LUsim: A Framework for Simulation-Based Performance Modeling and Prediction of Parallel Sparse LU Factorization

Pietro Cicotti[*] Xiaoye S. Li[†] Scott B. Baden[*]

April 15, 2008

Abstract

Sparse parallel factorization is among the most complicated and irregular algorithms to analyze and optimize. Performance depends both on system characteristics such as the floating point rate, the memory hierarchy, and the interconnect performance, as well as input matrix characteristics such as such as the number and location of nonzeros.

We present LUsim, a simulation framework for modeling the performance of sparse LU factorization. Our framework uses micro-benchmarks to calibrate the parameters of machine characteristics and additional tools to facilitate real-time performance modeling.

We are using LUsim to analyze an existing parallel sparse LU factorization code, and to explore a latency tolerant variant. We developed and validated a model of the factorization in SuperLU_DIST, then we modeled and implemented a new variant of SuperLU_DIST, replacing a blocking collective communication phase with a non-blocking asynchronous point-to-point one. Our strategy realized a mean improvement of 11% over a suite of test matrices.

1 Introduction

Sparse linear systems of equations arise in a wide range of applications in science and engineering, and are also computationally intensive. Direct methods are useful in problems that do not have a well-defined structure, or that are highly ill-conditioned.

Parallel sparse LU factorization has been subject of many studies and analytical models have been developed [1, 8, 9, 12]. The models developed in [1, 9, 12] focused on the algorithmic variants of sparse Cholesky which are suitable for solving symmetric, positive definite systems. The theoretical model developed

[*]Department of Computer Science and Engineering University of California, San Diego La Jolla, CA 92093-0404. {pcicotti,baden}@cse.ucsd.edu.
[†]Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. xsli@lbl.gov.

in [8] was the first attempt to analyze parallel sparse LU factorization. Although these models expose performance and scalability limitations qualitatively by making some simplified assumptions, they are far from being able to predict actual performance of a code on a real machine.

Analyzing the performance of sparse matrix operations on modern computer architectures is challenging than doing so for the dense counterparts. Memory accesses rather than the floating point operations are often the bottleneck, but sometimes the combined costs of the two must be taken into consideration. Analytical performance bounds exist for sparse matrix-vector multiply and triangular solve [13, 14]. In both matrix-vector multiplication and triangular solve, the matrix needs to be read only once, hence the ratio of flops to memory accesses is $\mathcal{O}(1)$. Since there is hardly any reuse, these sparse kernels are purely memory-bound on today's processors where memory speed is growing more slowly than processor speed.[1] Therefore, an analytical model based on cache miss counts could provide a rather accurate performance upper bound [13, 14].

In contrast, the factorization operations exhibit higher reuse, and the ratio of flops to memory accesses changes during the course of elimination. During early stages of elimination, factors are sparser and workload is memory-bound. During later stages, the factors are denser; level 3 BLAS operations are appropriate and hence the workload is compute-bound. It is infeasible to derive a closed form formula that could realistically predict performance. The inaccuracy is more pronounced for matrices that come from practical applications, as opposed to model problems, and hence do not have special structures [8]. It is therefore necessary to characterize system behavior by simulating its relevant components, such as the memory hierarchy and interconnect.

The authors of [8] developed a realistic simulation model to simulate the right-looking supernodal factorization algorithm implemented in SuperLU_DIST. The major improvements include a memory system simulator, a detailed modeling of BLAS kernels, and a communication bandwidth model for varying message sizes. The simulation model has greatly improved the prediction accuracy compared with the measured runtime [7]. However, this model is based on cycle count and requires estimation of the instructions involved in the BLAS routines. In addition, the model is tailored for a specific implementation and a specific architecture; therefore, it is hard to adapt to different algorithms as well as to different machines.

We have developed a framework for simulating sparse LU factorization that supports the development of performance models for existing and new algorithms, and eases the deployment to new architectures. The framework contains

[1]This phenomenon will continue with multi-core processors.

- micro-benchmarks to calibrate the characteristics of the target system,
- a memory hierarchy simulator,
- lookup tables combined with linear interpolation for estimating the cost of BLAS kernels with varying dimensions,
- functions to evaluate the communication cost
- a supernodal dependency graph for simulating advanced scheduling algorithms.

The micro-benchmarks provide the means of obtaining the parameters required by the simulation tools. Other tools support the modeling of a factorization algorithm in terms of memory access operations, BLAS routines, and communication operations. In addition, the supernodal dependency graph facilitates the implementation of models of dependency-driven parallel factorization algorithms. The approach taken is general and applicable to any algorithm, however, LUSIM is currently integrated with SuperLU_DIST.

Our goal is to use LUSIM to aid in the design and development of a latency tolerant version of SuperLU_DIST based on dependency-driven scheduling. To this end we have used LUSIM to develop a performance model of the algorithm implemented in SuperLU_DIST and validated the model on an IBM Power5-based mainframe. We developed a second version of the performance model to simulate a latency tolerant variant of the algorithm. This variant reorders communication and computation steps in a phase of factorization, and it replaces synchronous collective communication with equivalent asynchronous point-to-point communication. We implemented the latency tolerant variant and validated it against the corresponding model. We also used the model to identify the shape of the processor grid that achieved the best performance given a fixed number of processors. Finally, we simulated an asynchronous graph-based elimination algorithm to speculate and gain insight on the possible performance improvements.

The rest of the paper is organized as follows. Sec. 2 presents the factorization algorithm of SuperLU_DIST. Sec. 3 presents LUSIM framework, its components, and the performance model that we developed for SuperLU_DIST. Sec. 4 presents the latency tolerant variant of SuperLU_DIST and the corresponding model. Sec. 5 presents experiments and results, Sec. 6 the conclusions.

2 Parallel Sparse LU Factorization

Gaussian elimination (GE) is a direct algorithm for solving systems of equations and employs two steps [6]. Given a linear system $Ax = b$, the first phase factors A into two triangular matrices L and U (lower triangular and upper triangular); the second phase determines x by solving the two triangular systems $Ly = b$ and $Ux = y$, e.g., via forward and backward substitutions.

LU factorization proceeds in steps. Each step produces a column of L and a row of U. To favor locality and achieve higher performance, the algorithm can be formulated in blocked form. Each step of the blocked variant produces a block column of L and a block row of U. Fig. 1 shows the $k-th$ step. In this step, the block column and part of the block row (L_k,L'_k and U'_k) is obtained by GE on panel C_k. GE can be implemented with a sequence of $rank-1$ updates. The block row (U_k) is obtained by solving a triangular system for each of its columns ($R_k = L'_k U_k$). Then, a $rank-k$ update is applied to the trailing sub-matrix ($A'' = A' - L_k U_k$).

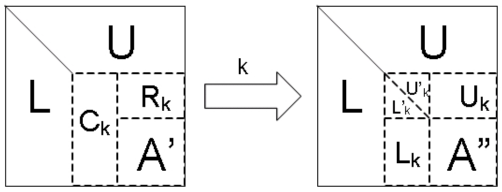


Figure 1: Step k of LU factorization: block column C_k is decomposed into L_k, L'_k, U'_k, block row R_k is updated into U_k, and A' is updated into A".

Two issues arise in the implementation of sparse parallel factorization. First, L an U are stored in a compressed format. Second, each processor will store only part of them. The implementation in SuperLU_DIST stores nonzeros in a compressed column storage format within the block boundaries; local blocks of L and U are stored in column and row major order, respectively. In addition, blocks are distributed with block cyclic layout on a 2D processors grid. (Fig. 2).

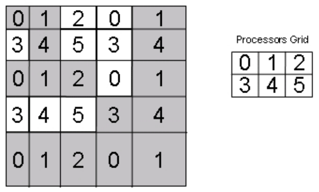


Figure 2: LU block cyclic distribution.

The implementation of SuperLU_DIST features a look-ahead technique to overlap communication with computation (the factorization algorithm is shown in Algorithm 1). Overlap is achieved dividing the update of A' in two stages: the first stage updates panel C_{k+1} (line 21), the second stage updates the remaining part of sub-matrix A' (line 30). Between the two stages, panel C_{k+1} is factored, and the new block column communication is initiated. In this way, communication can take place concurrently with the second stage of the update.

Algorithm 1 SuperLU_DIST factorization on each processor

```
if Part of block column $L(:,1)$ is local then
    $Factorization(L(local,1))$
    for all $p \in NEED(L(local,1))$ do
        $ISend(L(local,1))$
    end for
else
    $IRecv(L(remote,1))$
end if
for $k = 1$ to $N$ do
    if Part of block column $U(k,:)$ is local then
        $Update(U(k,local))$
        for all $p \in NEED(U(k,local))$ do
            $Send(U(k,local))$
        end for
    else
        $Recv(U(k,remote))$
    end if
    if $k < N$ then
        if Part of block column $L(:,k+1)$ is local then
            /* look-ahead one step */
            $A(local,k+1) \leftarrow A(local,k+1) - L(:,k) \times U(k,k+1)$
            $Factorization(L(local,k+1))$
            for all $p \in NEED(L(local,k+1))$ do
                $ISend(L(local,k+1))$
            end for
        else
            $IRecv(L(remote,k+1))$
        end if
    end if
    $A(local,k+2:N) \leftarrow A(local,k+2:N) - L(:,k) \times U(k,k+2:N)$
end for
```

3 LUsim

3.1 The Simulation Framework

We developed LUsim, a framework for simulation-based performance models that takes into account the costs of memory activity, interprocess communication, and compute intensive kernels. The framework consists of a set of micro-benchmarks, a memory hierarchy simulator, and the functionality to construct a graph (DAG) that represents data dependencies between blocks. A model is defined in a simulation module that uses the underneath tools to simulate a factorization algorithm.

3.2 Micro-benchmarks

LUsim provides a set of micro-benchmarks to probe the system's speed of memory operations, interprocess communication, and the relevant BLAS routines. The benchmarks are run off-line and the timings of the sample points are calibrated for later use in performance modeling. Time measurements are collected using provided timers. The current implementation supports *times*, *MPI_Wtime*, or *PAPI_get_real_usec*, but others can be added as needed.

3.2.1 Memory micro-benchmark

The purpose of this benchmark is to obtain an estimate of the memory access time so that it can be accounted for in the simulation of a factorization algorithm. The access time is obtained by timing memory updates. Each level in the memory hierarchy is measured in isolation to expose its access time. Given a level in the memory hierarchy, update locations are chosen systematically to hit in the selected level while missing in the lower levels. This operational regime can be achieved by carefully selecting the size of the range of addresses (N) and the access stride (s) [11]. Intuitively, for a given cache level l, if the N/s lines accessed (assuming s at least as large as a cache line) do not fit in the lower level caches but fit in the cache at l, every access will miss in the lower levels and hit at level l. Typically this condition is met by choosing N and s equal to the capacity and line size of the cache at the selected level. When N is larger than the largest cache, the operational regime is characterized by cache misses at every level and leads to an estimate of main memory access time.

The benchmark also measures the cost of a memory access in the case of a TLB miss. To collect this measure, the benchmark senses the page size and sets up an operational regime that guarantees misses at any level of the cache and in the TLB. Now the memory access time can be adjusted because the previous measurement included a TLB miss penalty every $\frac{pagesize}{s}$ accesses.

3.2.2 Network micro-benchmark

Point-to-point communication is benchmarked using a ping-pong program implemented in MPI where two MPI processes repeatedly exchange messages. The exchange is synchronous and when timed, provides a good estimate of the cost of point-to-point communication. By varying the size of the messages and timing the set of operations, the benchmark measures the transfer time for a number of representative message sizes. The times and sizes are plugged into the communication model $time = \alpha + \beta \times size$ to evaluate communication latency (α) and bandwidth (β^{-1}). First, α is obtained using messages of size 0; then α is considered constant for any larger message size and β is estimated for several representative sizes. On systems with SMP or multicore nodes, the cost of communication is much higher when communicating off-node, therefore it is important to run the benchmark for both cases. During the simulation of a factorization algorithm, the transfer time for a message can be approximated by using the same communication cost model. The cost is assessed distinguishing between intra-node and inter-node communication and selecting the value of β that is appropriate for the given message size.

LUSIM provides a broadcast benchmark as broadcast is a form of collective communication that can be used in factorization (e.g. during panel factorization). Broadcast times are heavily dependent the processor geometry, since broadcasting occurs over rows, columns, and rectangular subsets of processors. To this end LUSIM provides a broadcast benchmark. The benchmark times a sequence of broadcasts operations rooted on the same processor and repeats the measurements over the same spectrum of message sizes as with the point-to-point benchmark, and over a range of processor and node configurations. In this way, the benchmark exposes how the number of nodes and the number of processors on each node affect the cost of the broadcast. The number of nodes and processors should be taken into account to define a cost model based on α and β as before. By default, LUSIM uses the following model:

$$\log_2 P_n \times (\alpha_n + \beta_n \times s) + \log_2 P_s \times (\alpha_s + \beta_s \times s) \tag{1}$$

where P_n is the number of nodes involved, α_n and β_n are the parameters for inter-node communication, P_s is the number of processors involved on each node, α_s and β_s are the parameters for intra-node communication, and s is the size of the message. When the model does not agree with the results of the benchmark, it may be changed to reflect the results.

3.2.3 BLAS micro-benchmark

Three BLAS routines are used during factorization:

dger is used to factor block columns of L: this step is implemented using a sequence of *rank-1* updates, each corresponding to a row elementary operation in Gaussian elimination.

dtrsv is used to update the block rows of U: $A_r = L_c U_r$ and the new block row U_r is the result of a sequence of triangular solves $U_{rc} \leftarrow L_c^{-1} A_{rc}$, one for each column $c \in A_r$.

dgemm is used to update the trailing sub-matrix: $A' = A - L_c U_r$.

The micro-benchmarks time these routines with varying input dimensions. For **dger**, the dimensions of the two vectors correspond to the dimensions of the parameter space. The size of the vector that belongs to U is limited by the maximum size of a supernode, which is an adjustable parameter. In our experiments, we chose this to be ≈ 64 columns to strike a balance between parallelism, local process performance and load balance. However, there is no such limit on the size of the other vector in L. In our experiment, we calibrated the timings with sizes up to 4096; for the matrices we used, 4096 is an approximation of the maximum number of rows of a block column that a processor owns. For **dtrsv**, the first dimension of the parameter space is the size of the triangular block (which also corresponds to the size of the solution vector). The second parameter is the leading dimension of the block column to which the triangular block belongs. The leading dimension determines the stride within the columns of the triangular matrix, which affects the load/store speed of the operation, e.g. spatial locality. Again, in our calibration, we limit the leading dimension to be no more than 4096. For **dgemm**, the parameter space has three dimensions corresponding to the size of matrices to be multiplied (m,k,n). The three dimensions are limited by the maximum size of a supernode.

The parameter spaces are large and in some cases ranging from one to 4096 or more possible values in just one dimension. Timing each possible point of the parameter space is prohibitively expensive because it would be time consuming, and it would require a large amount of space to store the results. To reduce these requirements, in some areas of the space, only a subset of the points is tested. For the points that are not tested, the values are calculated using linear, bilinear or trilinear interpolations, or linear extrapolations when required during the simulation (the interpolation method depends on the number of parameters).

The micro-benchmarks are run once and the estimations are stored in files. The estimation can be loaded into tables when the simulation starts. LUsim provides support for loading and retrieving values via table lookup.

3.3 Memory Hierarchy Simulator

LUsim provides data structures and functions to simulate the memory hierarchy, populating the data structures with information obtained form the micro-benchmarks described in section 3.2.1. Hardware characteristics include cache capacity, line size, associativity, and the access times . Functions are provided to simulate access to the memory hierarchy. An access causes the simulator to update its state, and return an estimated time for the operation. Access functions evaluate differently the cost of accesses to memory during the BLAS calls. In this case, part of the memory access time is already accounted for by the benchmarks (section 3.2.1), but since the benchmarks are executed with a pre-warmed cache, the memory simulator only evaluates the cost of bringing the data block to the lowest level cache that is sufficiently large for storing it.

3.4 Data Dependency Graph

LUSIM creates a dependency graph to represent the data dependencies between blocks of matrices. The graph is built on top of the $L + U$ data structure. In the graph, a block D depends on a block S (there is an edge directed from S to D) if and only if the nonzeros in S are operands of operations that produce nonzeros in D. The dependencies can be summarized as follows:

- each block in the diagonal has edges directed to all the blocks below the diagonal and in the same block column (for the panel factorization and rank-1 updates),
- each block in the diagonal has edges directed to all the blocks above the diagonal and in the same block row (for the triangular solves), and
- each pair of blocks L_{rk} and U_{kc} have edges directed to the block A_{rc} for the rank-k update.

Fig. 3 shows the dependencies relevant to step 2 in a sample matrix. The arrows from block 1 to blocks 4 and 7 indicate that the nonzeros in 1 are required for the panel factorization. The arrows from block 1 to blocks 2 and 3 indicate that the nonzeros in 1 are required for the factorization of the block row. In the trailing sub-matrix, dependencies from the pairs of blocks $(4, 2)$, $(4, 3)$, $(7, 2)$, and $(7, 3)$ indicate the blocks required for the updates of blocks 5, 6, 8, and 9, respectively. Each vertex of the graph maintains additional information such as the number of rows and columns in the block, leading dimension, number of nonzeros, number of floating point operations needed to update the block, and the starting address of both the value and index vectors. This graph serves as computational meta data [10] which LUSIM uses to model an algorithm that schedule operations according to block dependencies. This approach is general purpose, and is not restricted to LU factorization.

3.5 The Model

Under LUSIM, we replace the factorization with a simulation module. The execution of SuperLU_DIST proceeds as in the original application, i.e., with all the preprocessing steps. When the factorization is

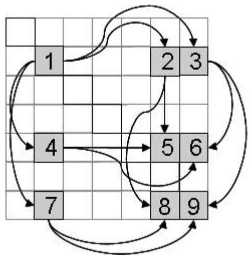


Figure 3: LUsim dependency graph example: dependencies involved in step 2

reached, the BLAS timing tables are loaded and control is transferred to the simulation module.

Algorithm 2 SuperLU_DIST: U block row update

```
1: for all $b \in MY\_UBLOCKS(k)$ do
2:    for all $j \in b$ do
3:       read row subscripts in column $j$
4:       if column $j$ is not empty then
5:          $dtrsv(L(k,k), j)$
6:       end if
7:    end for
8: end for
```

Algorithm 3 SuperLU_DIST: simulated U block row update

```
1: for all $p \in Processors \wedge UBLOCKS\_OF\_P(p,k) \neq \varnothing$ do
2:    $time[p] = time[p] + memory\_update(p, stack)$
3:    $time[p] = time[p] + memory\_read(p, index)$
4: end for
5: for all $b \in UBLOCKS(k)$ do
6:    $p \leftarrow OWNER(b)$
7:    $time[p] = time[p] + memory\_read(p, b)$
8:    for all $j \in b$ do
9:       if column $j$ is not empty then
10:          $time[p] = time[p] + lookup(dtrsv, sizeof(j))$
11:       end if
12:    end for
13: end for
14: for all $p \in Processors \wedge UBLOCKS\_OF\_P(p,k) \neq \varnothing$ do
15:    $time[p] = time[p] + memory\_update(p, stack)$
16: end for
```

The simulation module is a software module that combines several components of the framework to simulate an algorithm running on a specific architecture. To set up the environment, the simulation module simply instantiates the memory system. Then, the factorization algorithm is simulated in terms of BLAS routines, memory operations, and communication primitives.

We developed three models: one for simulating the algorithm currently implemented in SuperLU_DIST (Algorithm 1), one for a latency tolerant variant (discussed in Sec. 4), and one for a dependency-driven algorithm (discussed in Sec. 5.4). The simulations proceed by increasing a timer for each processor in the computation. Time is added for each BLAS routine, according to the value retrieved from the tables; for each memory access, according to the value returned when updating the state of the memory simulator; and for each communication according to the transfer time returned by the evaluation functions. In addition, when communication involves synchronization the timers are synchronized accordingly.

For illustrative purpose, Algorithm 2 shows the procedure that implements the update of a block row of U (Algorithm 1, line 11) and Algorithm 3 shows the corresponding procedure in the simulation module. Algorithm 3 shows how the cost of each simulated operation is collected. Memory access functions (e.g., *memory_update*, *memory_read*) take a parameter p to indicate the instance of the simulated memory system that belongs to processor p.

4 New Latency-reducing Factorization

The factorization in SuperLU_DIST implements a right-looking algorithm with a look-ahead technique [6]. The look-ahead enables overlapping the communication of an updated block column with the computation of the trailing sub-matrix update. However, the other communication phases are not overlapped with computation. In particular, communication during panel factorization is implemented with a series of broadcasts. The broadcasts, which are rooted at the processor that owns the diagonal block (we refer to this processor as a *diagonal processor*), carry a row of the upper triangular block in the diagonal to the processors owning part of the panel. In this way, these processors can factor the block column concurrently.

A sequence of broadcasts is expensive. The size of the broadcasted message is limited to the size of a row of the block column and decreases until it is just one number. Therefore, this communication phase does not take advantage of the available bandwidth. In addition, not only the delays due to latency increase with the number of broadcasts (which is equal to the number of columns in the panel), in many implementations the broadcast is a synchronous operation and keeps the sender waiting if the receivers are not ready.

Since the upper triangular block of the diagonal is not modified after the panel factorization, in an alternative implementation, the diagonal processor may perform all the local computation and then issue a series of non-blocking sends. There are three advantages: only one large message is sent taking advantage of the available bandwidth, each receiver experience just a latency delay, and there is no need to enforce synchronization at the sender.

The time of the panel factorization can be expressed as a function of number of processors owning part of the block column (P), columns in the block (n), latency (α), bandwidth (β^{-1}), and time spent in the local computation (c). For simplicity and for the purposes of discussion, we assume that the processors spend c time in the local computation to factor the panel concurrently. This assumption implies that the panel is equally divided between the processors. Under these assumptions, the time to complete a panel factorization in the current implementation can be represented as:

$$\sum_{r=1}^{n} \log_2 P(\alpha + \beta r) + c = \log_2 P(\alpha n + \beta \frac{n \times (n+1)}{2}) + c \tag{2}$$

On the other hand, the time to complete a panel factorization in the proposed implementation can be represented as:

$$P(\alpha + \beta \frac{n \times (n+1)}{2}) + c + c \tag{3}$$

At first, the cost expressed in Eq. (3) looks higher than that of Eq. (2). The local computation cost c contributes to the overall cost two times: first because no message is sent before the sender finishes computing, then because all the other processors compute their local factorization. In addition, the prefactor of the parenthesized expression is now P instead of $\log_2 P$. However, the contribution of α does not depend on n. More importantly, the diagonal processor completes the local computation in time c, then proceeds immediately and starts the look-ahead send earlier. This send is relevant because it provides the data for the linear solves for the block row update. In contrast, even if the other processors are delayed (c time), their dependents for the next sub-matrix update will also likely be waiting for the solves of the corresponding block-row; therefore, this wait might have less of an effect than it appears in Eq. (3).[2] Finally, though the communication cost strongly depends on the input matrix and the processor grid, the total number of messages sent is considerably reduced in the second case.

Other alternative methods can be considered in the future. For example, one broadcast can be used instead of multiple asynchronous point-to-point messages. This reduces the prefactor from P to $\log_2 P$ in Eq. (3) but introduces synchronization. Another possibility is for the diagonal processor to factor only the diagonal block and immediately initiate the send message before finishing the local panel factorization. This reduces the wait time of the other processors to a fraction of c. Although we are currently considering all these alternatives, in this paper we focus only on the algorithm described above. To better evaluate its performance, we developed a model which differs from the model described in Sec. 3.5, and simulates the Latency-reducing factorization algorithm. In addition, the model uses the estimate of the local computations based on the actual panel distribution with no assumption of even distribution.

[2]It is difficult to quantify the amount of overlap between communication and computation.

5 Experimental Results

5.1 Testbed

We run our experiments on Bassi, a scalable system located at the National Energy Research Scientific Computing Center (NERSC). Bassi has 111 compute nodes. Each node hosts eight Power5 dual-chip modules running at 1.9 GHz, with a single active core. The active core has exclusive access to 32KB of L1 cache, 1.875MB of L2 cache, 36MB of L3 cache. Processors on a node share 32 GB of memory and a two-link network adapter card. Nodes are connected through an High Performance Switch (*Federation switch*). Relevant characteristics of the Power5 are summarized in Table 5.1.

Table 1: Power5: Bassi Configuration

Clock Speed	1.9 GHz
L1 Data Cache Size	32KB
L1 Data Cache Associativity	4
L1 Data Cache Line Size	128
L2 Cache Size	1.875MB
L2 Cache Associativity	10
L2 Cache Line Size	128
L3 Cache Size	36MB
L3 Cache Associativity	12
L3 Cache Line Size	256
TLB Entries	1024
TLB Associativity	4

We ran the memory and network micro-benchmarks on Bassi. Table 2 shows results collected running the memory micro-benchmark on one processor. The timings are for cache access time, memory access time, memory access time adjusted taking TLB misses into account, and memory access time with TLB miss penalty. When all the processors on a node are utilized, we did not observe any difference in the timings. This is not surprising because in the benchmark, processes do not share memory and and run with task and memory affinity. The measured latencies were used to set up the memory simulator in our performance models.

Table 2: Memory Micro-Benchmark Results

	L1	L2	L3	Memory	Memory (A)	Memory (T)
Time (ns)	0.6	5.2	12.8	26.1	22.1	85.7

We ran the network micro-benchmark four times: on one node with 2 processes and with 8 processes to

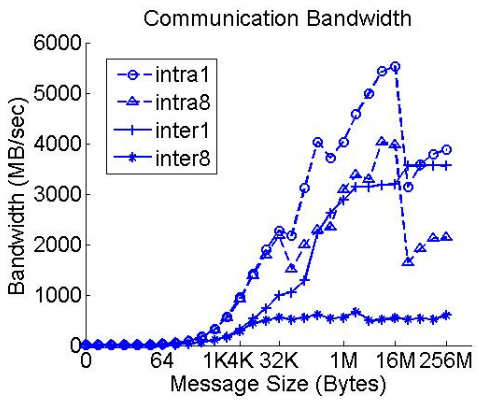


Figure 4: Interconnect bandwidth measured on Bassi.

Table 3: Network Micro-Benchmark Bandwidth Results

Size (Bytes)	64	128	256	512	1K	4K	32K	1M	16M	256M
Bandwidth Intra1 (MB/sec)	30	46	90	171	318	945	2469	4024	5536	3873
Bandwidth Intra8 (MB/sec)	30	44	87	165	309	920	2167	3089	3971	2133
Bandwidth Inter1 (MB/sec)	11	17	34	62	111	325	995	2878	2187	3562
Bandwidth Inter8 (MB/sec)	11	16	32	59	104	281	548	552	543	597

measure the cost of intra-node communication (Intra1 and Intra8), then on two nodes with 2 processes and 16 processes total (Inter1 and Inter8) to measure the cost of inter-node communication. In all the settings, pairs of processes exchange messages independently from other pairs. Results are shown in Fig. 4 and a representative subset of the data is presented in Table 3. It can be observed how, when all the processors are utilized, communication cost rises and the available bandwidth drops. The measured peak bandwidth for intra-node communication dropped from $5536MB/sec$ to $3971MB/sec$ for messages with size $16MB$. The measured peak bandwidth for inter-node communication dropped from $3562MB/sec$ to $597MB/sec$ for messages with size $256MB$. The benchmark approximated the point-to-point latency to $2\mu sec$ and $5\mu sec$ for intra-node and inter-node communication respectively. The measured latency and bandwidth values were used for communication cost evaluation in our performance models.

We also evaluated the accuracy of the default cost model (1). We ran the broadcast micro-benchmark for sizes ranging from 8 to 512 bytes. The range is large enough to cover all possible cases since at most 64 doubles is sent for each row. The runs were also repeated on several node and processor geometries. Compared to our measurements, the average relative error of the default model is 27%.

One machine-specific modification of the default model is to consider the two-link network adapter card, which leads to the following refined broadcast cost model:

$$max(\lceil \log_2(P_n/2) \rceil, 1) \times (\alpha_n + \beta_n \times s) + \log_2 P_s \times (\alpha_s + \beta_s \times s) \tag{4}$$

This model is based on the idea that the cost of sending messages can be reduced effectively by using the two links simultaneously. Compared to our measurements, the average relative error of the machine specific model is reduced to 9.5%.

Though relatively accurate, the model requires further improvement in some cases. For example, in the case of local node broadcast on 8 processors it was observed that the average relative error is 2.6%, for sizes from 8 to 64 bytes, and 23.7% for sizes from 128 to 512 bytes. Though not so pronounced, a similar trend is observed in other processors geometries.

Table 4: Benchmark matrices characterization: order(N), nonzeros(nnz), sparsity($nnz(A)/N$), and structural symmetry (Sym).

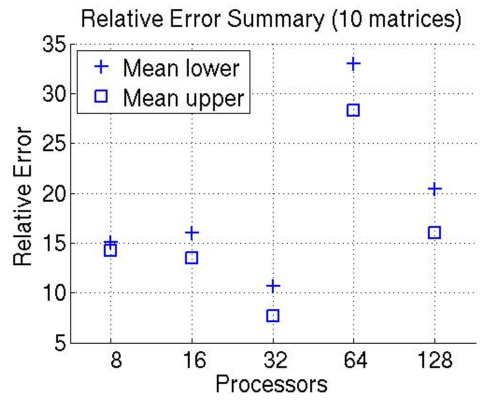
Name	N	$nnz(A)$	$nnz(L+U)$	$nnz(A)/N$	Sym
(BB) bbmat	38744	1771722	36074161	45.7	53%
(DD) dds15	834575	13100653	875305401	15.7	NA
(EC) ecl32	51993	380415	41938340	7.3	92%
(GJ) g7jac200	59310	717620	37369148	12.1	3%
(IE) inv-extrusion	30410	1793881	30245222	59	97%
(M1) matrix181	589698	95179212	898865100	161.4	NA
(MT) mixing-tank	29957	1990919	44562362	66.5	99%
(ST) stomach	213360	3021648	140580464	14.2	85%
(TO) torso1	116158	8516500	27742019	70.5	42%
(TT) twotone	120750	1206265	11360029	10.0	24%

Many of our test matrices come from the University of Florida Sparse Matrix Collection [5]. Matrix dds15 came from accelerator structural design [4], and matrix181 came from the fusion energy study [2]. All the matrices are instances of real engineering, scientific, and economic problems. Table 4 summarizes the characteristics of the matrices.

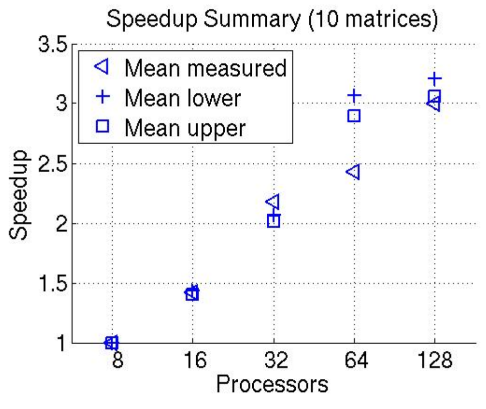
5.2 Validation of the Performance Model

To evaluate the accuracy of our model we factored all the matrices using 8, 16, 32, 64, 128 processors. Then we used our performance model to produce two estimates: a *lower* estimate, where we used the best memory access time (no TLB miss penalty) and the best communication time (only 1 processor per node used), and an *upper* estimate, where we used the worst memory access time (always TLB miss penalty) and the worst communication time (8 processors per node used).

In all the experiments, the model underestimated the real factorization time, even with the *upper* estimate. In all cases, the upper estimates are more accurate than the lower estimates. Fig. 5a shows the mean relative error for each processors configuration. Here and thereafter, the "mean" is taken over the ten test matrices. Noticeably, on 64 processors, the error is much higher than in the other cases. A more detailed analysis of the case showed that for many matrices, that configuration is very inefficient and doesn't provide much speedup compared to the 32 processors configuration. In fact, a profile of execution showed that the communication cost almost doubles in that case. However, the performance model does not seem to capture such inefficiency.



(a)



(b)

Figure 5: Relative Error and Speedup

Despite the relatively large error, the model is capable of predicting the speedup accurately. Fig. 5b shows the speedup achieved relatively to the 8 processors configuration. It can be observed that the mean speedup estimated, especially for the *upper* estimate, is very close to the mean speedup achieved. As expected, the speedup on 64 processors is not as accurate, but on 128 processors, the estimated speedup agrees again with the measurement. In the remaining of the paper, we present only results from the *upper* estimates as it proved to be more accurate than the *lower* estimates.

5.3 Evaluation of the Latency-reducing Factorization

The algorithm proposed in Sec. 4 aims at reducing the factorization time by overlapping communication and computation and by reducing the synchronization of the operations in the panel factorization. The model developed for the latency-reducing algorithm suggests that its performance compares favorably with the performance of the original algorithm in all cases but on 8 processors. However, the implementation compares favorably to the original version in all cases as shown in Fig. 6. The average speedup achieved in this case is 4%, which could explain why is not correctly predicted; however, the negative speedup detected suggests the need for revising the communication cost model. The performance model is imprecise also on the 128 processors case where it overestimates the speedup achieved which again, suggests the need for a deeper analysis of the communication cost model.

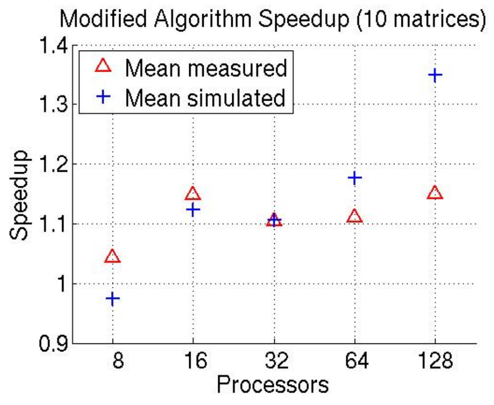


Figure 6: Speedup of factorization when using latency-reducing algorithm.

The Latency-reducing factorization algorithm performs better than the original one because of its ability to overlap communication with computation and to better tolerate the latencies. In almost all the cases tested, the new algorithm outperformed the original one, achieving an 11% average speedup. Only in four cases performance degraded by 1% to 4%. Also, a 46% peak speedup was observed for matrix *torso1* on 16 processors. Encouraged by the results, we further investigated a latency-tolerant algorithm for the entire factorization, which will be discussed in Sec. 5.4.

5.4 Speculative Dependency-Driven Execution

We developed LUSIM to model the performance of parallel sparse factorization and help predicting the performance of new factorization algorithms. In fact, the main purpose is to help in the design of a latency-tolerant algorithm where scheduling is completely dependency-driven. Here we propose a performance model for a first rough approximation of a dependency-driven algorithm. The model does not take into account the overhead of scheduling or any cost other than communication, memory access to retrieve data, and BLAS routines. We believe these overheads to be low, and will comment on them in future work.

The model is based on the model of the original implementation with two notable differences: communication is always asynchronous, and computations are scheduled according to the availability of the data the operands. While the graph is constructed with supernode granularity, the algorithm here modeled works at panel granularity. This means that communication is still based on whole block rows and block columns with the exception of the diagonal block. In that case, however, the broadcasts are replaced by point-to-point sends as in the modified algorithm described in Sec. 4. Computation is also set at panel granularity; for example, a block column factorization takes place as a set of local *rank-1* updates that span the whole block column.

The graph is used to check for availability of the operands of the subsequent operation. As a side benefit, the edges of the graph enables direct retrieval of blocks of data. Since the blocks of a block column are stored out of order, using the graph saves some of the time spent traversing the index vector to find the desired block of data.

We ran the graph-based model on all 10 matrices. Table 5 shows in the second column the average speedup achieved by the model of the new panel algorithm (also shown in Fig. 6), and in the third column the average speedup achieved the graph-based model. The model of the dependency-driven algorithm is encouraging as it further improves the speedup achieved by the new panel algorithm. In particular, on 128 processors it achieves 70% speedup on average, where the 35% speedup predicted by the modified model.

Table 5: Dependency-driven model: speedup.

Processors	Latency-reducing Algorithm	Dependency-Driven Algorithm
8	0.97	1.01
16	1.12	1.21
32	1.11	1.22
64	1.18	1.34
128	1.35	1.70

6 Conclusions

This paper described LUsim, a framework for performance modeling of parallel sparse LU factorization. With LUsim we developed a model for the factorization algorithm implemented in SuperLU_DIST. We used the model to predict the factorization time for 10 test matrices on 8, 16, 32, 64, and 128 processors. Despite a relatively large mean error on the 64 processor configuration, the model accurately predicted the speedups for all five processors configurations. We used the model to predict the performance of an improved implementation with a new latency-reducing factorization. In this case, the model accurately predicted the performance improvement over the original algorithm. The implementation of the latency-reducing factorization achieved improvements comparable to the prediction in most cases, and exceeded the prediction otherwise. Over 10 matrices and 5 processors configurations ranging from 8 to 128 processors, the new implementation realized a 11% average speedup and a 46% speedup in the best case. Finally, we developed a model to speculate on the performance gains from a dependency-driven factorization algorithm.

We will extend the work in the following directions:

- We plan to complete a detailed performance model of the factorization algorithm in SuperLU_DIST in order to expose how each phase in the computation contributes to the overall cost, and to which extent architectural details affect this contributions.
- We plan to investigate and develop latency-tolerant factorization along two directions. We demonstrated how to increase the overlap of computation with communication and will continue improving SuperLU_DIST as suggested in Sec. 4. We speculated on the potential of a dependency-driven factorization algorithm and we will implement such a dependency-driven algorithm. To this end we are considering using *Tarragon*[3], a run-time system for dependency-driven execution.

Acknowledgments

This research was supported in part by NSF contract ACI0326013, and in part by the Director, Office of AdvancedScientific Computing Research of the U.S. Department of Energy undercontract number DE-AC03-76SF00098. It used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Cleve 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.

[2] Center for Extended MHD Modeling (CEMM). URL: http://w3.pppl.gov/cemm/.

[3] Pietro Cicotti and Scott B. Baden. Poster reception—asynchronous programming with tarragon. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 159, New York, NY, USA, 2006. ACM Press.

[4] Community Petascale Project for Accelerator Science and Simulation (COMPASS). URL: https://compass.fnal.gov/.

[5] Davis Tim. The University of Florida Sparse Matrix Collection. In *NA Digest*, volume 29. June 1997. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[6] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, 1996.

[7] L. Grigori and X. S. Li. Towards an accurate performance modeling of parallel sparse factorization. *Applicable Algebra in Engineering, Communication, and Computing*, 18(3):241–261, 2007.

[8] Laura Grigori and Xiaoye S. Li. Performance analysis of parallel right-looking sparse LU factorization on two dimensional grid of processors. In *Proceedings of PARA'04*, LNCS 3732, Springer, pages 768–777, 2006.

[9] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel and Distributed Systems*, 8:502–520, 1997.

[10] P.H.J. Kelly, O. Beckmann, A. Field, and S. Baden. Themis: Component dependence metadata in adaptive parallel applications. *Parallel Processing Letters*, 11(4):455–470, December 2001.

[11] Rafael H. Saavedra and Alan J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times. Technical Report USC-CS-93-546, University of Southern California, 1993.

[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 191–209. Springer-Verlag, New York, 1993.

[13] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, and R. Nishtala. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the SC2002*, Baltimore, November 2002.

[14] R. Vuduc, S. Kamil, J Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *Proceedings of the ICS 2002: Workshop on Performation Optimizations via High-Level Languages and Libraies*, New York, June 2002.