Performance Analysis of Parallel Supernodal Sparse LU Factorization

Laura Grigori Xiaoye S. Li
Lawrence Berkeley National Laboratory, MS 50F-1650
One Cyclotron Road, Berkeley, CA 94720, USA.
email: {lgrigori, xsli}@lbl.gov

Abstract

We investigate performance characteristics for the LU factorization of large matrices with various sparsity patterns. We consider supernodal *right-looking* parallel factorization on a bi-dimensional grid of processors, making use of static pivoting. We develop a performance model and we validate it using the implementation in SuperLU_DIST, the real matrices and the IBM Power3 machine at NERSC. We use this model to obtain performance bounds on parallel computers, to perform scalability analysis and to identify performance bottlenecks. We also discuss the role of load balance and data distribution in this approach.

1 Introduction

A valuable tool in designing a scalable parallel algorithm is to analyze its performance characteristics for various classes of applications and machine configurations. Very often, good performance models reveal communication ineficiency and memory access contention that bound the overall performance. Modeling these aspects in detail can give insights into the performance bottlenecks and help improve the algorithm. The goal of this paper is to analyze performance characteristics and scalability for the LU factorization of large matrices with various sparsity patterns.

For dense matrices, the factorization algorithms have been shown to exhibit good scalability, like the algorithm in ScalaPACK [2, 4], where the efficiency can be approximately maintained as the number of processors increases when the memory requirements per processor are held constant.

However, for sparse matrices, the efficiency is much harder to predict since it largely depends on the sparsity pattern of the matrix. Several results exist in the literature [1, 6, 11], which were obtained for particular classes of matrices arising from the discretization of a physical domain. They show that factorization is not always scalable with respect to memory use. For sparse matrices resulting from two-dimensional domains, the best parallel algorithm lead to an increase of the memory at a rate of $O(P \log P)$ with increasing P [6]. It is worth mentioning that for matrices resulting from three-dimensional domains, the best algorithm is scalable with respect to memory size.

In this paper we use a simple and classical model that describes an ideal machine architecture in terms of processor speed, network latency and bandwidth. Using this machine model and several input characteristics (order of the matrix, number of nonzeros, etc), we concentrate on analysing the parallel runtime and efficiency of LU factorization of arbitrary sparse matrices. We consider supernodal *right-looking* parallel factorization on a bi-dimensional grid of processors, making use of static pivoting. This analysis allows us to obtain performance bounds on parallel computers, to perform scalability analysis, to identify performance bottlenecks and to discuss the role of load balance

and data distribution. We validate our analytical runtime model using the actual implementation in SuperLU_DIST [8], the real matrices and the IBM Power3 machine at NERSC.

The rest of the paper is organized as follows: Section 2 presents a background on the sparse LU factorization and the methodology used for its performance analysis. Section 3 introduces a performance model for the right-looking factorization, with its scalability analysis. The experimental results validating the performance model are presented in Section 4 and Section 5 draws the conclusions.

2 Background

Consider factorizing an unsymmetric and sparse $n \times n$ matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U. We discuss a parallel execution of this factorization on a bi-dimensional grid of processors. The matrix is partitioned into $N \times N$ blocks of submatrices using unsymmetric supernodes (columns of L with the same nonzero structure [3]). These blocks of submatrices are further distributed among a bi-dimensional grid $P_r \times P_c$ of P processors ($P_r \times P_c \leq P$) using a block cyclic distribution. With this distribution, a block at position (I, J) of the matrix will be mapped on the process at position $((I-1) mod P_r, (J-1) mod P_c)$ of the grid. $U(K, J)$ ($L(K, J)$) denotes a submatrix of U (L) at row block index K and column block index J.

The performance model we analyze for the sparse LU factorization is close to the performance model used for dense factorization algorithms in ScaLAPACK [4]. Processors have local memory and are connected by a network that provides each processor direct links with any of its 4 direct neighbours (mesh-like).

To simplify analysis and to make the model easier to understand, we make the following assumptions:

- We use one parameter to describe the processor flop rate, denoted γ, and we ignore communication collisions. We estimate the time for sending a message of m items between two processors as $\alpha + m\beta$, where α denotes the latency and β the inverse of the bandwidth.
- We approximate the cost of a broadcast to p processors by $\log p$ [4]. Furthermore, the LU factorization uses a pipelined execution to overlap some of the communication with computation, and in this case the cost of a broadcast is estimated by 2 [4].
- We assume that the computation of each supernode lies on the critical path of execution, that is the length of the critical path is N. We also assume that the load and the data is evenly distributed among processors. Later in Sections 3 and 4, we will provide the experimental data to show that these assumptions are reasonably realistic.

The parallel efficiency $E(N, P)$ for a problem of size N on P processors is defined as:

$$E(N, P) = \frac{1}{P} \frac{T_s(N)}{T(N, P)}$$

where $T(N, P)$ is the runtime of the parallel algorithm and $T_s(N)$ is the runtime of the best sequential algorithm. N can be the size of the input matrix, as in [4], or the amount of work, as in [6]. An algorithm scales well if the efficiency is an increasing function of N/P, the problem size per processor.

3 Right-looking sparse LU factorization

In the following we estimate the runtime of the right-looking sparse LU factorization. We start with an evaluation of the sequential runtime. We than analyze the parallel runtime, considering a rectangular grid $P = P_r \times P_c$ and a square grid of processors $P = \sqrt{P} \times \sqrt{P}$.

3.1 Runtime estimation

Algorithm 1 describes a right-looking factorization and Figure 1 illustrates the respective execution on a rectangular grid of processors. This algorithm loops over the N supernodes. In the K-th iteration, the first $K-1$ block columns of L and block rows of U are already computed. At this iteration, first the column of processors owning block column K of L factors this block column $L(K:N,K)$; second, the row of processors owning block row K of U performs the triangular solve to compute $U(K,K+1:N)$; and third, all the processors update the trailing matrix using $L(K+1:N,K)$ and $U(K,K+1:N)$. This third step requires most of the work and also exhibits most of the parallelism in the right-looking approach.

Algorithm 1 Right-looking factorization

for $K := 1$ to N **do**
  Factorize block column $L(K:N,K)$
  Perform triangular solves: $U(K,K+1:N) := L(K,K)^{-1} \times A(K,K+1:N)$
  for $J := K+1$ to N with $U(K,J) \neq 0$ **do**
    for $I := K+1$ to N with $L(I,K) \neq 0$ **do**
      Update trailing submatrix:
      $A(I,J) := A(I,J) - L(I,K) \times U(K,J)$
    end for
  end for
end for

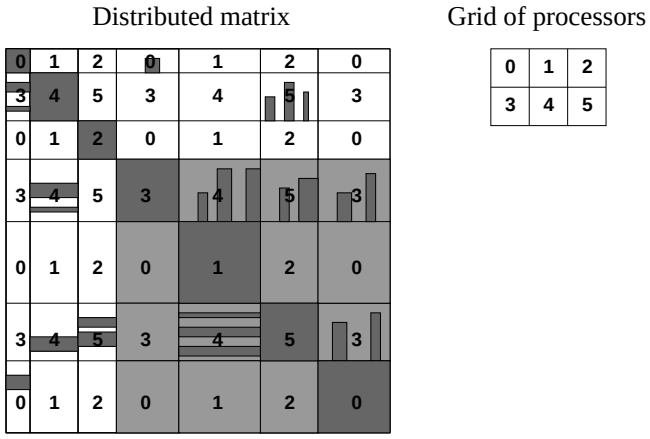

Figure 1: Illustration of parallel right-looking factorization

We use the following notation to estimate the runtime to factorie an $n \times n$ matrix:

- c_k - number of off-diagonal elements in each column of block column K of L;

- r_k - number of off-diagonal elements in each row of block row K of U;
- $nnz(L)$ - number of nonzeros in the off-diagonal blocks of L;
- $nnz(U)$ - number of nonzeros in the off-diagonal blocks of U;
- $M = 2\sum_{k=1}^{n} c_k r_k$ - total number of flops in the trailing matrix update, counting both multiplications and additions;
- $F = nnz(L) + M$ - total number of flops in the factorization.

With the above notation, the sequential runtime can be estimated as:

$$T_s = nnz(L)\gamma + M\gamma = F\gamma \tag{1}$$

To analyze the performance of the algorithm, we consider the elapsed time of each iteration K, and then sum over all the iterations, thus approximating the total runtime. This assumes that each iteration lies on the critical path of execution. For sparse matrices, it is possible that only some of the processors participate in each iteration; it can even happen that two different iterations are executed simultaneously by two disjoint sets of processors. In that case, the structure of the matrix must be analyzed in order to determine the computation on the critical path, thus obtaining a better estimate of the runtime.

Length of critical path Before continuing with the runtime analysis, we first compute the length of the critical path and verify that using N in our assumption is valid. Algorithm 2 computes the length of the critical path. $lcpRL$ denotes the overall length of the critical path, while $lcpProcs$ denotes the length of the critical path for each processor. At each iteration, we determine the set of processors owning a block which is updated at this iteration and we determine the processor with the longest critical path l. During this iteration the processors will need to synchronize at the broadcast step, which is good reason to adjust to l the length of the critical path of all the processors in this set, and than increment it by 1.

Algorithm 2 Compute length of critical path in right-looking factorization

```
for p := 1 to P do
  lcpProcs[p] := 0
end for
for K := 1 to N do
  l := 0
  for p := 1 to P such that p owns a block to be modified in this iteration do
    l := max(l, lcpProcs[p])
  end for
  for p := 1 to P such that p owns a block to be modified in this iteration do
    lcpProcs[p] := l + 1;
  end for
end for
lcpRL := 0;
for p := 1 to P do
  lcpRL := max(lcpRL, lcpProcs[p])
end for
```

Table 1 shows the length of the critical path computed by this algorithm as a fraction of the order of the supernodal matrix N ($lcpRL/N$). Notice that with one exception, the lengths of the

	P = 4	P = 16	P = 32	P = 64	P = 128
onetone1	0.89	0.85	0.82	0.76	0.73
twotone	0.92	0.91	0.90	0.90	0.89
wang4	1.00	1.00	0.99	0.99	0.98
af23560	1.00	1.00	1.00	0.99	0.99
venkat01	1.00	1.00	0.99	0.98	0.97
rma10	1.00	0.98	0.97	0.96	0.94
ecl32	0.63	0.63	0.62	0.62	0.62
ir.K-sM	1.00	1.00	1.00	0.99	0.99
bbmat	1.00	1.00	1.00	1.00	1.00
inv-extr1	1.00	1.00	1.00	1.00	0.99
ex11	1.00	1.00	1.00	1.00	1.00
fidapm11	1.00	1.00	1.00	1.00	1.00
dds15.K-sM	1.00	1.00	1.00	1.00	1.00
mixingtank	1.00	1.00	1.00	1.00	1.00
dds.quadratic.K-sM	1.00	1.00	1.00	1.00	1.00

Table 1: Fraction of the iteration steps which are on the critical path (lcpRL/N).

critical paths of all the matrices are very close to N. The exception is the matrix ecl32, for which the length of the critical path represents only 60% of the size of the supernodal matrix N. We also observe that with an increasing number of processors, the length of the critical path is subject to very small decrease (almost none). Thus, assuming a critical path of length N is well justified.

Rectangular grid of processors We consider now in detail the K-th iteration of algorithm 1, divided into 5 steps. We assume each processor in the column processors owning block column K gets $s \cdot c_k/P_r$ elements, where $s \cdot c_k$ is the number of nonzeros in the block column K and P_r is the number of the column processors. Block row K of U is distributed in a similar manner.

1. Factor supernode K formed by s columns: compute the diagonal block $L(K,K)$; broadcast the diagonal block $L(K,K)$ to the column of processors that holds supernode K; scale and perform rank-1 update of the trailing submatrices $L(K+1:N,K)$.

$$\frac{2}{3}s^3\gamma + \alpha + s^2\beta + \frac{c_k}{P_r}s\gamma + \left(2\frac{c_k}{P_r}\sum_{i=1}^{s-1} i\right)\gamma$$

2. Send supernode $L(:,K)$ to the processors who need it within the rows of processors. Using a pipelined execution, this is overlapped with the factorization of supernode $K+1$:

$$2\left(\alpha + \frac{c_k}{P_r}s\beta\right)$$

3. Parallel triangular solves $U(K,K+1:N) = L^{-1}(K,K)A(K,K+1:N)$:

$$s(s+1)\frac{r_k}{P_c}\gamma$$

4. Send block row $U(K, K+1:N)$ to the processors who need it within the columns of processors:

$$\log P_r \left(\alpha + \frac{r_k}{P_c} s\beta\right)$$

5. Update trailing matrix $A(K+1:N, K+1:N)$:

$$2s \frac{c_k}{P_r} \frac{r_k}{P_c} \gamma$$

The total computation time over a rectangular grid of processors $T(N, P_r \times P_c)$ can be expressed as a sum over the number of iterations of the previously presented steps. In a way similar to [2], we neglect the time to factorize supernode K (step 1), and we also neglect the time of the parallel triangular solves (step 3), considering that the broadcast following these steps dominates the time of these operations.

$$T(N, P_r \times P_c) \approx \frac{M}{P}\gamma + (2N + N \log P_r)\alpha + \left(2\frac{nnz(L)}{P_r} + \frac{nnz(U)}{P_c} \log P_r\right)\beta$$

Square grid of processors On a square grid of processors, $P = \sqrt{P} \times \sqrt{P}$, the above equation becomes:

$$T(N, \sqrt{P} \times \sqrt{P}) \approx \frac{M}{P}\gamma + (2N + \frac{1}{2} N \log P)\alpha + \frac{(2nnz(L) + \frac{1}{2} nnz(U) \log P)}{\sqrt{P}}\beta$$

The first term represents the parallelization of the rank-s update. The second term represents the number of broadcasting messages. The third term represents the volume of communication overhead.

3.2 Scalability analysis

Let us examine scalability on a square grid of processors of size P, where the efficiency of the right-looking algorithm is given by the following formula:

$$\epsilon(N, \sqrt{P} \times \sqrt{P}) = \frac{T_s(N)}{PT(N, \sqrt{P} \times \sqrt{P})} \quad (2)$$

$$\approx \left[\frac{M}{F} + \frac{NP \log P}{F}\frac{\alpha}{\gamma} + \frac{(2nnz(L) + nnz(U) \log P)\sqrt{P}}{F}\frac{\beta}{\gamma}\right]^{-1} \quad (3)$$

The interesting question here is which of the three terms dominates efficiency (depending on the size of the matrix and the sparsity of the matrix). The preliminary remark is that, for very large and dense problems (N, F and $nnz(L+U)$ large), the first term significantly affects the parallel efficiency.

For the other cases, we can compare the last two terms to determine which one is dominant. That is, if we ignore 2 and $\log P$ factors in the third term, we have to compare $\sqrt{P}\frac{\alpha}{\beta}$ with $\frac{nnz(L+U)}{N}$. Assuming that the network's latency-bandwidth product is given ($\frac{\alpha}{\beta}$), we can determine if the ratio of the latency to the time per flop (α/γ term) or the ratio of the inverse of the bandwidth to the flop rate (β/γ term) dominates efficiency. Overall, we can make the following statements for different cases:

Case 1 For sparser problems ($\sqrt{P}\frac{\alpha}{\beta} > \frac{nnz(L+U)}{N}$), the α/γ term dominates efficiency.

Case 2 For denser problems ($\sqrt{P}\frac{\alpha}{\beta} < \frac{nnz(L+U)}{N}$), the β/γ term dominates efficiency.

Case 3 For problems for which $\frac{nnz(L+U)}{N}$ is close to $\frac{\alpha}{\beta}$, the β/γ term can be dominant on smaller number of processors, and with increasing number of processors the α/γ term can become dominant.

Note that even for Case 2, the algorithm behaviour varies during the factorization: at the begining of the factorization, where the matrix is generally sparser and the messages are shorter, α/γ term dominates the efficiency, while at the end of the factorization where the matrix becomes denser, β/γ term dominates the efficiency.

Let us now consider matrices in Case 2. For these matrices, in order to maintain a constant efficiency, $\frac{F}{nnz(L+U)}$ must grow proportionally with $\sqrt{P}$. For a scalable algorithm in terms of memory requirements, P should grow with the memory $nnz(L+U)$. Thus, for problems for which the condition $F \propto nnz(L+U)^{3/2}$ is satisfied, when the latency and the variation of $\log P$ are ignored, the efficiency can be approximately maintained constant. In reality, even for these matrices, α/γ term as well as $\log P$ factor will induce efficiency degradation.

Matrices with N unknowns satisfying this condition are matrices arising from discretization of Laplacian operator on three-dimensional finite element grids. Using nested dissection, the number of nonzeros of such a sparse matrix is on the order of $O(N^{4/3})$ while the amount of work is on the order of $O(N^2)$. Maintaining a fixed efficiency requires that the number of processors P grows proportionally with $N^{4/3}$, the size of the matrix. In essence, the efficiency for these problems can be approximately maintained if the memory requirement per processor is constant. Note that α/γ term grows with $N^{1/3}$, which also contributes to efficiency loss.

4 Experimental results

In the previous sections we presented an analytical estimation of the parallel execution time for right-looking sparse factorization algorithm, making several simplifying assumptions that will most certainly induce deviations from the results obtained with actual implementation runs. The aim of this section is to compare the analytical results obtained against experimental results performed on a IBM Power3 machine at NERSC, with real world matrices.

We have used a test set of large matrices from various application domains. These matrices and their characteristics are presented in Table 2 which includes the matrix order, the number of supernodes N, the number of nonzeros in the matrix A, the number of nonzeros in the factors L and U, and the number of floating-point operations. The matrices are ordered according to the last column, $nnz(L+U)/N$, which we use as a measure of the sparsity of the matrix.

The first goal is to show how the experimental results support the analytical performance model developed in Section 3. For this, we use the analytical performance model to predict the speedup that each matrix should attain with an increasing number of processors. Then we plot the predicted speedup against the speedup obtained by SuperLU_DIST. The plots in Figure 2 display these results, where the predicted speedup for each matrix M is denoted by Mp, and the actually obtained (measured) speedup is denoted by Mm.

As the plots show, the analytical performance model predicts well the performances on a small number of processors, while the predicted speedup starts to deviate above the measured speedup with an increase in the number of processors. This is because on a smaller number of processors

Matrix	Order	N	$nnz(A)$	$nnz(L+U)$ $\times 10^6$	$Flops(F)$ $\times 10^9$	$nnz(L+U)/N$ $\times 10^3$
onetone1	62424	18461	341088	3.15	1.35	0.16
twotone	120750	64726	1224224	10.7	7.32	0.16
wang4	26064	15620	177196	10.5	8.78	0.67
af23560	23560	10440	484256	11.8	5.41	1.13
venkat01	62424	9454	1717792	11.8	2.41	1.25
rma10	46835	6427	2374001	9.36	1.61	1.45
ecl32	51993	25827	380415	41.4	60.45	1.60
ir.K-sM	186230	46431	2910202	113.1	147.87	2.43
bbmat	38744	11212	1771722	35.0	25.24	3.12
inv-extr1	30412	6987	1793881	28.1	27.26	4.02
ex11	16614	2033	1096948	11.5	5.60	5.65
fidapm11	22294	3873	623554	26.5	26.80	6.84
dds15.K-sM	834575	100491	13100653	877.63	1578.86	8.73
mixingtank	29957	4609	1995041	44.7	79.57	9.69
dds.quadratic.K-sM	380698	33282	15844364	642.98	2121.74	19.31

Table 2: Benchmark matrices.

the assumptions made are rather realistic, but on a larger number of processors the assumptions are deviating from reality, and this will degrade the scalability of the algorithm.

We can still remark that the analytical performance model gives indications on the behaviour of the right-looking factorization. This aspect can be better observed with the help of Figure 3, where we plot four curves, corresponding to $4, 16, 32$ and 64 processors respectively. Each curve plots the measured speedup for all the matrices in our test set against the efficiency predicted by our analytical model (Eqn.(3)). Since the efficiency is defined as $E = \frac{T_s}{PT_p}$ and the speedup is defined as $S = \frac{T_s}{T_p}$, for the same number of processors, each curve should increase linearly with the predicted efficiency. We can observe this trend, except for matrix *ecl*32 for which the analytical performance model is inaccurate. This is because the model assumption that the length of the critical path is N is not quite right, see Table 1 in Section 3.

The second goal of the experiments is to study the actual efficiency case by case for all the test matrices. One approach to do this is to observe how the factorization runtime degrades as the number of processors increases for different matrices. For this we report in Table 3 the runtime in seconds of the SuperLU_DIST solver for all the matrices in our test set. These results illustrate that good speedups can be obtained on a small number of processors and show how efficiency degrades on a larger number of processors. As one would expect, the efficiency degrades faster for problems of smaller size (number of flops smaller), and slower for larger problems. We expect that for even larger problems, the algorithm will scale to a larger number of processors.

We now examine how the actual model parameters (sparsity, α, β, and γ) affect the performance. On the IBM Power3, the latency observed is 8.0 usecs and the bandwidth for our medium size of messages is 494 MB/s [12], and thus the ratio of latency to inverse of bandwidth is $\alpha/\beta \approx 4 \times 10^3$. The last column in Table 2 shows that the algorithm's efficiency for some of the matrices is clearly dominated by the α/γ term (Case 1), like matrices onetone1, twotone, and wang4. For matrices dds.quadratic.K-sM, mixingtank, dds15.K-sM, fidapm11, inv-extr1, the efficiency is significantly influenced by the β/γ term (Case 2).

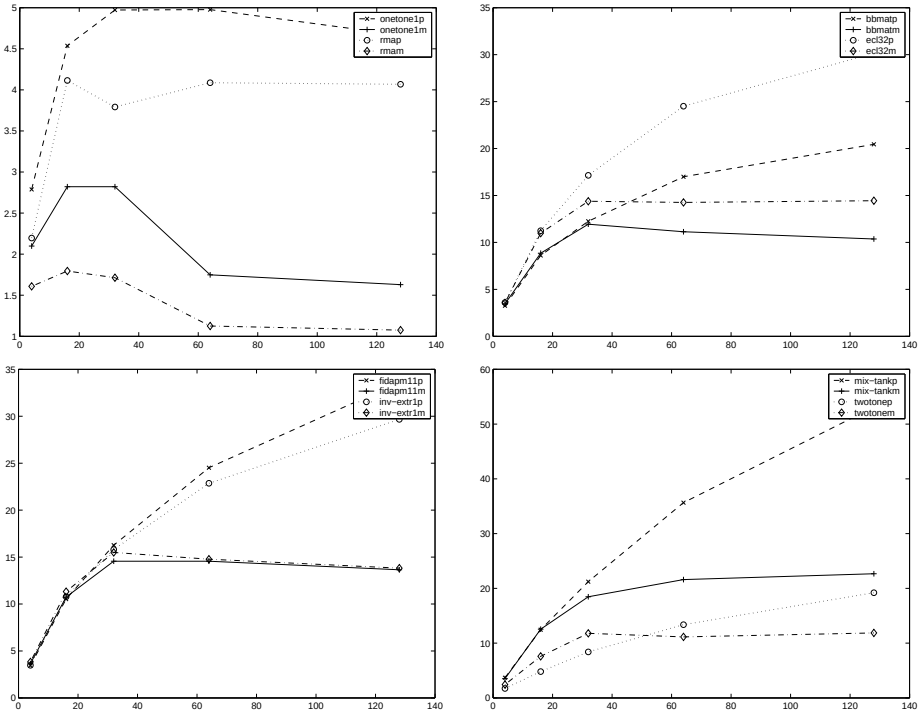

Figure 2: Predicted speedup by our analytical performance model versus obtained speedup by SuperLU_DIST for several of the matrices in our test set. For each matrix M, the predicted speedup is denoted by Mp, while the measured speedup is denoted by Mm.

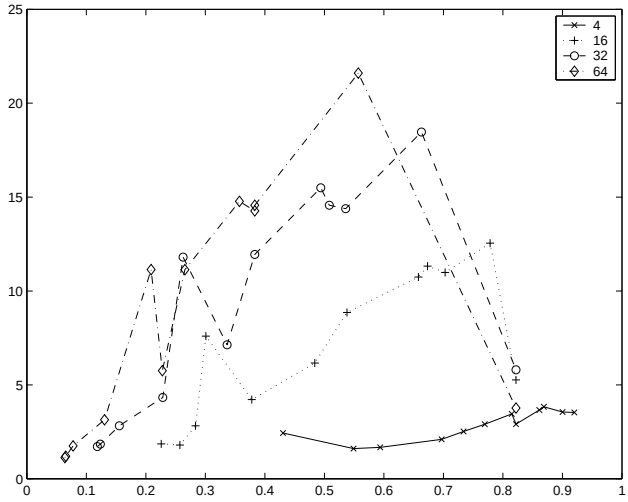

Figure 3: Right-looking: speedup versus estimated efficiency for all the matrices in the test set

	P=1	P = 4	P = 16	P = 32	P = 64	P = 128
onetone1	5.98	2.85	2.12	2.12	3.42	3.67
twotone	183.12	75.27	24.12	15.52	16.44	15.44
wang4	14.79	5.09	2.81	2.55	3.93	3.99
af23560	9.95	3.95	2.36	2.30	3.17	3.38
venkat01	4.75	2.84	2.56	2.58	3.96	3.86
rma10	3.41	2.12	1.90	1.99	3.03	3.17
ecl32	104.27	29.34	9.49	7.25	7.31	7.22
ir.K-sM	245.61	73.61	28.17	22.45	23.60	23.69
bbmat	67.37	19.50	7.61	5.64	6.05	6.50
inv-extr1	73.13	19.08	6.46	4.72	4.95	5.29
ex11	9.49	3.27	1.54	1.33	1.65	2.07
fidapm11	51.99	14.21	4.84	3.57	3.47	3.81
dds15.K-sM			810.24	145.88	126.06	133.64
mixingtank	119.45	33.87	9.52	6.47	5.53	5.27
dds.quadratic.K-sM			212.33	123.59	90.62	75.79

Table 3: Runtimes (in seconds) for right-looking factorization on 2D grid of processors

Matrices af23560 and ex11 have an approximately equal number of flops, and almost similar runtimes on one processor. We observe that efficiency degrades faster for af23560 than for ex11. This is because the efficiency of af23560 is mostly influenced by the α/γ term (Case 1), while the efficiency of ex11 is mainly influenced by the β/γ term (Case 2), and thus its performance degrades slower than for af23560.

The third goal of our experiments is to analyze the different assumptions we have made during the developement of the performance model. Consider again the efficiency formula 3 obtained in Section 3. For the first term, we assume that the load is evenly distributed among processors, while for the third term we assume that the data is evenly distributed among processors. Note that the second term is independent of these assumptions.

We now analyze the distribution of the load as well as of the data among processors for all the matrices in our test set. The first term assumes a well balanced computation of the rank-s update on P processors. We now measure the load balance, denoted by LB. For this we define the entire load F to be the number of floating point operations to factorize the matrix. We then compute the load lying on the critical path F_{CP} by adding at each iteration the load of the most loaded processor in this iteration. More precisely, consider f_{pi} being the load of processor p at iteration i (number of flops performed by this processor at iteration i). Then $F_{CP} = \sum_{i=1}^{N} \max_{p=1}^{P} f_{pi}$. The load balance factor is computed as $LB = \frac{F_{CP}P}{F}$. In other words, LB is the load of heaviest processors lying on the critical path divided by the average load per processor. The closer this factor approaches 1, the better is the load balance. Contrary to the "usual" way, when the load balance factor is computed as the average load divided by the maximum load among all the processors, a more precise computation of the load balance is obtained.

The results are presented in table 4, in the rows denoted by LB. We can observe that the workload distribution is good for large matrices on a small number of processors. But it can suffer an important degradation for some of the matrices like rma10 or venkat01, where the load balance can degrade by a factor of 2 when increasing the processor number by a factor of 2. Consequently,

efficiency will also suffer an important degradation.

The third term reflects our data distribution assumption. We suppose that when a block column K of size sc_k is distributed among P_r processors, each processor will own $\frac{sc_k}{P_r}$ elements. To verify this assumption we compute a data distribution factor as follows. For each column block K we determine the processor owning the maximum number of nonzero elements of this block column. We then sum over all the block columns and we denote by D_{CP} this sum. We denote by D the number of nonzeros in the factor L. Then we compute the distribution factor DB as being $DB = \frac{D_{CP}P_r}{D}$. That is, we divide the amount of data lying on the critical path by the average number of nonzeros on each processor. The results are displayed in table 4, rows DB. We can see that a good data distribution is obtained for almost all the matrices. Similar results were observed for the data distribution of U

These experiments show that a 2D distribution of the data on a 2D grid of processors leads to a balanced distribution of the data. For some matrices, it also leads to a balanced distribution of the load among processors. For these matrices load imbalance is not the main factor affecting efficiency. But for some of the matrices we have however observed a poor load balance on large number of processors. For example, matrix *bbmat* has a load balance factor of 4.88 on 128 processors. Moreover, when we look at different steps of factorization, we observe that the load balance factor can go up to 7. So even when the overall load balance is not very poor, we can still have a severe load imbalance at different steps of the factorization. We conclude that on large number of processors, load imbalance significantly contributes to efficiency degradation.

5 Conclusions

In this paper we have analyzed a performance model for the sparse right-looking LU factorization algorithm and we have validated this model using the SuperLU_DIST solver, real world matrices and the IBM Power3 machine at NERSC.

Using this model, we first analyzed the efficiency of this algorithm with increasing number of processors and problem size. We concluded that for matrices satisfying a certain relation (namely $F \propto nnz(L+U)^{3/2}$) between their problem size and their memory requirements, the algorithm is scalable with respect to memory use. This relation is satisfied by matrices arising from 3D model problems. For these matrices the efficiency can be approximately maintained constant when the number of processors increases and the memory requirements per processor is constant. But for matrices arising from 2D model problems, the algorithm is not scalable with respect to memory use [6].

Secondly, we analyzed the efficiency of this algorithm for fixed problem size and increasing number of processors. Using the matrices with various sparsity patterns in our test set, we observed that good speedups can be obtained on smaller number of processors. On larger number of processors, the efficiency degrades faster for sparser problems which are more sensitive to the latency of the network. A 2D distribution of the data on a 2D grid of processors lead to a balanced distribution of the data. It also lead to a balanced distribution of the load on smaller number of processors. But the load balance is usually poor on larger number of processors. We believe that load imbalance and insufficient amount of work F relative to communication overhead are the main reasons for poor efficiency on large number of processors.

Two main performance improvements can be made to the current right-looking factorization. The first comes from algorithmic improvement. We can use a better matrix to processor mapping, like the subtree-to-subcube mapping [6] or the proportinal mapping [10]. These mappings reduce the number of messages and the overall communication overhead. The second comes from

		P = 4	P = 16	P = 32	P = 64	P = 128
onetone1	LB	1.23	1.92	2.43	3.28	5.08
	DB	1.14	1.44	1.44	2.01	2.01
twotone	LB	2.30	3.14	3.35	3.69	4.11
	DB	1.52	1.87	1.87	2.24	2.24
wang4	LB	1.14	1.45	1.80	2.26	3.19
	DB	1.08	1.23	1.24	1.56	1.56
af23560	LB	1.28	2.05	2.92	4.21	6.67
	DB	1.12	1.40	1.40	1.97	1.97
venkat01	LB	1.52	3.25	5.18	8.48	14.92
	DB	1.25	1.81	1.81	2.90	2.90
rma10	LB	1.66	2.75	5.62	9.13	16.07
	DB	1.27	1.80	1.79	2.86	2.86
ecl32	LB	1.09	1.28	1.52	1.80	2.37
	DB	1.05	1.14	1.14	1.35	1.35
ir.K-sM	LB	1.13	1.42	1.71	2.09	2.88
	DB	1.07	1.22	1.22	1.52	1.52
bbmat	LB	1.21	1.75	2.35	3.17	4.88
	DB	1.08	1.28	1.28	1.68	1.68
inv-extr1	LB	1.14	1.42	1.87	2.45	3.51
	DB	1.07	1.19	1.19	1.54	1.54
ex11	LB	1.27	1.90	2.58	3.53	5.32
	DB	1.11	1.34	1.34	1.79	1.79
fidapm11	LB	1.15	1.46	1.83	2.31	3.13
	DB	1.07	1.20	1.20	1.49	1.49
dds15.K-sM	LB	1.15	1.52	1.89	2.36	3.29
	DB	1.08	1.26	1.26	1.60	1.60
mixingtank	LB	1.08	1.25	1.43	1.63	2.04
	DB	1.04	1.13	1.13	1.30	1.30
dds.quadratic.K-sM	LB	1.08	1.26	1.47	1.73	2.27
	DB	1.06	1.18	1.18	1.43	1.43

Table 4: Load and data distribution for right-looking factorization on a 2D grid of processors

the underlying system improvement, in particular reducing the communication latency for small messages. This is because the algorithm incurs many small messages involving MPI point-to-point and broadcast operations, and so is very sensitive to latency, especially for sparser problems.

References

[1] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorization algorithms. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 159–191. Springer Verlag, 1994.

[2] J. Choi, J. Demmel, D. I., J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. *LAPACK Working Note 95*, 1995.

[3] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[4] J. K. Dongarra, R. A. van de Geijn, and D. W. Walker. Scalability Issues Affecting the Design of a Dense Linear Algebra Library. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994.

[5] J. R. Gilbert and J. W. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Anal. Appl.*, 14(2):334–352, April 1993.

[6] A. Gupta, G. Karypis, and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[7] S. M. Hadfield and T. A. Davis. Potential and Achievable Parallelism in Unsymmetric-Pattern Multifrontal LU Factorization. Technical Report TR-94-027, University of Florida, 1994.

[8] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software.* To appear. Also Technical Report LBNL-49388.

[9] L. S. Ostrouchov, M. T. Heath, and C. H. Romine. Modeling Speedup in Parallel Sparse Matrix Factorization. Technical Report ORNL/TM-11786, Oak Ridge National Laboratory, 1991.

[10] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM Journal on Scientific Computing*, pages 1253–1257, 1993.

[11] R. Schreiber. Scalability of sparse direct solvers. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 191–211. Springer Verlag, 1994.

[12] A. Wong. Private communication. Lawrence Berkeley National Laboratory, 2002.