Towards an Accurate Performance Modeling of Parallel Sparse Factorization

Laura Grigori Xiaoye S. Li

May 26, 2006

Abstract

We present a performance model to analyze a parallel sparse LU factorization algorithm on modern cached-based, high-end parallel architectures. Our model characterizes the algorithmic behavior by taking account the underlying processor speed, memory system performance, as well as the interconnect speed. The model is validated using the SuperLU_DIST linear system solver, the sparse matrices from real applications, and an IBM POWER3 parallel machine. Our modeling methodology can be easily adapted to study performance of other types of sparse factorizations, such as Cholesky or QR.

1 Introduction

Finding the solution of sparse linear systems of equations by a direct method is at the heart of many scientific and engineering applications and its performance has an important impact on the application's overall performance. There has been much research dedicated to efficient parallel implementation of the sparse LU factorization. Although the factorization has been made more and more scalable, loss of performance is often observed with increasing number of processors, and more so with very sparse and unsymmetric problems. It is difficult to predict the performance of a parallel sparse factorization, which largely depends on the sparsity of the matrix, and the sparsity patterns vary with applications. Several analytical results exist in the literature [3, 8, 12], which were obtained for particular classes of matrices arising from the discretization of certain PDEs.

In our previous work we have considered a supernodal right-looking parallel factorization on two dimensional grids of processors, using static pivoting [7]. We have derived an analytical estimation of the parallel runtime for arbitrary input matrices. This was obtained by considering a simple machine architecture and by making several simplified assumptions on the parallel algorithm. We assumed that the floating-point operations are performed at a constant rate throughout the factorization, and that the messages are transferred at a fixed bandwidth value independent of their sizes. We also assumed that the load and the data is evenly distributed among processors. The analytical model allowed us to obtain performance upper bounds on parallel computers, to perform scalability analysis, and to derive a relation between parallel runtime and matrix sparsity. We used the SuperLU_DIST [11] solver to compare its actual runtime with the model's predicted speed, and observed that the speedup predicted by the analytical performance model started to deviate above the measured speedup of SuperLU_DIST with increasing number of processors.

In this paper we present a much refined model that removes the unrealistic assumptions. In addition, our new model takes account not only the speeds of CPU and interconnect network, but also the memory system performance. We will show that the new model can accurately predicit the actual performance of the code. First, we develop a runtime model for the sequential algorithm. Instead of estimating the runtime by counting the floating-point operations alone and assuming that they are performed at a constant speed, we also evaluate the cost of memory accesses, because in a sparse factorization, a nontrivial amount of time is often spent in streaming through data. The cost of memory accesse is different when the data is already in cache or when it has to be fetched from

main memory. Thus, different latencies are incurred when the data are fetched at different levels of the memory hierarchy. Second, for the parallel runtime, we include the classical latency-bandwidth communication cost model, but with varying bandwidth values which were calibrated off-line based on the message size distribution throughout the factorization. We validate our performance model using the right-looking factorization algorithm implemented in SuperLU_DIST, the representative matrices from several application and an IBM POWER3 parallel machine.

The rest of the paper is organized as follows. Section 2 introduces a performance analysis model for the right-looking sparse LU factorization. Section 3 analyzes the performance of the numerical kernels. Section 4 describes our methodology for estimating the number of cache misses. Section 5 describes a performance model that estimates the execution time of the sequential and parallel factorization. The results validating the performance model are presented in Section 6 and Section 7 draws the conclusions.

2 Background

Algorithm 1 presents a right-looking algorithm that factorizes a sparse unsymmetric $n \times n$ matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U. The matrix is partitioned into $N \times N$ blocks of submatrices using unsymmetric supernodes (columns of L with the same nonzero structure). The algorithm loops over N supernodes. At the k-th step, the first $k-1$ block columns of L and block rows of U are already computed. Now, the block column $L(k:N,k)$ is factored, the triangular solves are performed to compute $U(k,k+1:N)$, and the trailing matrix is updated using $L(k+1:N,k)$ and $U(k,k+1:N)$. The last 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
  (1) Factorize block column L(k:N, k)
  (2) Perform triangular solves: U(k, k+1:N) := L(k,k)^{-1} × A(k, k+1:N)
  for j := k+1 to N with U(k,j) ≠ 0 do
    for i := k+1 to N with L(i,k) ≠ 0 do
      (3) Update trailing submatrix:
      A(i,j) := A(i,j) − L(i,k) × U(k,j)
    end for
  end for
end for
```

The parallel algorithm in SuperLU_DIST uses a data distribution on a two dimensional process grid and a look-ahead technique to overlap communication and computation, as shown in Algorithm 2. Figure 1 illustrates the data distribution on a 2D grid of six processors. The blocks of the matrix partitioned by supernodes are distributed among $P_r \times P_c$ $(= P)$ processors using a block cyclic distribution. A block at position (i,j) of the matrix $(0 \le i,j < N)$ will be mapped on the process at position $(i \bmod P_r, j \bmod P_c)$ 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. We use $(L+U)(k,j)$ to denote a block of L or U at block row k and block column j.

The look-ahead technique in Algorithm 2 appears in the update of the trailing matrix. At step k, we first update the block column $(k+1)$ and perform the block factorization (steps (3.1) and (1.2)). The column processes owning block column $k+1$ then send it to all the processes that need it using nonblocking sends, while the potential receivers post nonblocking receives (steps (a.3) and (a.4)). Afterwards, the rest of the trailing matrix (block columns $k+2:N$)) is updated (step (3.2)), which overlaps with the communication in steps (a.3) and (a.4). Steps (c.1) and (c.2) contains the matching waits for the nonblocking send and receive from steps (a.3) and (a.4).

The performance model we developed has the following characteristics:

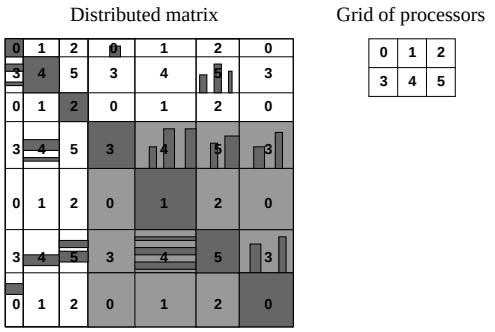

Figure 1: Illustration of parallel right-looking factorization

Algorithm 2 Parallel right-looking factorization in SuperLU_DIST

let $myPE$ be my process number
if $myPe$ owns blocks of the block column $L(:,1)$ **then**
  (1.1) Factorize block column $L(:,1)$
  (a.1) Nonblocking Send $L(:,1)$ to processes in my row that need it
else
  (a.2) Nonblocking Receive $L(:,1)$ from a process in my row
end if
for block $k := 1$ to N **do**
  if $myPe$ owns blocks of the block column $L(:,k)$ **then**
    (c.1) Wait for posted send of $L(:,k)$ to complete
  else
    (c.2) If $myPe$ needs $L(:,k)$, wait for posted receive of $L(:,k)$ to complete
  end if
  if $myPe$ owns blocks of the block row $U(k,:)$ **then**
    (2) Perform triangular solves: $U(k,k+1:N) := L(k,k)^{-1} \times A(k,k+1:N)$
    (b.1) Send $U(k,:)$ to processes in my column that need it
  else
    (b.2) Receive $U(k,:)$ from a process in my column if I need it
  end if
  if $(k+1) \leq N$ **then**
    if $myPe$ owns blocks of the block column $L(:,k+1)$ **then**
      (3.1) Update $A(:,k+1) := A(:,k+1) - L(:,k) \times U(k,:)$
      (1.2) Factorize block column $L(:,k+1)$
      (a.3) Nonblocking Send $L(:,k+1)$ to processes in my row that need it
    else
      (a.4) Nonblocking Receive $L(:,k+1)$ from a process in my row if $myPe$ needs it
    end if
  end if
  for $j := k+2$ to N with $U(k,j) \neq 0$ **do**
    for $i := k+2$ to N with $L(i,k) \neq 0$ **do**
      if $myPe$ owns block $A(i,j)$ **then**
        (3.2) Update trailing submatrix: $A(i,j) := A(i,j) - L(i,k) \times U(k,j)$
      end if
    end for
  end for
end for

- Our model differentiates which part of the code is spent time mostly on floating-point operations (i.e., flops-bound) and which part of the code is spent time mostly on streaming through data (i.e., memory-bound). Flops-bound means that the number of cycles spent in floating-point operations is larger than that spent in loads and stores, and hence we consider that the loads and stores are overlapped with computation. For the loops that are flops-bound, the model uses one parameter to describe the processor's speed, denoted as γ, which is the time taken to perform one flop. For in-cache data, we assume that the processor performs at the peak speed.

- We consider a hierarchy of caches with a perfect nesting and that accessing different memory levels incurs different latencies. We use α_i to denote the level i cache access latency (in cycles), and α_{mem} to denote the main memory access latency. We only account for the capacity and compulsory misses but ignore the conflict misses. We assume that the cache memories are fully associative.

- We do not model TLB misses in detail. But in our experimental section, we count the effect of TLB misses by considering a value for α_{mem} that includes a memory access and a TLB miss.

- For interprocessor communication, 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 ignore the effect of message collisions. The value of the bandwidth is dependent on the size of message being transferred.

3 Performance Analysis of the Relevant BLAS Routines

In this section we model the performance of the numerical kernels used in Algorithm 2: the matrix multiply and update routines (steps (3.1) and (3.2)), the triangular solve routine (step (2)) and the rank one update routine (steps (1.1) and (1.2)). In SuperLU_DIST, these correspond to calls to the BLAS routines `DGEMM, DGER` and `DTRSV`.

The triangular solve and the rank one update operations (`DTRSV` and `DGER`) are memory-bound, since there is only $O(1)$ flop associated with each data item and most of the time is spent in streaming through data. For the matrix multiply and update operations (`DGEMM`), depending on the size of the matrices, the model determines if the routine is memory-bound or flops-bound. We assume that all the computing kernels are implemented using register blocking, which ensures that the number of loads/stores between registers, cache and main memory are reduced and that the multiple functional units can be effectively utilized. When using the vendor-supplied BLAS library, the number of registers used for every routine is not known. We determined the number of registers experimentally such that the hardware counter results match well with our estimations.

Analysis of DGEMM. Most of the computation in SuperLU_DIST takes place in the rank-k updates (steps (3.1) and (3.2) of Algorithm 2, which are performed using `DGEMM`. The operation is $C = aAB + bC$, where C is an $M \times N$ dense matrix, A is an $M \times K$ dense matrix, B is an $K \times N$ dense matrix and a and b are scalars. We use the superscalar implementation [9, 10] described in Algorithm 3 to analyze `DGEMM`.

Algorithm 3 Superscalar matrix multiply routine

```
for J = 1 to N step N_B do
  for L = 1 to K step K_B do
    for I = 1 to M step M_B do
      T_1 = a × A(I, L)
      C(I, J) = bC(I, J) + T_1^T B(L, J)
    end for
  end for
end for
```

The algorithm is designed for cache reuse: a block of matrix B is brought into cache and is then reused by looping over block rows of matrix A. At each step of inner loop, a block of matrix A is multiplied with a and copied into a temporary buffer T_1, which avoids bad leading dimension problem. Then a block of matrix C is updated with the result of the product of T_1 with a block of B. Loop unrolling is used to ensure good performance of the innermost in-cache matrix multiplication, which we assume is done as follows. E_a registers are used for the elements of T_1, E_b registers are used for the elements of B and E_aE_b registers are used for the elements of C. An $E_a \times E_b$ block of $C(I, J)$ is computed in the innermost loop. First, the $E_a \times E_b$ block of C is multiplied with b and stored in the registers, second, the registers are updated with elements of the product $T_1^T B(L, J)$, and third, the content of the registers are stored back to the block of C. The innermost loop for the product $T_1^T B(L, J)$ requires $(E_a + E_b)$ loads and $(2 \times E_a \times E_b)$ multiply and add operations.

For the in-cache matrix multiply operation $C(I, J) = bC(I, J) + T_1^T B(L, J)$, the number of floating-point operations for multiplying C with b is given by the second term of Equation (3). The number of loads and flops for the product $T_1^T B(L, J)$ are given in Equation (1) and the first term of Equation (3). The number of stores for copying the registers to a block of C is given in Equation (2).

$$Loads = \underbrace{\frac{M_B N_B K_B}{E_b}}_{Matrix\ T_1} + \underbrace{\frac{M_B N_B K_B}{E_a}}_{Matrix\ B} \quad (1)$$

$$Stores = \underbrace{M_B N_B}_{Matrix\ C} \quad (2)$$

$$Flops = \underbrace{2M_B N_B K_B}_{Multiply-Add} + \underbrace{M_B N_B}_{Multiply} \quad (3)$$

If the innermost loop is flops-bound and the blocks are all in cache, the performance of the in-cache block multiplication $C(I, J) = bC(I, J) + T_1^T B(L, J)$ can be estimated by using the number of Stores in Equation (2) and the number of Flops in Equation (3), which gives $(2M_B N_B K_B + M_B N_B) \times \gamma + M_B N_B \times \alpha_1$. If the innermost loop is memory-bound, the performance can be estimated as $M_B N_B \times \gamma + (M_B N_B + M_B N_B K_B/E_b + M_B N_B K_B/E_a) \times \alpha_1$.

If the innermost loop of the in-cache matrix multiplication is flops-bound, the overall performance of Algorithm 3 is given by:

$$Loads = \underbrace{MNK/N_B}_{Copy\ A\ into\ T_1} \quad (4)$$

$$Stores = \underbrace{MNK/N_B}_{Copy\ A\ into\ T_1} + \underbrace{MNK/K_B}_{Copy\ from\ registers\ to\ C} \quad (5)$$

$$Flops = \underbrace{2MNK}_{Multiply-Add} + \underbrace{MN}_{Multiply} \quad (6)$$

The choice to base our simulation on the superscalar matrix multiplication routine was motivated by the experiments presented in the IBM POWER3 Introduction and Tuning Guide [2] (page 155), which compare its performance to the performance of `DGEMM` implemented in IBM's ESSL POWER3-enhanced library. The results showed the performances are close for square matrices. The authors expect that ESSL may perform better for rectangular matrices.

The routine `DGEMM` is used at each step k of Algorithm 2, step (3) for the rank-k update of a block of the trailing matrix. To estimate the performance of step (3), we use the following notations. $Flops_{dgemm}$ denotes the number of floating-point operations for all the calls to `DGEMM` that is computed as follows. For the flops-bound calls is computed as in Equation (6). For the memory-bound calls, only the second term in Equation (6) is added to $Flops_{dgemm}$. $Loads_{dgemm}$ denotes the the number of loads computed in Equation (4). When the matrix multiplication is memory-bound, the number of loads computed with Equation (1) is added to $Loads_{dgemm}$. $Stores_{dgemm}$ denotes the number of stores of all calls to `DGEMM` as computed in Equation 5.

Analysis of DGER. Consider the rank one update $A = A - xy^T$, where A is a $M \times N$ dense matrix, x is a dense vector of dimension M and y is a dense vector of dimension N.

Since the matrix uses a column oriented storage, we assume that the outermost loop loops over columns, which minimizes the cache and TLB misses. The optimal unrolling factor of the innermost loop should be such that the size of each subcolumn is the same as the cache line size. But because of the floating-point register limitation, the unrolling factor may be restricted to a smaller number. The innermost loop then will operate on the number of columns in a vertical block. We denote the unrolling of the outermost loop as E_y and of the innermost loop as E_x. The number of loads and stores are calculated as:

$$Loads = \underbrace{MN}_{Matrix\ A} + \underbrace{\frac{MN}{E_y}}_{Vector\ x} + \underbrace{N}_{Vector\ y} \quad (7)$$

$$Stores = \underbrace{MN}_{Matrix\ A} \quad (8)$$

The routine `DGER` is used in step (1) of Algorithm 2 for factorizing block column $L(k:N,k)$. This factorization is based on rank-1 updates and scaling. We estimate the memory accesses of step (1) by summing the loads and stores corresponding to all the calls to `DGER`, which are denoted as $Loads_{dger}$ and $Stores_{dger}$.

Analysis of DTRSV. Consider the triangular solve $Lx = y$, where L is a $N \times N$ lower triangular dense matrix, and x and y are dense vectors. In our analysis we consider a column oriented (axpy) algorithm. The estimation of the number of loads and stores is similar to the one described by Vuduc et al. [14]. It assumes that register blocking is used such that the matrix is partitioned into register blocks of size $E_L \times E_L$, which leads to E_L vector loads for each register block.

$$Loads = \underbrace{\frac{N(N+1)}{2}}_{Matrix\ L} + \underbrace{\frac{N}{2}\left(\frac{N}{E_L}+1\right)}_{Vector\ x} + \underbrace{N}_{Vector\ y} = \frac{N^2}{2}\left(1+\frac{1}{E_L}\right) + \frac{3N}{2} \quad (9)$$

$$Stores = \underbrace{\frac{N}{2}\left(\frac{N}{E_L}+1\right)}_{Vector\ x} \quad (10)$$

The computation of the row block $U(k,:)$ (step (2)) of Algorithm 2 is performed through calls to `DTRSV`. An estimation of the memory accesses of this step can be therefore computed by adding the number of loads and stores associated with all the calls to `DTRSV`, which we denote as $Loads_{dtrsv}$ and $Stores_{dtrsv}$.

4 Estimation of Cache Misses

In this section we first present analytical lower and upper bounds for the number of cache misses. We then describe a method that provides better estimation of the cache misses by simulating the factorization algorithm.

Considering the Lq cache, we use l_q to denote its line size and C_q to denote its capacity, both in doubles.

At step k of Algorithm 2, the working storage consists of the space needed to store the block columns k and $k+1$ of L, the block row k of U, the blocks of the trailing matrix that will be updated, and several temporary arrays used during the updates.

We now give lower and upper bounds for the number of times each block of L and U is brought into cache. Consider block $L(i,j)$, we use x_{ij} to denote the number of updates performed on $L(i,j)$,

and y_{ij} to denote the cost of finding $L(i,j)$ when it is the destination of an update. C_j denotes the number of nonzero blocks in block row j of U, which equals the number of updates performed at step j and for which $L(i,j)$ is the source of an update. We denote $sz(L(i,j))$ as the size (in doubles) of the block $L(i,j)$ (nonzero values and indices). Our estimation takes into account the storage by columns, the leading dimension and the line size of the cache.

A lower bound M_L on the number of cache misses is obtained by assuming that there is only one compulsory miss for every block. That is, during the first update on the block $L(i,j)$, this block is brought into cache, and it will stay in cache throughout the rest of the updates when it is either source or destination, and during its own factorization.

$$M_L \approx \frac{1}{l_q} \sum_{i,j=1}^{N} (sz(L+U)(i,j) + y_{ij}) \tag{11}$$

An upper bound M_U for the number of misses is obtained by assuming that for every block, there is a compulsory/capacity miss each time when it is the destination of an update, when its factorization is computed, and when it is the source of an update.

$$M_U \approx \frac{1}{l_q} \sum_{i,j=1}^{N} ((x_{ij} + C_j + 1)\, sz(L+U)(i,j) + x_{ij} y_{ij}) \tag{12}$$

Note that these analytic bounds do not take into account the working space needed at each step of the algorithm compared to the size of the cache C_q. If the working space fits in cache, at most one compulsory/capacity miss is generated for each element of the working space. If some of the blocks updated at step $k-1$ are also updated at step k and the blocks are still in cache, then fewer misses will be incurred. If the working space does not fit in cache, it means that the cache is not large enough to hold the block row $U(k,:)$ and the block column $L(:,k)$ during the column factorization, triangular solves and the update of the trailing matrix. Then, when updating the trailing matrix, in addition to the misses incurred for every updated block, there is one additional capacity miss for the elements of $U(k,:)$ (values and indices), and for every block of $U(k,:)$, there is one capacity miss for the elements of $L(:,k)$ (values and indices). This discussion shows that it is difficult to derive realistic analytic bounds for the number of cache misses. We expect that at the beginning of the factorization, when the matrix is very sparse, the working space will fit in cache. But when the trailing matrix becomes denser, the working space will no longer fit in cache.

Hence, to obtain accurate account of cache misses, we developed a simulator that simulates the factorization algorithm and the memory system behavior. Every level of cache is associated with a policy for evicting blocks from cache when necessary, such as the least recently used (LRU) or the round robin policy. Every time a block of the matrix is accessed, the simulator checks if the block is already in cache and at which level. If it exists and the eviction policy is LRU, this block is marked as the least recently used. If the block is not in cache, it is is brought in cache, and the counter for the cache misses is incremented. When the cache capacity is exceeded, some blocks are evicted. The experimental results in Section 6 show that this simulation leads to results close to the cache misses obtained from the hardware counters.

5 Runtime Estimation

We first present an estimation of the sequential algorithm, and then describe the communication model that extends the sequential runtime estimation to an estimation of the parallel runtime.

Sequential runtime estimation. In an earlier work on performance analysis of sparse matrix-vector multiply [14], only the cost of memory accesses needs to be accounted, because that algorithm is entirely memory-bound and there is little data reuse. Sparse factorization algorithms on the other

hand contains both flops- and memory-bound kernels, and the kernel mix changes at different stages of the factorization, which complicates our analysis.

Assume that there are C cache levels. We denote by h_i the number of hits and by m_i the number of misses at the ith cache level. The hits for $i \geq 1$ are computed as $h_{i+1} = m_i - m_{i+1}$. The sequential runtime is estimated as follows:

$$T_{seq} = Flops_{dgemm}\gamma + \sum_{i=1}^{C-1} h_i\alpha_i + m_C\alpha_{mem} \tag{13}$$

The memory access cost is computed using the number of loads and stores performed for the memory-bound parts of Algorithm 2. For steps (1.1) and (1.2), the number of loads and stores is given by $Loads_{dger}$ and $Stores_{dger}$. For step (2), the number of loads and stores is given by $Loads_{dtrsv}$ and $Stores_{dtrsv}$. For steps (3.1) and (3.2), we sum up the number of loads and stores in the calls to `DGEMM` (denoted by $Loads_{dgemm}$ and $Stores_{dgemm}$), and the number of loads and stores in copying the matrix blocks into a temporary buffer to prepare for the call to `DGEMM` (denoted by $Loads_{temp}$ and $Stores_{temp}$). The L1 cache hit h_1 is computed as follows:

$$\begin{aligned} h_1 \;=\; & Loads_{dger} + Stores_{dger} + Loads_{dtrsv} + Stores_{dtrsv} + Loads_{dgemm} + Stores_{dgemm} + \\ & + Loads_{temp} + Stores_{temp} - m_1 \end{aligned}$$

As described in Section 3, $Loads_{dgemm}$ represents the number of loads performed in the memory-bound loops of `DGEMM`, plus the number of loads involved in copying the matrix blocks to temporary buffers. $Stores_{dgemm}$ represents the number of stores involved in copying into the temporary buffers or copying from the registers back to the matrix destination. The time spent in the flops-bound loops of `DGEMM` is counted in the first term of T_{seq} in Equation (13) ($Flops_{dgemm}\gamma$).

Parallel runtime estimation. In our previous work, under some simplyed assumptions, we have established the following parallel runtime estimation using a square grid of processors [7]:

$$T(N, \sqrt{P} \times \sqrt{P}) \approx \frac{F}{P}\gamma + (2N + \frac{1}{2}N\log P)\alpha + \frac{(2nnz(L) + \frac{1}{2}nnz(U)\log P)}{\sqrt{P}}\beta\,, \tag{14}$$

where, N is the order of the matrix; F is the total number of flops in the factorization; $nnz(L)$ is the number of nonzeros in the off-diagonal blocks of L; $nnz(U)$ is the number of nonzeros in the off-diagonal blocks of U. The first term represents the parallelization of the computation. The second term represents the number of broadcast messages. The third term represents the volume of communication.

For more accurate modeling of the parallel runtime, we performed a simulation of Algorithm 2. This simulation has several purposes. First, the simulation can provide a rather accurate estimation of the cache misses which cannot be determined analytically otherwise. Second, the simulation also allows us to include load imbalance factor in the model. Third, using simulation we can analyze the critical path behavior by considering load balance at each step and the amount of overlap of computation and communication.

We use the commonly adopted latency-bandwidth cost model to analyze the interprocessor communication. An improvement we made is that different values of the inverse of the communication bandwidth β are used based on the size of the messages.

The time spent in each computation step is estimated using Equation (13), where the number of loads, stores, flops and cache misses are the numbers associated with that step.

The time spent in communication is estimated as follows. In steps (b.1) and (b.2), messages are communicated using blocking sends and receives, which is implemented as follows. Consider for example that processor PE0 needs to send a message of size m to processors PE1 and PE2. PE0 first sends it to PE1 and then sends it to PE2. The time spent by all the processors PE0, PE1 and PE2

in this communication is estimated as $2(\alpha + m\beta)$. In steps (a.1), (a.2), (a.3) and (a.4), messages are communicated using nonblocking sends and receives, in which case the time is estimated as $(\alpha+m\beta)$.

The parallel runtime of Algorithm 2 is obtained by computing for each processor the runtime of the steps that lie on the critical path as follows. The time spent in the initialization of the look-ahead technique is added to the runtime of the participating processors (steps (1.1), (a.1) and (a.2)). The computation of a block row of U via triangular solves (step (2)) and the send/receive of a block row of U (steps (b.1) and (b.2)) lie on the critical path, and their times are added to the runtime of the processors involved in these steps. For the processors that own blocks of the block column $L(:, k+1)$, the update and the factorization of block column $L(:, k+1)$ lie on the critical path. Then, the larger of the time spent in the non-blocking send (step (a.3)) and the time spent in updating the trailing submatrix (step (3.2)) is counted as on the critical path. For the processors that do not own blocks of the block column $L(:, k+1)$, the larger of the time spent in receiving block column $L(:, k+1)$ (step (a.4)) and the time spent in updating the trailing matrix (step (3.2)) is added to the parallel runtime. This communication/computation overlap is determined in steps (c.1) and (c.2), when several processors belonging to the same row of the grid of processors wait for the posted sends and receives to complete. Their runtime is then determined by the processor that waits the longest.

6 Experimental Results

In this section we compare the analytic bounds and the simulation results against the actual performance of the code. We used the IBM SP RS/6000 distributed memory machine at NERSC (National Energy Research Scientific Computing Center). The system contains 2944 compute processors distributed among 184 compute nodes. Each node of 16 processors has 16 to 64 Gbytes of shared memory. Each POWER3 processor is clocked at 375 Mhz and has a peak performance of 1.5 GFlops. It has a L1 data cache of 32 KBytes and a L2 cache of 8192 KBytes. The cache line for both L1 and L2 is 128 bytes. The L1 access latency is $\alpha_1 = 1$ cycle. A data access that misses L1 but hits L2 incurs a load latency of $\alpha_2 = 9$ cycles. A data access that misses both L1 and L2 cache incurs a latency α_{mem} between 35 and 139 cycles [14]. Each processor has two floating-point units, each of which can execute one compound floating-point multiply-add (FMA) operation per cycle. Hence POWER3 can execute up to four floating-point operations per cycle. In addition, POWER3 can commit two loads/stores per cycle, and so the L_1 access latency α_1 is divided by two in our estimations.

We used several medium to large size matrices from different application domains. The characteristics of the matrices are given in Table 1, which includes the matrix order, the number of nonzeros in the input matrix A, the number of nonzeros in the factors L and U, and the number of floating-point operations in the factorization. The matrices are available from the University of Florida Sparse Matrix collection [5].

Matrix	Order	$nnz(A)$	$nnz(L+U)$ $\times 10^6$	$Flops$ $\times 10^9$
AF23560	23560	484256	11.5	5.02
BBMAT	38744	1771722	35.6	25.34
ECL32	51993	380415	41.9	60.66
FIDAPM11	22294	623554	26.7	26.81
INV-EXTR1	30412	1793881	28.9	28.73
STOMACH	213360	3021648	140.6	145.18

Table 1: Benchmark matrices.

The goal of our experiments is three-fold. Firstly, we want to compare our estimation of the sequential runtime with that of SuperLU_DIST. This would validate the accuracy of our model in counting the number of loads, stores and cache misses incurred throughout the factorization. Secondly, we want to compare the model predicted parallel runtime with the parallel runtime of SuperLU_DIST with varying number of processors. And thirdly, we use our model to analyze the performance characteristics of SuperLU_DIST.

Our model uses several parameters that have to be tuned on the target machine for which the simulation is performed, such as block size used in `DGEMM`. When there is no documentation, we determined their values as follows. The computing kernels were executed on the target architecture and the PAPI [4] results for loads, stores and cache misses were collected. Then we choose the parameters that provide a good fit for our estimation of number of loads, stores and cache misses.

In the parameter tuning phase, we have to estimate the size of the blocks MB, NB and KB used in the superscalar matrix multiplication algorithm presented in Section 3. In our experiments we have used $MB = 32$, $NB = 48$ and $KB = 48$. We also have to estimate the number of registers used for loop unrolling. For the `DGEMM` routine, we have found that a 4 by 2 unrolling matches well our estimation of number of loads. Note that this is also the unrolling level used on the IBM POWER2 [1], which ensures that the multiple functional units are fully utilized.

To estimate the number of cache misses, our simulation can consider different policies for replacing lines in cache. For L1 cache, POWER3 replaces lines using a round-robin policy within a congruence class. The congruence class is defined by the 128-way associativity of the L1 cache. For L2 cache, real addresses are directly mapped to the cache. In our simulation, we assume cache is fully associative, and we use round-robin for L1 cache and least recently used policy for L2 cache.

Figure 2 presents the number of loads and stores obtained by our performance model, and compares with the PAPI results of loads and stores obtained for SuperLU_DIST. Our model estimates well the number of loads for all the matrices, but does not do so well with the number of stores—the estimation can be smaller than PAPI numbers (up to 44% for matrix BBMAT). We also observed such underestimation when tuning our `DGEMM` model for the POWER3 architecture. We think that the overhead incurred by a call to `DGEMM` is nontrivial and is not taken into account by our model. This overhead affects the overall number of stores. But since the number of stores is generally 3 or 4 times smaller than the number of loads, the underestimation of number of stores does not have much impact on the overall estimation.

Figure 2 also presents the results of cache misses comparing our estimation, the analytic lower and upper bounds, and the PAPI results. The lower and upper bounds do not depend on the cache size, and are the same for L1 and L2 cache. For the sake of clarity, we do not include the upper bound in the plot of the L2 results. The lower bound underestimates severely the number of L1 misses (by two orders of magnitude). It also underestimates the number of misses in L2 cache, very often by one order of magnitude. For L1 cache, the upper bound is usually twice the number of PAPI misses. But as expected, it overestimates severely the number of L2 cache misses. These results show that it is difficult to identify analytic bounds for the number of misses for sparse factorizations.

Our estimation predicts well the number of cache misses, both for L1 and L2 caches. For the L1 cache, our simulation underestimates up to 17% for matrix AF23560 and overestimates up to 15% for matrix BBMAT, when compared to the PAPI L1 cache miss results. For the L2 cache, for three matrices our estimation underestimates the number of L2 cache misses (up to 33% for matrix FIDAPM11).

In the plot depicting the L2 cache misses we also include the number of TLB misses. Note that SuperLU_DIST incurs large number of TLB misses. For AF23560, the number of TLB misses is larger than the number of L2 misses. For BBMAT, the number of TLB misses is half the number of L2 misses, and for the other matrices it is less than half. This indicates that using the minimum value of the memory access latency would lead to an underestimation of the runtime.

Figure 3 presents the performance results in number of cycles. The lower bound is obtained by using the minimum value of memory access latency for α_{mem} (35 cycles) in Equation (13), which assumes a TLB hit. The upper bound is obtained by taking the maximum value for α_{mem} (139 cycles), which assumes a TLB miss. Both bounds are smaller than the measured runtime. The

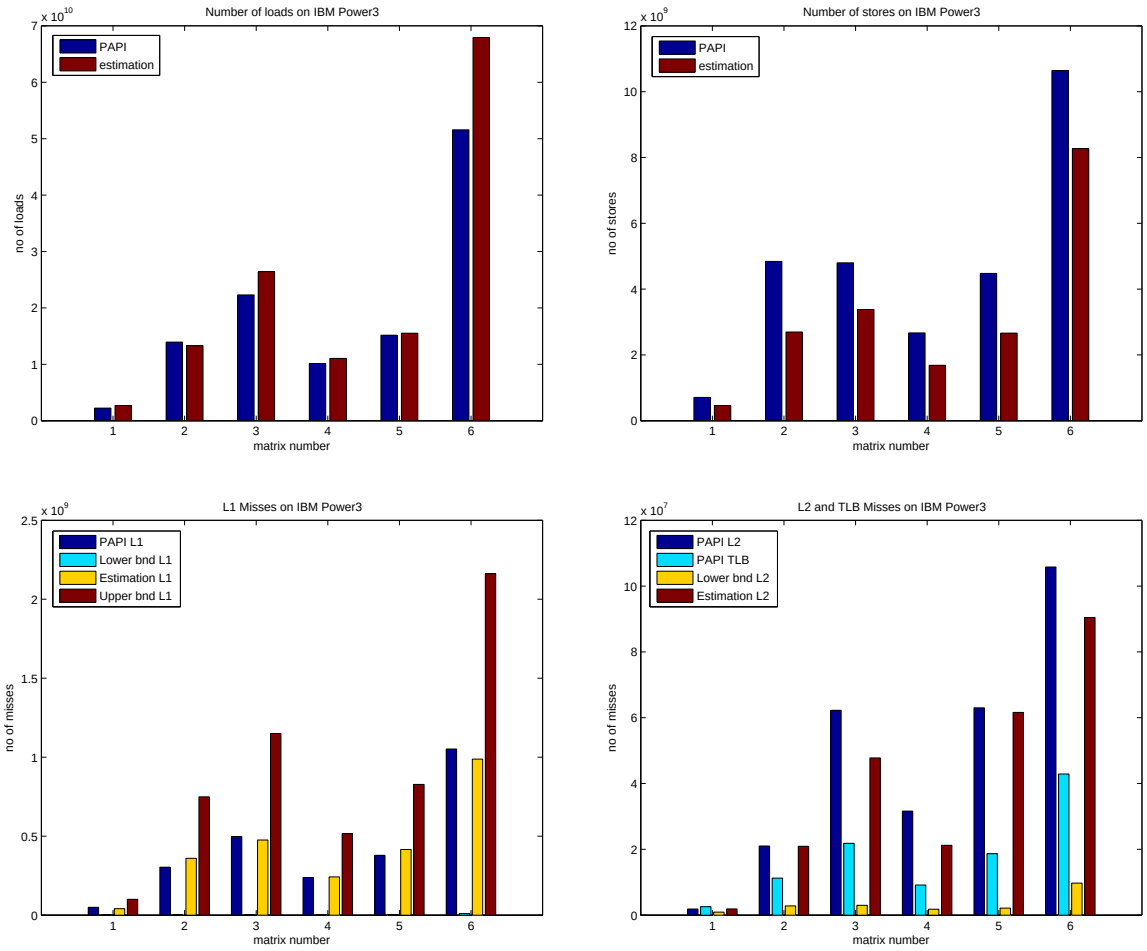

Figure 2: The two upper plots show the number of loads/stores obtained by our estimation and by PAPI measurements. The two lower plots show the number of cache misses obtained by our estimation (Estimation L1 and Estimation L2), by PAPI measurements (PAPI L1 and PAPI L2), and the analytical lower and upper bounds (Lower bnd L1, Lower bnd L2, and Upper bnd L1). The last plot also includes the number TLB misses obtained from PAPI (PAPI TLB).

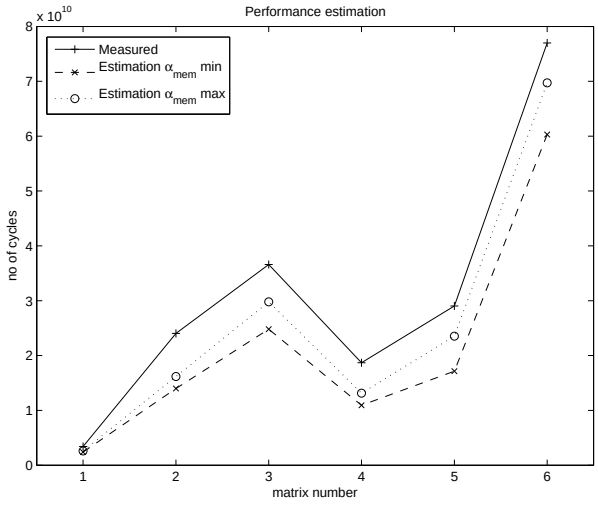

Figure 3: Uniprocessor performance in number of cycles.

upper bound can underestimate the runtime with up to 32% for BBMAT. The underestimation is larger for the lower bound (around 40% for BBMAT, FIDAPM11 and INV-EXTR1.

For parallel runtime estimation, we first need to determine the two parameters α and β used in the communication cost model. To this end, we ran the point-to-point pingpong micro-benchmark available at the NERSC web site [13] to callibrate the times needed to transfer messages of various sizes that appeared throughout the factorization. The latency value α is 8.0 microseconds. The bandwidth values used for β are tabulated in Table 2. This approach is an improvement over our previous work, where we used a single bandwidth value independent of the message size.

Message size (Bytes)	10	10^2	10^3	10^4	10^5	10^6	10^7
Bandwidth (MB/s)	1	10	81	278	274	437	430

Table 2: Bandwidth values used in the model, which depends on the message size.

Figure 4 compares the runtimes in number of cycles obtained by our simulation and by SuperLU_DIST with increasing number of processors. We also present in Table 3 the runtimes in seconds of SuperLU_DIST. Our simulation predicts well the actual performance of SuperLU_DIST.

Our performance model examines the amount of computation on the critical path. In Algorithm 2, the computation performed in steps (2), (3.1) and (1.2) lies on the critical path. That is, the computation of a block row of U, the update and the factorization of block column $k+1$ of L lie on the critical path. Since the algorithm implemented in SuperLU_DIST uses a look-ahead technique, the computation corresponding to step (3.2) (update of the trailing matrix at step k) is overlapped with the communication involved in steps (a.3) and (a.4) (send/receive of block column $k+1$ of L). Hence at each step of the algorithm, we add the computation corresponding to step (3.2) to the critical path if it is more important than the communication involved in steps (a.3) and (a.4). In Figure 4 we include the plots of the amount of computation on the critical path. The difference between the estimated runtime and the computation on the critical path corresponds to the amount of communication lying on the critical path. These plots show that on small number of processors, the computation is overlapped with the communication and the computation dominates the critical path. With increasing number of processors, the critical path is dominated by communication. This is because there are more messages of smaller size, and the time spent in communication is larger.

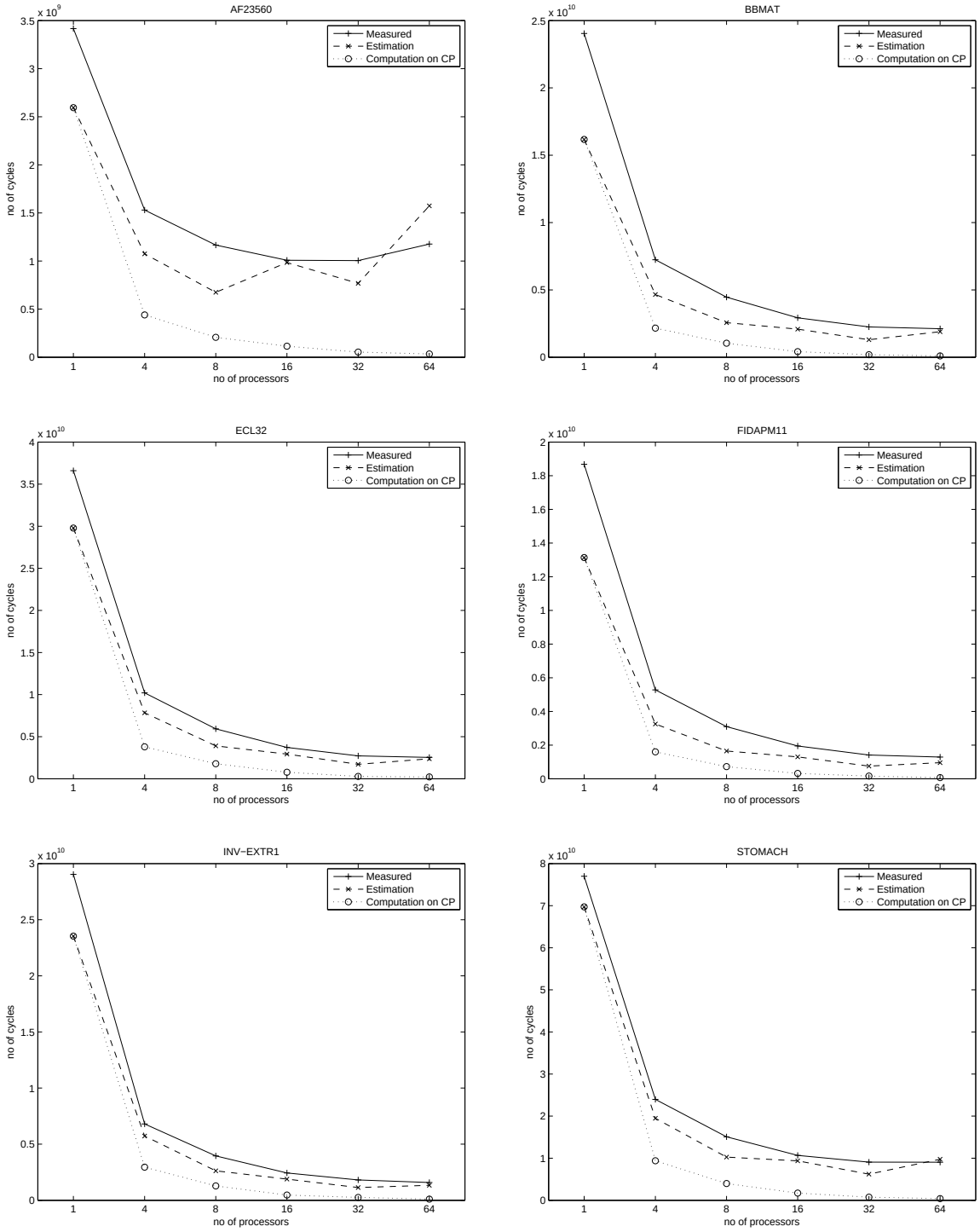

Figure 4: Runtimes in number of cycles. "Measured" represents the number of cycles obtained from running SuperLU_DIST. "Estimation" represents the number of cycles obtained by our simulator. "Computation on CP" represents the number of cycles due to computation that lies on the critical path.

		P=1	P = 4	P = 8	P = 16	P = 32	P = 64
af23560	time	9.11	4.08	3.11	2.69	2.68	3.14
	LB	1.00	1.39	1.98	2.74	4.50	7.03
bbmat	time	64.13	19.31	11.90	7.80	6.00	5.64
	LB	1.00	1.23	1.48	1.72	2.38	3.26
ecl32	time	97.6	27.24	15.81	9.93	7.24	6.76
	LB	1.00	1.12	1.32	1.52	1.92	2.39
fidapm11	time	49.83	14.08	8.26	5.19	3.77	3.44
	LB	1.00	1.17	1.40	1.65	2.22	2.92
inv-extr1	time	77.45	18.14	10.54	6.50	4.84	4.23
	LB	1.00	1.16	1.37	1.59	2.08	2.63
stomach	time	205.33	63.96	40.26	28.47	24.19	24.10
	LB	1.00	1.18	1.44	1.73	2.46	3.46

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

Note that for AF23560 our simulation overestimates the runtime. We believe this is due to the loss of accuracy associated with the communication cost model.

We also examined load balance LB throughout the parallel factorization. We define the load L to be the computation involved in the entire factorization. We then compute the load lying on the critical path L_{CP} by adding at each step the load of the most loaded processor. More precisely, consider l_{pi} being the load of processor p at step i (i.e., the computation in number of cycles performed by this processor at step i, which takes into account the number of loads/stores and cache misses as in Equation (13)). Then, $L_{CP} = \sum_{i=1}^{N} \max_{p=1}^{P} l_{pi}$. The load balance factor is computed as $LB = \frac{L_{CP}P}{L}$. In other words, LB is the load of heaviest processors lying on the critical path divided by the average load per processor. The closer is this factor to 1, the better balanced is the workload. Table 3 shows that the workload distribution is good on a small number of processors. But it can degrade quickly for some matrices, such as AF23560, leading to quick performance degradation.

7 Conclusions

We developed a performance model to analyze a parallel sparse LU factorization algorithm on modern cached-based, high-end parallel architectures. Our model characterizes the algorithmic behavior by taking account the underlying processor speed, memory system performance, as well as the interconnect speed.

Unlike Level 2 sparse BLAS operations, such as sparse matrix-vector multiply or sparse triangular solution, where performance is usually solely dominated by memory system performance, the sparse factorization algorithms usually contain a mixture of Level 2 and Level 3 BLAS routines. Depending on sparsity of the matrix, which changes at different stages of factorization, the kernels can be either memory-bound or flops-bound. We have taken into account this feature in our analysis, and our new model is more accurate in predicting performance.

Another improvement over the previous work is to use different bandwidth values to model interconnect communication speed for different message sizes. This is very important for sparse factorization algorithms, because throughout factorization, the message sizes can differ by several orders of magnitudes.

We have validated our model using the SuperLU_DIST linear system solver on an IBM POWER3 parallel machine, and showed that the model-predicted runtime is close to the measured time. Thus, this model can be realistically used to predict code performance on newly designed architectures and for new matrices.

Furthermore, our model reveals that the loss of parallel efficiency on a large number of processors is due to load imbalance and the communication cost. This points to the possible future work to improve the code. For example, we can use better matrix distribution scheme to reduce load imbalance, and use a higher level of look-ahead scheme to futher the overlap of communication with computation.

Finally, our modeling methodology can be easily adapted to study performance of other types of sparse factorizations, such as Cholesky or QR.

References

[1] R. C. Agarwal, F. G. Gustavson and M. Zubair Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Develop.*, 38(5):563–576, 1994.

[2] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide. International Business Machines, October 1998. http://www.redbooks.ibm.com.

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

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[5] T. Davis University of Florida Sparse Matrix Collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997 http://www.cise.ufl.edu/research/sparse/matrices

[6] 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.

[7] L. Grigori and X. S. Li. Performance Analysis of Parallel Right-Looking Sparse LU Factorization on Two-Dimensional Grid of Processors. Proceedings of PARA'04 Workshop on State-of-the-art in Scientific Computing, LNCS 3732, pp 768–777, 2005.

[8] 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.

[9] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[10] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: Portability and Optimization Issues . *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[11] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

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

[13] D. Skinner. IBM SP Parallel Scaling Overview.
http://www.nersc.gov/news/reports/technical/seaborg_scaling

[14] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, Katherine A. Yellick. Automatic Performance Tuning and Analysis of Sparse Triangular Solve. *ICS 2002: Workshop on Performance Optimizations via High-Level Languages and Libraries*, June 2002.

[15] R. C. Whaley, A. Petitet and J. K. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–25, 2001.