steps of the LU factorization.

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

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 there is one or more columns for which this inequality does not hold, we consider the large diagonal batch pivoting as unstable and we will fall back to the original column-by-column pivoting to maintain numerical stability.

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 side of the inequality (2). This change allows more freedom in pivot selections and consequently we can choose more pivots such that inter-processor 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 each participating processor would send c_K rows (one for a candidate pivot at each column) and altogether the 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*.

The submatrix factorization at PE (for pivot selection) is performed with only the 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. However, such cost is negligible compared with the saving on the communication overhead when running on platforms with slow message passing.

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 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 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 there is one or more columns for which this inequality does not hold, we consider the speculative batch pivoting as unstable and we will fall back to the original column-by-column pivoting to maintain numerical stability.

5 Experimental Evaluation of Communication-reduction Techniques

We have implemented the techniques described in the previous section using MPI. The implementation was made on top of our original S^+ solver. The main objective of our evaluation is to demonstrate the effectiveness of these techniques on parallel computing platforms with different 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 LU factorization performance and numerical stability of the implemented code respectively. Section 5.4 provides a direct comparison of the new S^+ with latest versions of SuperLU_DIST [12] and MUMPS [11].

5.1 Evaluation Setting

Application parameters The parallel sparse LU factorization code in our evaluation uses 2D data mapping. We view p available processors 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. We preprocess all matrices using the column approximate minimum degree ordering (COLAMD) [26]. In our code, we set the threshold pivoting parameter u at 0.1. For the two batch pivoting schemes, we set the numerical test error threshold parameter ϵ at 0.001. We specify that a supernode can contain at most 28 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. The above parameters were chosen empirically. Our experiments have shown that slightly different parameter settings do not change our evaluation results significantly. All our experiments use double precision numerical computation.

We assess the effectiveness of individual techniques by comparing the performance of several different versions of the application:

#1. *ORI*: the original S^+ [8,9] using 2D data mapping. This version does not contain 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. Most specifications of these platforms were given earlier in Section 3. Each machine in the PC cluster has one Gigabyte main memory. The IBM Regatta has 32 Gigabyte memory.

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 [27]. Column 2 in the table lists the number of columns/rows

Matrix	Order	$\|A\|$	factor entries / $\|A\|$: SuperLU	S^+	Floating point op counts: SuperLU	S^+	Application domain
heart1	3557	1387773	2.22	5.76	1554 million	8611 million	Bioengineering problem
olafu	16146	1015156	6.06	11.06	2184 million	3778 million	Structure engineering
raefsky3	21200	1488768	5.46	12.65	2684 million	9682 million	Structure engineering
af23560	23560	484256	23.70	39.45	5020 million	4893 million	Airfoil modeling
av41092	41092	1683902	6.52	28.42	5600 million	84828 million	Partial differential equation
ex11	16614	1096948	10.52	18.44	6003 million	10412 million	Finite element modeling
raefsky4	19779	1328611	9.90	38.26	7772 million	13508 million	Structure engineering
mark3jac100sc	45769	285215	47.26	160.21	9401 million	88678 million	MULTIMOD Mark3 modeling
wang3	26064	177168	63.15	379.33	9865 million	41488 million	Semiconductor device simulation
mark3jac140sc	64089	399735	52.44	209.76	16186 million	213640 million	MULTIMOD Mark3 modeling
ns3Da	20414	1679599	11.34	29.87	16933 million	69334 million	Finite element modeling
torso1	116158	8516500	3.26	6.46	22595 million	48276 million	Bioengineering problem
g7jac160sc	47430	656616	42.39	75.47	37834 million	81879 million	Social security system modeling
sinc18	16428	973826	29.74	120.56	46499 million	220441 million	Material science problem
g7jac200sc	59310	837936	44.50	77.85	53548 million	108443 million	Social security system modeling
ecl32	51993	380415	109.75	277.61	60713 million	212253 million	Semiconductor device simulation

Table 1
Test matrices and their statistics.

for each matrix and column 3 shows the number of nonzeros in the original matrices. In columns 4 and 5 of the table, we list the total number of nonzero factor entries divided by $|A|$ for dynamic factorization (reported by SuperLU_DIST [12]) and static symbolic factorization (reported by S^+). These numbers were reported when SuperLU_DIST runs with the minimum degree matrix ordering [28] of $A^T + A$ and S^+ runs with the column approximate minimum degree matrix ordering (COLAMD) [26]. Static symbolic factorization produces more factor entries due to over-estimation of fill-in entries. In columns 6 and 7, we show the number of factorization floating point operations, reported by SuperLU_DIST and S^+ respectively. We use the SuperLU_DIST floating point operation counts to calculate FLOPS rates reported later in this paper. Therefore the FLOPS rates in this paper do not include the over-estimations caused by static symbolic factorization in solvers like S^+.

5.2 LU Factorization Performance

We examine the the LU factorization performance of all test matrices for 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 tech-

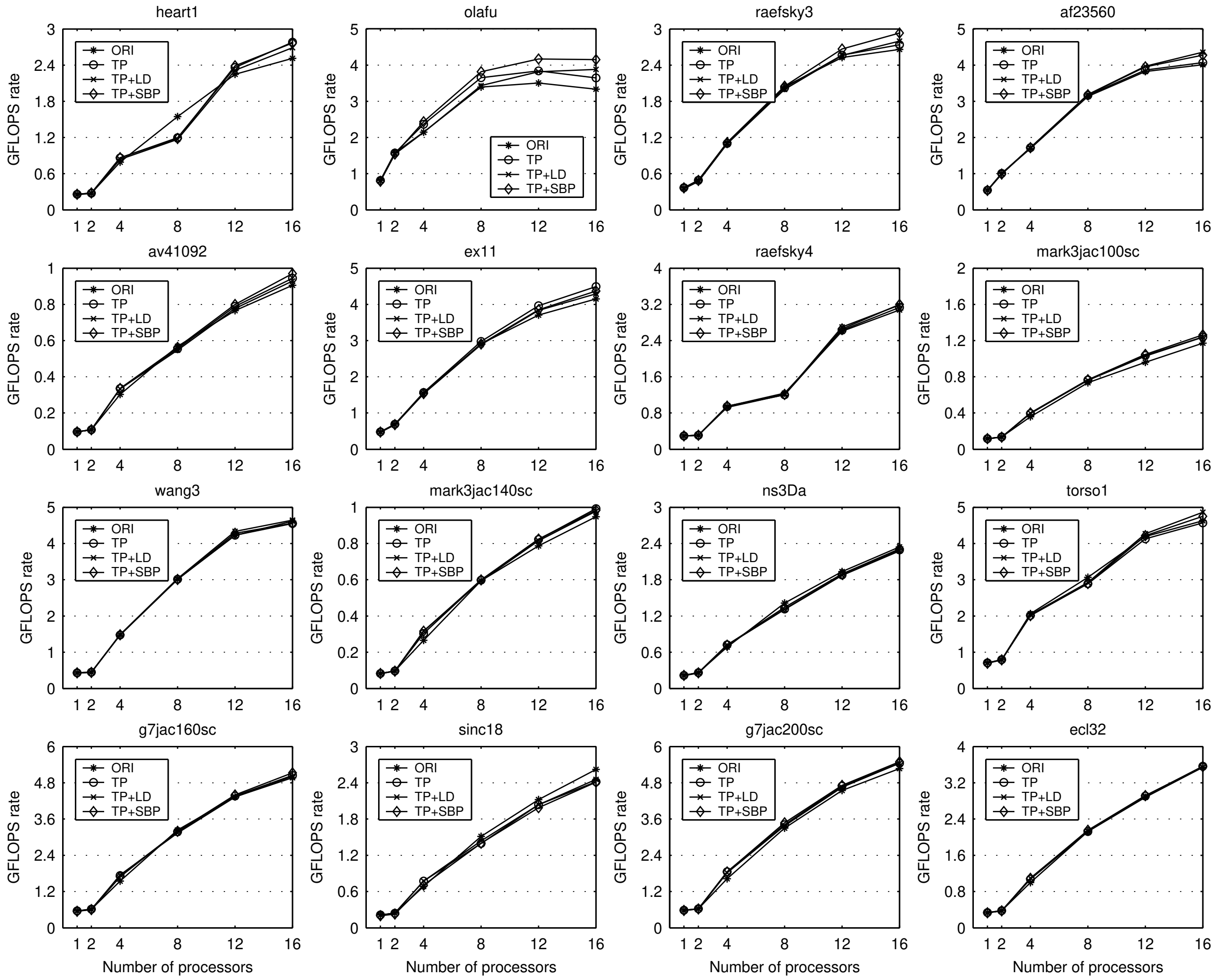

Fig. 7. LU factorization performance on the IBM Regatta using a shared memory-based MPI runtime system.

niques (threshold pivoting, large diagonal batch pivoting, or speculative batch pivoting). Moreover, we even found slight performance degradation of threshold pivoting for matrix sinc18. Such 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.

Figure 8 illustrates the application GFLOPS performance on Regatta/MPICH. The results show very little benefit of threshold pivoting. At the same time, we find the effectiveness of speculative batch pivoting is quite significant for many of the test matrices. Particularly for olafu, raefsky3, af23560, raefsky4, and wang3 at 16 processors, the speculative batch pivoting improves the application performance by 186%, 96%, 61%, 52%, and 99% respectively. In contrast, the large diagonal batch pivoting is not very effective in enhancing

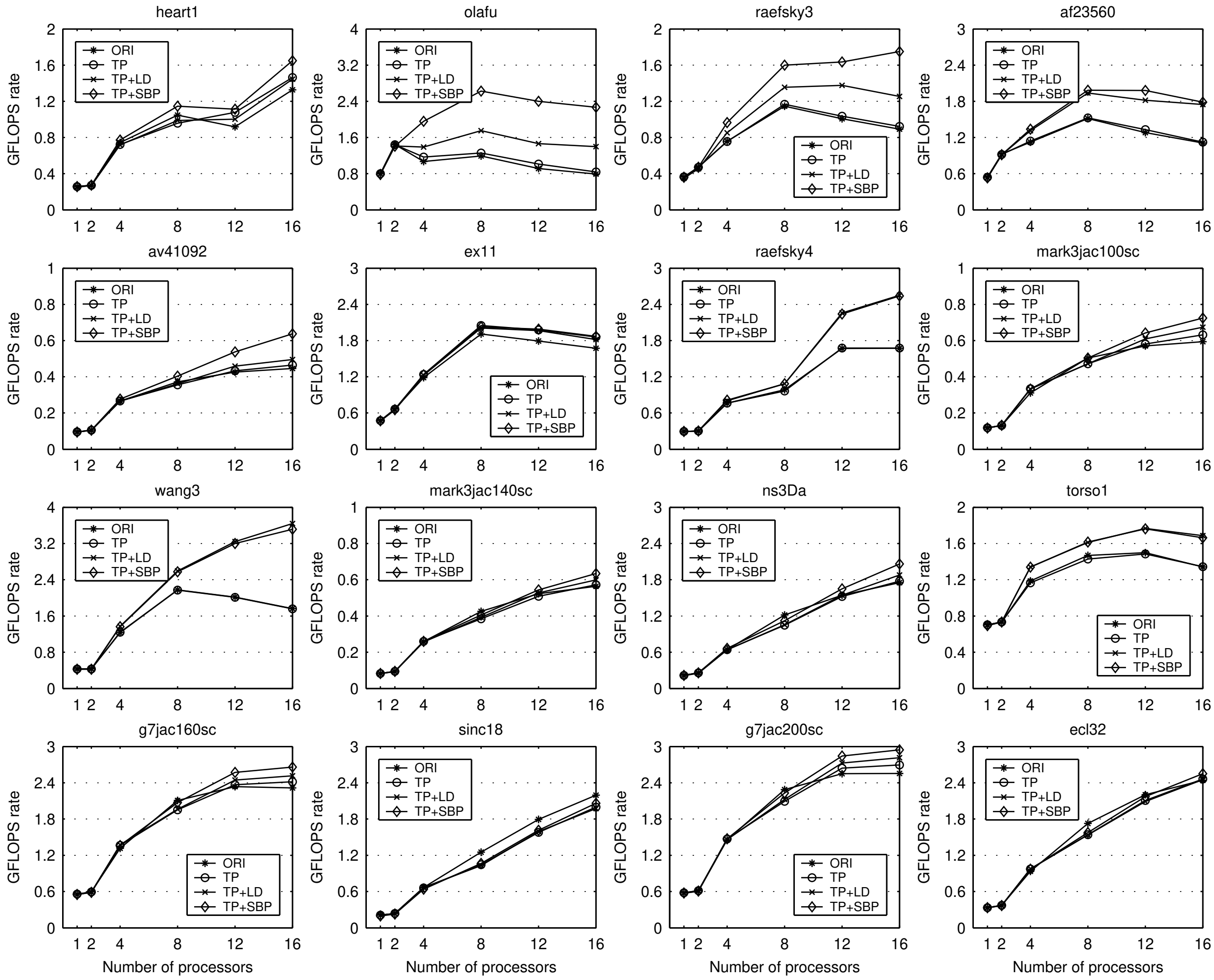

Fig. 8. LU factorization performance on the IBM Regatta using MPICH with the p4 communication device.

the performance. This is because many batch pivotings under LD do not pass numeric stability test and must fall back to column-by-column pivoting.

Results in Figure 9 illustrate the application performance on the PC cluster, which has much worse message passing performance 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 show substantial performance benefit for all test matrices — ranging from 15% for sinc18 to 460% 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 certain performance benefit of threshold pivoting for some matrices (up to 18%). We also notice poor scalability (with 8 or more processors) for the several small matrices: olafu, raefsky3, af23560, and ex11. However, the other (mostly larger) matrices exhibit scalable performance for up to 16 processors.

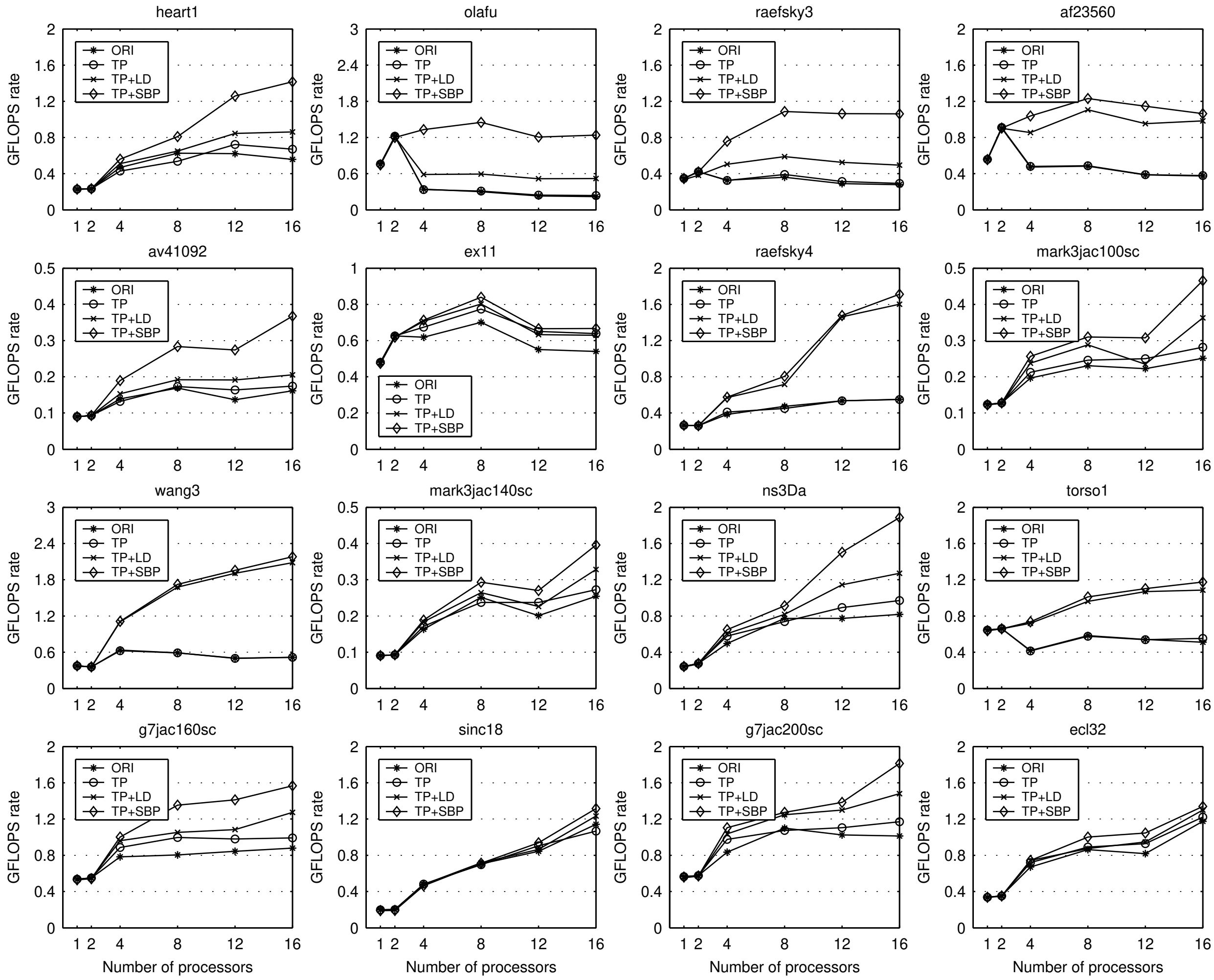

Fig. 9. LU factorization performance on the PC cluster.

To better illustrate the benefits of LD and SBP, we measure the number of gather-broadcast synchronizations during pivot selection for different solvers (shown in Figure 10). 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 10 show significant message reduction for TP+LD (19–96%) and for TP+SBP (41–96%).

5.3 Numerical Stability

We examine 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. We then define the

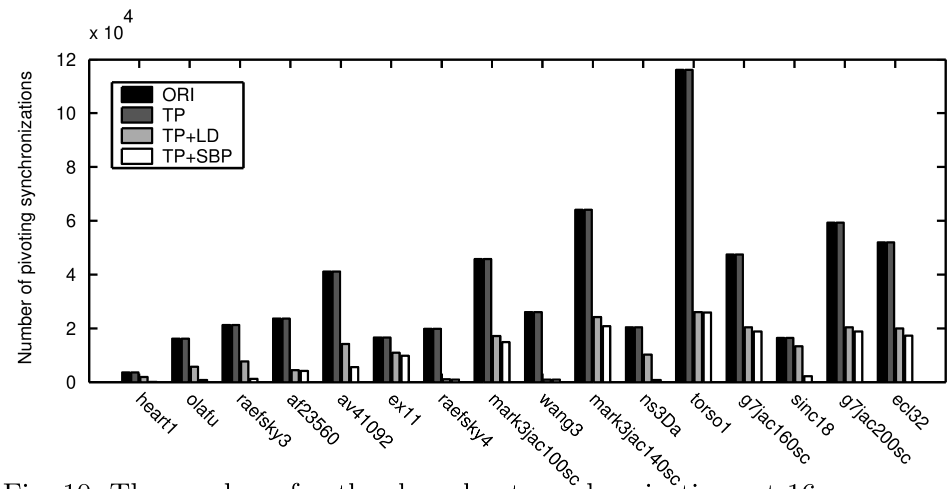

Fig. 10. The number of gather-broadcast synchronizations at 16 processors.

Matrix	ORI	TP	TP+LD	TP+SBP
heart1	8.9 E-12	3.1 E-11	3.3 E-09	2.8 E-11
olafu	7.9 E-12	1.4 E-11	9.6 E-12	7.0 E-12
raefsky3	2.0 E-10	1.8 E-09	6.0 E-06	5.8 E-09
af23560	1.7 E-14	5.5 E-14	1.7 E-09	4.0 E-14
av41092	2.6 E-11	3.0 E-08	1.5 E-05	1.1 E-06
ex11	2.6 E-13	1.8 E-11	5.4 E-11	2.1 E-11
raefsky4	4.7 E-11	9.0 E-09	2.3 E-07	3.8 E-09
mark3jac100sc	5.9 E-10	3.9 E-08	5.7 E-06	5.9 E-08
wang3	3.1 E-15	3.1 E-15	3.0 E-15	3.1 E-15
mark3jac140sc	8.4 E-10	3.4 E-08	3.7 E-06	5.4 E-08
ns3Da	1.6 E-14	1.0 E-11	3.7 E-09	2.4 E-11
torso1	5.0 E-14	3.8 E-12	1.2 E-10	1.7 E-12
g7jac160sc	5.5 E-13	1.6 E-10	1.5 E-07	5.4 E-11
sinc18	2.4 E-11	1.8 E-09	5.4 E-07	1.4 E-09
g7jac200sc	8.0 E-13	5.5 E-11	2.9 E-07	6.4 E-10
ecl32	1.5 E-07	5.5 E-06	3.7 E-04	3.7 E-06

Table 2
Numerical errors at 16 processors.

numerical error of the solution as:

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

where $\cdot_i$ indicates the ith element in the vector. This is also the "backward error" used in SuperLU [29]. We choose all unit unknowns in our error calculation, or $b = A\cdot(1.0\ 1.0\ \cdots\ 1.0)^T$.

Table 2 lists numerical errors of ORI, TP, TP+LD, and TP+SBP for our test matrices 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 av41092 and ecl32, the absolute errors are 1.5E-05 and 3.7E-04 respectively for TP+LD. In comparison, TP+SBP still maintains a high degree of numerical stability and no matrix exhibits an error larger than 3.7E-06. More importantly, the speculative batch pivoting incurs no obvious additional error over threshold pivoting.

5.4 Comparison with SuperLU_DIST and MUMPS

To assess the absolute performance of the TP+SBP version of S^+ on platforms with slow message passing, we compare it with solvers SuperLU_DIST [12] (version 2.0) and MUMPS [11] (version 4.5.0) on the Linux cluster. 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 because it permits the accurate prediction of fill-ins ahead of the factorization and it eliminates the need for pivoting-related inter-processor communications during the factorization. However, static pivoting cannot guarantee numerical stability of the LU factorization. Therefore, SuperLU_DIST employs post-solve iterative refinement to improve the stability of the results. The MUMPS solver uses a multifrontal method and its pivoting does not require any additional inter-processor communication messages (although it might increase the size of messages during factorization due to fill-ins). In our experiments, all solvers use their default "fill-in"-reduction ordering schemes. Both S^+ and MUMPS use the column approximate minimum degree ordering (COLAMD) [26] while SuperLU_DIST employs the minimum degree ordering [28] of $A^T + A$.

Table 3 shows the factorization time of the three solvers and the post-solve refinement time of SuperLU_DIST on the PC cluster. Since different solvers may achieve peak performance at different numbers of processors, we show the best factorization time among 1, 2, 4, 8, 16, 20, and 24-processor results for each solver. Comparing S^+ with SuperLU_DIST, we find that SuperLU_DIST has slightly better factorization time. However, it incurs additional iterative refinement time in order to maintain numerical stability. Note that the refinement has to be performed for solving each linear system problem while problem sets with the same coefficient matrix A but different right-hand side b only require a single factorization. This is also one fundamental performance tradeoff between direct solvers and iterative solvers.

Results in Table 3 show that MUMPS is faster than S^+ and SuperLU_DIST for most matrices. This is partially attributed to its highly optimized computation

Matrix	Best factorization time			Refinement time
	S^+ (TP+SBP)	SuperLU_DIST	MUMPS	SuperLU_DIST
heart1	1.10 sec (16 procs)	0.61 sec (8 procs)	1.58 sec (2 procs)	0.18 sec
olafu	1.41 sec (8 procs)	1.56 sec (8 procs)	0.50 sec (4 procs)	0.13 sec
raefsky3	2.42 sec (8 procs)	1.86 sec (2 procs)	0.95 sec (2 procs)	0.26 sec
af23560	2.45 sec (8 procs)	4.10 sec (2 procs)	1.24 sec (2 procs)	0.42 sec
av41092	15.25 sec (16 procs)	5.65 sec (2 procs)	2.90 sec (2 procs)	0.35 sec
ex11	7.14 sec (8 procs)	3.06 sec (4 procs)	1.17 sec (2 procs)	0.28 sec
raefsky4	2.98 sec (16 procs)	3.47 sec (4 procs)	2.97 sec (2 procs)	0.19 sec
mark3jac100sc	20.19 sec (16 procs)	15.40 sec (2 procs)	12.24 sec (1 procs)	1.28 sec
wang3	4.52 sec (16 procs)	5.75 sec (2 procs)	3.13 sec (2 procs)	0.35 sec
mark3jac140sc	40.54 sec (16 procs)	23.10 sec (8 procs)	26.24 sec (1 procs)	1.84 sec
ns3Da	8.98 sec (16 procs)	4.51 sec (16 procs)	4.64 sec (2 procs)	0.50 sec
torso1	19.22 sec (16 procs)	14.98 sec (2 procs)	*11.87 sec (2 procs)*	5.10 sec
g7jac160sc	24.15 sec (16 procs)	23.64 sec (8 procs)	12.24 sec (1 procs)	1.58 sec
sinc18	27.11 sec (16 procs)	13.70 sec (16 procs)	24.26 sec (2 procs)	5.50 sec
g7jac200sc	29.52 sec (16 procs)	31.61 sec (8 procs)	18.81 sec (1 procs)	2.03 sec
ecl32	40.54 sec (16 procs)	17.76 sec (8 procs)	10.79 sec (2 procs)	1.00 sec

Table 3

Performance comparison with SuperLU_DIST and MUMPS. Factorization time results are the best of 1, 2, 4, 8, 12, 16, 20, 24-processor results on the PC cluster. We also indicate the number of processors at which the peak performance is achieved. Note that MUMPS 4.5.0 runs out of memory at 1, 2, and 4 processors for torso1. We used the latest MUMPS 4.6.1 to produce results for torso1.

routines (written in Fortran). We find that MUMPS achieves its peak performance mostly at one or two processors. In fact, the MUMPS performance at 4 or more processors is much worse than its uni-processor performance. At one hand, this provides one evidence that existing solvers often do not scale well on slow message passing platforms such as Ethernet-connected PC clusters. On the other hand, since MUMPS achieves very high performance at one or two processors (in many cases better than other solvers at any processor count), there is not much scope to exploit parallelism.

Table 4 shows the numerical stability of the three solvers. MUMPS's numerical stability is somehow better than S^+'s (particularly for matrices raefsky3, raefsky4, and ecl32). Overall, both solvers can achieve acceptable numerical stability for all test matrices. We find 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 refinement time also increases to 2.45 seconds at this setting.

Matrix	S^+ (TP+SBP)	SuperLU_DIST	MUMPS
heart1	2.8 E-11	4.0 E-16 (2 IR steps)	1.4 E-14
olafu	7.0 E-12	7.2 E-08 (1 IR step)	2.3 E-13
raefsky3	5.8 E-09	4.9 E-16 (1 IR step)	7.2 E-15
af23560	4.0 E-14	2.5 E-16 (2 IR steps)	1.8 E-15
av41092	1.1 E-06	*7.8 E-01 (1 IR step)*	2.0 E-09
ex11	2.1 E-11	7.8 E-08 (2 IR steps)	1.3 E-14
raefsky4	3.8 E-09	2.5 E-07 (1 IR step)	1.3 E-14
mark3jac100sc	5.9 E-08	2.7 E-16 (3 IR steps)	1.3 E-10
wang3	3.1 E-15	1.2 E-16 (2 IR steps)	1.3 E-15
mark3jac140sc	5.4 E-08	3.3 E-16 (3 IR steps)	7.6 E-12
ns3Da	2.4 E-11	2.7 E-16 (2 IR steps)	2.5 E-15
torso1	1.7 E-12	1.5 E-14 (8 IR steps)	9.6 E-14
g7jac160sc	5.4 E-11	9.7 E-16 (3 IR steps)	2.7 E-11
sinc18	1.4 E-09	2.3 E-16 (11 IR steps)	1.9 E-10
g7jac200sc	6.4 E-10	1.2 E-15 (3 IR steps)	3.4 E-10
ecl32	3.7 E-06	2.2 E-16 (2 IR steps)	1.8 E-12

Table 4
Comparison with SuperLU_DIST and MUMPS on numerical stability. We also indicate the number of iterative refinement steps for SuperLU_DIST.

6 Runtime Application Adaptation

An efficient application design would naturally attempt to minimize its communication overhead. However, communication-reduction design choices are not straightforward when compromises on other aspects of the application have to be made. For parallel sparse LU factorization, the speculative batch pivoting can decrease the inter-processor synchronization frequency for the pivot selections while the threshold pivoting can reduce the inter-processor row exchanges. At the same time, these techniques may incur additional computation overhead and weaken the solver's numerical stability. The worthiness of these techniques would depend on the platform properties and application characteristics. Specifically, the overall communication overhead is affected by the message passing performance on the underlying computing platforms, including the inter-processor link latency and bandwidth. In addition, it is also influenced by the application communication needs such as the synchronization frequency and message sizes.

Our goal is to construct an adaptive application that can automatically determine whether the communication-reduction techniques should be employed according to the characteristics of the underlying computing platform and the input matrix. This can be achieved by building a performance model that predicts the effectiveness of communication-reduction techniques under given platform properties and application characteristics. Such performance models have been constructed for other high-performance computing applications in the past, both on the application computation performance [30,31] and on the message passing performance [32,33]. However, it is challenging to build accurate performance models for irregular applications such as the parallel sparse LU factorization because their data structures and execution behaviors are hard to predict. For instance, the computation and communication patterns for dense matrix operations usually depend only on the input matrix size while the complete matrix nonzero pattern and numerical values can significantly affect the application behaviors for parallel sparse LU factorization.

6.1 Sampling-based Application Adaptation

Our approach in this paper is to employ runtime sampling to estimate the benefit of communication-reduction techniques under the current setting and then determine whether these techniques should be employed. At the high level, we divide each application execution into two phrases. The sampling phase includes the first $\alpha \cdot N$ $(0 < \alpha < 1)$ steps of the LU factorization shown in Figure 2. During this phase, we first execute the TP+SBP version of $Factor(K)$ (described in Section 4.3) but we always abort it (as if the numerical stability test always fails) and then perform the original $Factor(K)$ without any communication-reduction techniques. This allows us to assess the potential effectiveness of TP+SBP without actually employing it and thus free of its side effects (*e.g.*, weakened numerical stability) in the sampling phase. We keep track of potential communication reductions (both in the blocking synchronization count and in the communication volume) of speculative batching pivoting and threshold pivoting. We count a blocking synchronization when one MPI process must wait for a message from another MPI process to proceed. Some communications are inherently concurrent (*e.g.*, the gathering of local pivot candidates from all processors to a designated processor) and we count them only once.

At the end of the sampling phase, we accumulate the estimated reductions in the blocking synchronization count and in the communication volume (denoted as $\mathrm{S_{reduction}}$ and $\mathrm{V_{reduction}}$ respectively). Also let $\mathrm{L_{msg}}$ and $\mathrm{B_{msg}}$ denote the message passing latency and bandwidth of the underlying platform, which can be measured using simple microbenchmarks. We then calculate the potential saving of the communication-reduction techniques in proportion to the sampling phase elapsed time ($\mathrm{T_{sampling}}$) as:

$$\mathrm{p_{saving}} = \frac{\mathrm{S_{reduction}} \cdot \mathrm{L_{msg}} + \frac{\mathrm{V_{reduction}}}{\mathrm{B_{msg}}}}{\mathrm{T_{sampling}}}$$

We subsequently use the sampling phase potential saving to guide the rest of the application execution. Specifically, we would employ the TP+SBP version

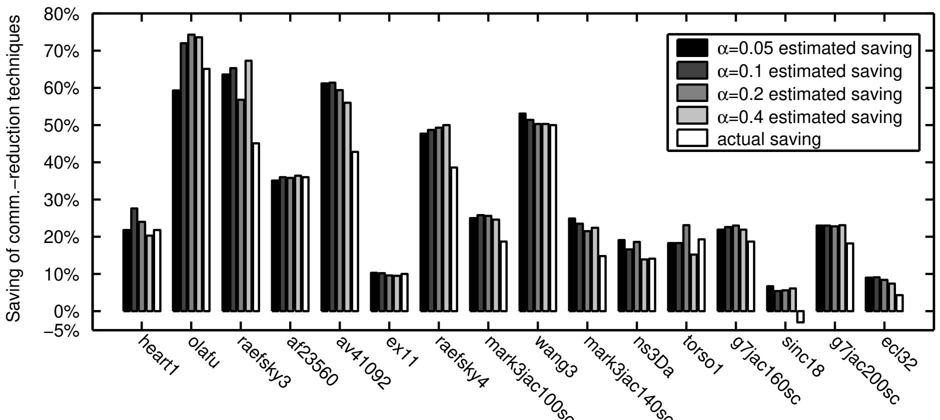

Fig. 11. Our adaptation approach's sensitivity to the α parameter. Results are at 16 processors for the IBM Regatta using MPICH with the p4 device.

of $Factor(K)$ for the rest of the LU factorization if p_{saving} exceeds a specified threshold β $(0 < \beta < 1)$. We use the original $Factor(K)$ otherwise.

We examine the parameter setting for our sampling-based application adaptation. The sampling length parameter α should provide a balance between the sampling accuracy and its overhead. The longer the sampling phase is, the more accurately the communication-reduction statistics collected over the sampling phase would reflect those of the whole application execution. However, a longer sampling phase incurs more runtime overhead. Further, since no communication-reduction techniques are applied during the sampling phase, a longer sampling phase also reduces the potential benefit of these techniques. We perform experiments to learn our approach's sensitivity to the α parameter in practice. In the experiments, we compare the sampling-phase estimated saving of communication reduction techniques at different α settings (0.05, 0.1, 0.2, and 0.4) with the actual saving. We only show results on Regatta/MPICH (Figure 11) because it is a more interesting case for runtime adaptation than PC cluster and Regatta/shmem (in the latter two cases the choices on whether to employ the communication-reduction techniques are quite obvious). Results suggest that the sampling-phase estimation is not very sensitive to the choice of α in practice. We also find that for some matrices (raefsky3, av41092, raefsky4, mark3jac100sc, mark3jac140sc, and sinc18), the sampling-phase estimations at all tested α settings are larger than the actual saving. This indicates different execution patterns between the front portion and the later portion of LU factorization. However, such estimation inaccuracies are quite small (within 20% in all cases). We choose $\alpha = 0.2$ for the experiments described later in this paper.

The setting of the performance saving threshold β can control the tradeoff between performance and numerical stability. More specifically, it represents how much performance benefit is worth the risk of slightly weakened numerical stability. This parameter setting is largely a user decision and we choose $\beta =$

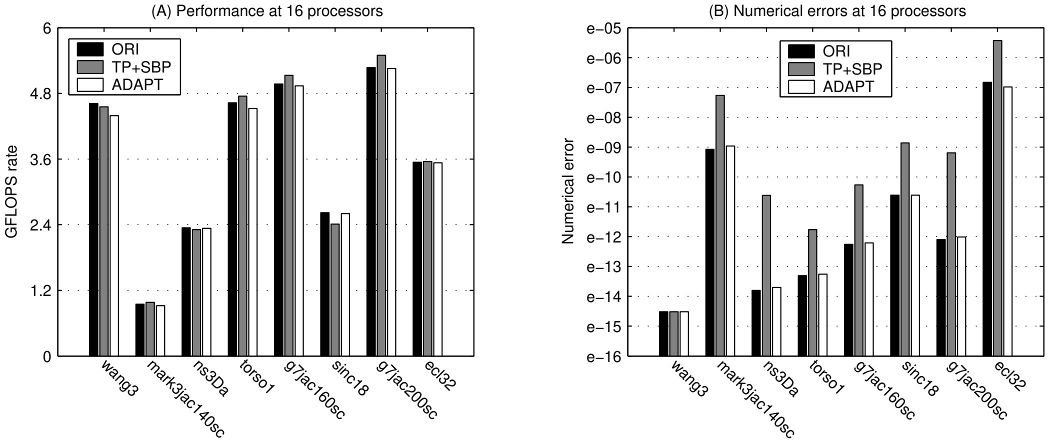

Fig. 12. Performance and numerical errors of static and adaptive approaches on the IBM Regatta using a shared memory-based MPI runtime system. Results are at 16 processors. Note that the Y-axis for figure (B) is in logarithmic scale.

0.2 for the experiments described next.

6.2 Experimental Evaluation on Runtime Application Adaptation

We evaluate the effectiveness of our sampling-based runtime application adaptation with different message passing platforms and input matrices. The evaluations are performed on three MPI platforms specified in Section 3: a PC cluster, an IBM Regatta running MPICH p4 device, and the IBM Regatta using shared memory message passing. To save space, here we only show results for the eight largest matrices (in terms of floating point operation counts) in our test collection. We compare our adaptive approach (denoted as *ADAPT*) with two static approaches: ORI and TP+SBP.

Figures 12, 13, and 14 illustrate the performance and numerical errors of static and adaptive approaches on Regatta/shmem, Regatta/MPICH, and the PC cluster respectively. Results in Figure 12(A) show that the communication reduction techniques (TP+SBP) provide little or no performance benefit on Regatta/shmem. ADAPT automatically disables these techniques for all test matrices and thus able to achieve similar numerical stability as ORI (shown in Figure 12(B)).

Figure 13(A) shows that the performance benefit of TP+SBP is quite pronounced for many input matrices on Regatta/MPICH. ADAPT discovers this through sampling and employs the communication-reduction pivoting techniques for all matrices except sinc18 and ecl32. We notice that the ADAPT performance is slightly inferior to TP+SBP even when the communication-reduction techniques are employed. This is due to the sampling overhead and that these techniques are not applied during the sampling phase.

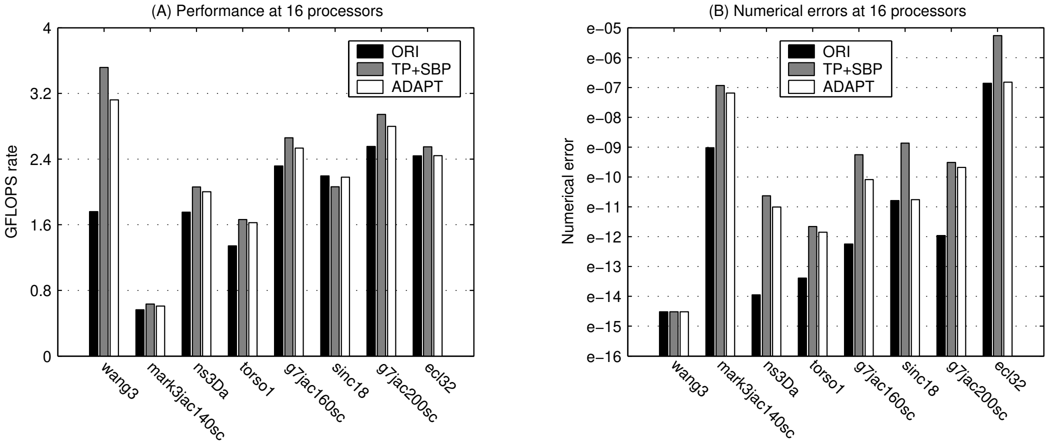

Fig. 13. Performance and numerical errors of static and adaptive approaches on the IBM Regatta using MPICH with the p4 device. Results are at 16 processors.

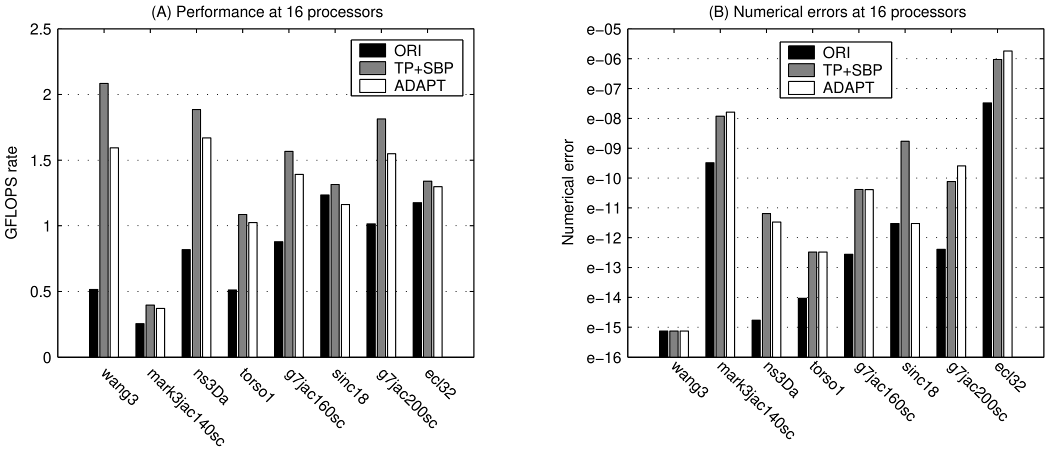

Fig. 14. Performance and numerical errors of static and adaptive approaches on the PC cluster. Results are at 16 processors.

Figure 14 (A) shows that the performance benefit of TP+SBP is substantial for all input matrices except sinc18 (15%) on the PC cluster. ADAPT correctly employs communication-reduction pivoting techniques for these matrices. The ADAPT performance is slightly inferior to TP+SBP, again due to the sampling overhead and that the communication-reduction techniques are not applied during the sampling phase.

In summary, our experiments find that the sampling-based application adaptation can estimate the potential performance saving of communication-reduction pivoting techniques and make appropriate decisions on whether to employ them for our test scenarios.

7 Related Work

Parallel sparse LU factorization has been extensively studied in the past [7,3,10,9,11,5,12,6]. Most existing solvers are deployed and evaluated on tightly coupled parallel computers with high message passing performance. Little attention has been paid on application performance on much slower message passing platforms.

Malard employed threshold pivoting to reduce inter-processor row interchanges for dense LU factorization [23]. Duff and Koster [24,25] and Li and Demmel [12] 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 novel matrix reordering technique to exploit large-grain parallelism in solving parallel sparse linear systems [34]. Although larger-grain parallelism typically results in less frequent inter-processor communications, their work only targets work-stealing style solvers on shared memory multiprocessors. It is not clear whether their technique can be useful for message passing-based solvers. Previous studies explored broadcasting/multicasting strategies (often tree-based) for distributing pivot columns or rows while achieving load balance [35,23]. In comparison, our work focuses on the performance on platforms with slow message passing 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 [36]. Hunold *et al.* proposed multilevel hierarchical matrix multiplication to improve the application performance on the PC cluster [37]. A recent work by Amestoy *et al.* considers hybrid scheduling with mixed (memory usage and FLOPS speed) equilibration objectives [13]. Our work in this paper addresses a different platform-related problem — adaptive parallel sparse LU factorization on platforms with different message passing performance.

8 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 15–460% on an Ethernet-connected 16-node PC cluster. Communication-reduction techniques may incur extra computation overhead and they may also slightly weaken numerical stability. Considering different tradeoffs of these techniques on different message passing platforms, this paper also proposes a sampling-based runtime application adaptation approach that automatically determines whether the communication-reduction techniques should be employed for given message passing platform and input matrix.

Given the high application porting costs and the increasing diversity of the available parallel computing platforms, it is desirable to construct self-adaptive applications that can automatically adjust themselves and perform well on different computing platforms. This is particularly challenging for irregular applications due to potential runtime data structure variations and irregular computation/communication patterns. Our work in this paper makes one step forward by tackling one such irregular application.

Software Availability

The implemented code that includes the communication-reduction techniques described in this paper is incorporated into a parallel sparse linear system solver (S^+ version 1.1). S^+ can be downloaded from the web [38].

References

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

[2] T. A. Davis, I. S. Duff, A Combined Unifrontal/multifrontal Method for Unsymmetric Sparse Matrices, ACM Trans. Math. Software 25 (1) (1999) 1–19.

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

[4] X. S. Li, Sparse Gaussian Elimination on High Performance Computers, Ph.D. thesis, Computer Science Division, EECS, UC Berkeley (1996).

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

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

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

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

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

[10] C. Ashcraft, R. G. Grimes, SPOOLES: An Object-oriented Sparse Matrix Library, in: Proc. of the 9th SIAM Conf. on Parallel Processing for Scientific Computing, San Antonio, Texas, 1999.

[11] P. R. Amestoy, I. S. Duff, J. Koster, J.-Y. L'Excellent, A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling, SIAM J. Matrix Anal. Appl. 23 (1) (2001) 15–41.

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

[13] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Hybrid Scheduling for the Parallel Solution of Linear Systems, Parallel Computing 32 (2) (2006) 136–156.

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

[15] J. J. Dongarra, J. D. Croz, S. Hammarling, R. Hanson, An Extended Set of Basic Linear Algebra Subroutines, ACM Trans. Math. Software 14 (1988) 18–32.

[16] C. Fu, X. Jiao, 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, 1997.

[17] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated Empirical Optimization of Software and the ATLAS Project, Parallel Computing 27 (1–2) (2001) 3–35.

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

[19] M. Cosnard, 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, 2000.

[20] A. R. Curtis, J. K. Reid, The Solution of Large Sparse Unsymmetric Systems of Linear Equations, J. Inst. Maths. Applics. 8 (1971) 344–353.

[21] I. Duff, Practical Comparisons of Codes for the Solution of Sparse Linear Systems, in: Sparse Matrix Proceedings, 1979, pp. 107–134.

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

[23] J. Malard, Threshold Pivoting for Dense LU Factorization on Distributed Memory Multiprocessors, in: Proc. the ACM/IEEE Conf. on Supercomputing, Albuquerque, NM, 1991, pp. 600–607.

[24] I. S. Duff, 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) (1999) 889–901.

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

[26] T. A. Davis, J. R. Gilbert, S. I. Larimore, E. G. Ng, A Column Approximate Minimum Degree Ordering Algorithm, ACM Trans. Math. Software 30 (3) (2004) 353–376.

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

[28] A. George, J. Liu, The Evolution of the Minimum Degree Ordering Algorithm, SIAM Review 31 (1989) 1–19.

[29] J. W. Demmel, J. R. Gilbert, X. S. Li, SuperLU Users' Guide (Oct. 2003).

[30] G. Marin, J. Mellor-Crummey, Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models, in: Proc. of the ACM SIGMETRICS, New York, NY, 2004, pp. 2–13.

[31] A. Snavely, L. Carrington, N. Wolter, Modeling Application Performance by Convolving Machine Signatures with Application Profiles, in: Proc. of the 4th IEEE Workshop on Workload Characterization, Austin, TX, 2001.

[32] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, T. von Eicken, LogP: Towards A Realistic Model of Parallel Computation, in: Proc. of the ACM Symp. on Principles and Practice of Parallel Programming, San Diego, CA, 1993, pp. 1–12.

[33] G. Rodriguez, R. Badia, J. Labarta, Generation of Simple Analytical Models for Message Passing Applications, in: Proc. of the 10th Euro-Par Parallel Processing Conf., Pisa, Italy, 2004.

[34] K. A. Gallivan, B. A. Marsolf, 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, Manchester, UK, 1994, pp. 419–430.

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

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

[37] S. Hunold, T. Rauber, G. Rünger, Multilevel Hierarchical Matrix Multiplication on Clusters, in: Proc. of the 18th ACM Conf. on Supercomputing, Saint-Malo, France, 2004, pp. 136–145.

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