Performance Modeling Tools for Parallel Sparse Linear Algebra Computations

Pietro Cicotti [a] Xiaoye S. Li [b] and Scott B. Baden [a]

[a] *Department of Computer Science and Engineering University of California, San Diego La Jolla, CA 92093-0404. {pcicotti,baden}@cse.ucsd.edu.*

[b] *Corresponding Author: Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. xsli@lbl.gov.*

Abstract. We developed a Performance Modeling Tools (PMTOOLS) library to enable simulation-based performance modeling for parallel sparse linear algebra algorithms. The library includes micro-benchmarks for calibrating the system's parameters, functions for collecting and retrieving performance data, and a cache simulator for modeling the detailed memory system activities. Using these tools, we have built simulation modules to model and predict performance of different variants of parallel sparse LU and Cholesky factorization algorithms. We validated the simulated results with the existing implementation in SuperLU_DIST, and showed that our performance prediction errors are only 6.1% and 6.6% with 64 processors IBM power5 and Cray XT4, respectively. More importantly, we have successfully used this simulation framework to forecast the performance of different algorithm choices, and helped prototyping new algorithm implementations.

Keywords. Performance modeling, linear algebra, parallel sparse factorizations

1. Introduction

Developing accurate performance models for parallel sparse linear algebra algorithms becomes increasingly important because the design space of parallelization strategies is large and implementing each strategy is labor-intensive and requires significant amount of expertise. Accurate performance modeling and prediction of different algorithms could narrow down the design choices and point to the most promising algorithms to implement. It is intractable to derive closed form analytical models for most sparse matrix algorithms because performance of such algorithms depends on both the underlying computer system and the input characteristics. In many parallel sparse matrix algorithms, computations often alternate between "memory-bound" phases and "CPU-bound" phases [3]. In addition, the sparsity and the nonzeros locations determine the amount and the granularity of communication as well as load distribution. This also presents a significant challenge for accurate performance modeling. On the other hand, *simulation-based performance models* offer the potential for taking into account details of the input and reproduce the relevant system behaviors. We propose a methodology for creating performance models where the computation is represented by a sequence of interleaved memory operations, the calls to the BLAS routines, and the inter-process communications. Our model simulates the steps of the algorithm as driven by the input

and charges for the cost of memory accesses, local arithmetic calculations, and communications.

Our simulation framework consists of two components. The first component is a low level library of Performance Modeling Tools, PMTOOLS, which is based on our previous work in modeling parallel sparse LU factorization [1]. The PMTOOLS library can be generally useful for modeling any parallel linear algebra algorithms, dense or sparse. With slight modification, PMTOOLS can also be used to study performance of larger application codes. The second component is application-specific simulation module, which depends on the specific algorithm to be modeled and which consults PMTOOLS to obtain the running times of the low level operations. In the following sections, we will describe each component in more detail, and show the simulation results with validations.

Our objective is twofold: 1) we would like to predict performance of existing implementations on different architectures, including hypothetical machines that do not exist yet. To this end, we build the application-specific simulation module by mimicking computations and communications in the actual implementation including software/hardware interactions. 2) We would like to use this simulation framework to help design and prototype new algorithms, and eliminate bad algorithm choices. This is an even more valuable and novel use of this predictive framework, which distinguishes our work from the others in the area of performance evaluation and benchmarking.

2. Performance Modeling Tools

PMTOOLS is a collection of tools intended to calibrate performance of the machine's individual components. It contains the micro-benchmarks, the data structures and management routines for storing and retrieving the data collected by the micro-benchmarks, and a cache simulator to represent the memory hierarchy at runtime. Each micro-benchmark is run off-line and measures the times taken by a basic operation, such as the execution of a BLAS routine, over a parameter space of interest and the data are collected into the tables once and for all. These data represent the cost functions of the relevant operations at the simulation time. Since the configuration space can be extremely large, some configurations are omitted. The omitted values are later estimated using various interpolation or curve-fitting algorithms [5,12]. Currently, PMTOOLS contains the following three models.

1) Memory model. PMTOOLS provides a cache simulator capable of combining several cache instances. Each instance is parameterized according to the cache characteristics (e.g. capacity and associativity) of the target system. Thus, we can compose an entire memory hierarchy simulator that also includes the TLBs. The simulator has two functions: to maintain the state of the memory hierarchy throughout a simulation and to estimate the cost of each operation. We designed a memory micro-benchmark to measure the latency and bandwidth of each level of the cache. The latencies are measured by timing the updates at various memory locations. Each level in the memory hierarchy is measured in isolation by choosing the locations of the updates in a controlled operational regime, so that each access hits the desired cache level. The latency for this level is derived accurately [9]. The bandwidth is measured by timing sequential memory accesses. The measured bandwidth is applicable as long as the number of consecutive memory locations transferred is greater than a threshold that triggers hardware prefetching.

2) Model of linear algebra kernels. Most high level linear algebra algorithms can be expressed as a number of calls to the BLAS routines, which provide a nice abstract interface between the high level algorithms and the low level arithmetic engine. In PMTOOLS, we use micro-benchmarks to measure performance of the BLAS kernels of interest, with varying matrix/vector dimensions. The timings are measured off-line and stored in the lookup tables, and will be retrieved later during the application's simulation. Since the parameters space can be very large, timing each possible point is both time-consuming and requires large tables. Instead, we benchmark the routines of all the small dimensions which are sensitive to timings, but only for a subset of the other larger dimensions. During simulation, the probing function first searches the tables for the given dimensions; if such dimension is not found, an estimate is obtained by linear interpolation using the closest times available in the table (or extrapolation if the dimensions are outside the bounds). We note that this simple interpolation/extrapolation can be improved by a more sophisticated scheme to improve prediction accuracy [12].

3) Communication model. The communication model is based on point-to-point message passing. The other communication primitives such as collective communication are modeled as a sequence of point-to-point transfers. We perform the measurement of varying message sizes off-line, store the timings in a lookup table, and retrieve the timings during the actual simulation. The cost of a transfer is measured using a ping-pong micro-benchmark, and again, the probing function either consults the lookup table for the cost of a given message size, or does interpolation for the message sizes not directly measured. Our table includes the ping-pong timings between two processors, as well as the timings with simultaneous ping-pongs among all pairs of processors. We also support the SMP or multicore CMP nodes in that the micro-benchmark is run to measure both intra-node and inter-node communication. The probing function can take the information about the number of nodes and the two processes involved, and return the appropriate measured cost.

3. Application-specific Simulation Modules

To demonstrate the effectiveness of PMTOOLS for achieving our first objective, we consider an existing implementation of sparse LU factorization in SuperLU_DIST [6]. The LU factorization algorithm uses supernodal block structure in the kernel. The factored matrices L and U are stored in compressed format and distributed with block cyclic layout on a 2D processor grid. Each unit of work (or a vertex in the dataflow graph) is defined as a block column/row factorization in a right-looking fashion, and the factorization proceeds sequentially along the outer K-dimension. A single level of pipelining (i.e. one step look-ahead) was implemented by hand across the kth and $(k+1)$st iterations to shorten the critical path. Figure 1 illustrates such a block distribution. The highlighted block column and row correspond to the kth step of outer factorization.

We developed a simulation module which is a close mimic of the factorization algorithm except that we do not perform the actual operations, but only advance the simulated runtime. This is done by charging the times for each BLAS function and MPI function, using the probing function provided by PMTOOLS. The memory system is also closely simulated as if the algorithm is run on a real machine. The simulation module first instantiates the memory state for the cache simulator in PMTOOLS. During simulation, whenever the algorithm involves a memory access, the module consults the cache simu-

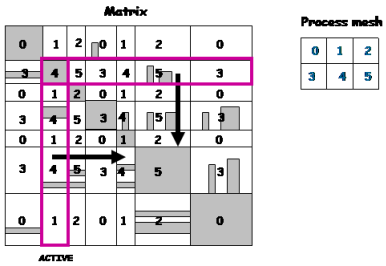


Figure 1. Sparse LU block cyclic distribution.

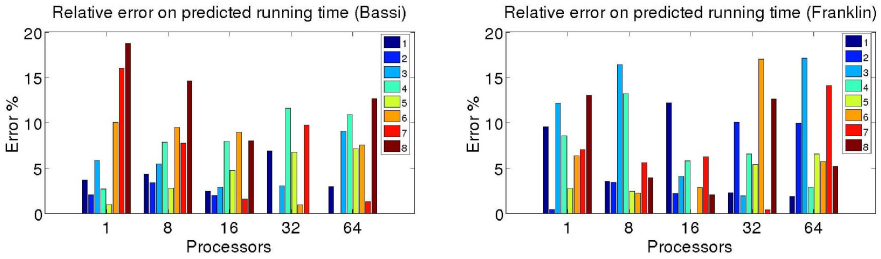
Algorithm 1 SuperLU_DIST: simulated update of the kth block row of U.

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**

lator to obtain the access time and to trigger an update to the state of the memory. As an illustration, Algorithm 1 shows the simulated procedure that corresponds to the update of a block row of U. The procedure shows how the cost of each simulated operation is collected. Memory access functions (e.g., $memory\_update$ and $memory\_read$) take a parameter p to indicate the instance of the simulated memory system that belongs to processor p.

We validated the model on eight unsymmetric matrices selected from the University of Florida Sparse Matrix Collection [2]: **bbmat, ecl32, g7jac200, inv-extrusion, mixing-tank, stomach, torso1, and twotone**. We used up to 64 processors with two different machines at NERSC, one is an IBM Power5 (`bassi`) and another is a Cray XT4 with quad-core AMD Opteron nodes (`franklin`). Figure 2 shows the percentage absolute errors of the predicted times versus the actual times for the eight matrices (shown in the legend). In most cases, our simulated time is accurate within 15% error. The average absolute prediction errors among all matrices and processor configurations are only 6.1% and 6.6% for the IBM Power5 and the Cray XT4, respectively. This level of accuracy is remarkable for such complicated sparse matrix calculations.

In addition to analyzing performance of the existing implementation, we have used this simulation model to forecast the performance of different algorithm choices, and successfully improved the parallel efficiency.



(a) IBM Power5 (b) Cray XT4

Figure 2. Accuracy of the simulated factorization times compared to the actual times. Each bar represents one matrix.

1) Shape of the processor grid. For a fixed processor count, the shape of the 2D processor grid can affect the factorization speed. We used our model to predict the optimal shape of the processor grid with 16 processors. Figure 3 shows the actual running times and the simulated times for the eight matrices. The model was able to correctly sort the grid shapes using the simulated execution times, and the processor grid shape 2×8 is the best in most cases.

2) Latency-reducing panel factorization. The panel factorization at each step of the outer loop usually lies on the critical path. The original algorithm performs a series of broadcasts for each rank-1 update. An alternative design is to use asynchronous point-to-point communication, reducing the latency cost. We ran the simulation with different strategies, and then chose the best simulated one to implement. This optimization led to 20-25% improvement for the entire factorization on 64 processors.

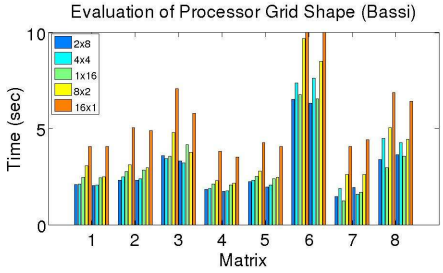


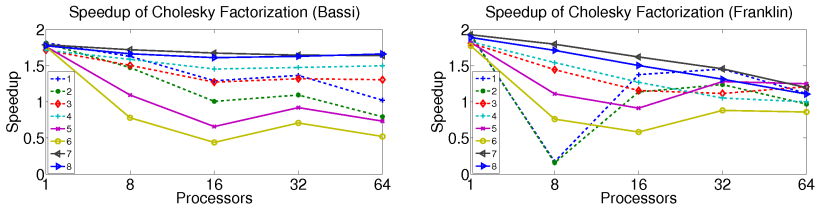
Figure 3. Choices of the shapes of the processor grid with 16 processors. Each matrix has 10 bars, divided into 2 groups of 5 each. The left group is from the actual runs, and the right group is from simulation. For each matrix, the two bars of the same color represent the same shape.

4. Prototyping New Factorization Algorithms by Simulation

We have been designing and prototyping parallel sparse Cholesky factorization using the SuperLU_DIST code base. Cholesky factorization works for symmetric positive definite matrices, which requires half of the operations and half of the memory compared to LU factorization. That is, after factorization, the upper triangular matrix is equal to the transpose of the lower triangular matrix, and so the algorithm only needs to compute the lower triangular matrix.

In a symmetric factorization, the computation of the block rows and the update of the upper triangular part of the trailing submatrices are not required. However, the block rows are still needed for updating the lower triangular part. By symmetry, these blocks reside only in the symmetric block column. A simple adaptation of SuperLU_DIST's sparse data layout and communication is as follows (see Figure 1). At each step of the outer factorization the block column of L is aggregated and replicated on the processors that own the block column. Then, each of these processors sends the entire block column along its processor row. Sending the entire block column ensures that all the processors involved in the update of the trailing submatrix are provided with all the data required. Like the LU factorization in SuperLU_DIST, we implemented a single level of pipelining scheme for better overlapping communication with computation and shortening the critical path. The implementation in the SuperLU_DIST framework is straightforward, but the drawbacks are that it sends more data than necessary and imposes synchronization of the processors along each block column.

We built a simulation module to analyze performance of this algorithm. For our experiments we used a suite of 8 matrices (**2cubes_sphere, boneS01, ship_001, smt, bmw7st_1, hood, nd3k, and thread**) and ran on an IBM Power5, and a Cray XT4. Our simulation predicts that collective communication becomes a performance bottleneck and seems to hinder scalability. In fact, it can be observed that comparing the timings of Cholesky factorization and LU factorization, when the increased communication cost outweighs the reduction in computation time, there is almost no speedup and sometimes there is even a slowdown. This effect is most noticeable for small matrices and when scaling up the number of processors, see Figure 4. However, in some cases we observed very large discrepancy between predicted and measured time where the actual running time is unexpectedly high (e.g. on 64 processors the factorization is 3 times slower that on 1 processor); we are currently investigating the causes of what appears to be a performance bug in the Cholesky algorithm.



(a) IBM Power5 (b) Cray XT4

Figure 4. Speedup achieved by Cholesky factorization relative to LU factorization applied to the same problem. Each line represents one matrix.

An alternative parallelization is block-oriented in which the sparse factor is represented by blocks (in contrast to block column in SuperLU_DIST). Each block is sent to only those processors that need it, and the communication will not be restricted along each processor row. This eliminates the need for synchronization of each column processors but may involve transferring more messages of smaller size. This was shown to work reasonably well on earlier machines [8], but it is not clear on the newer machines which have much faster processors relative to interconnect speed. Since a block-oriented approach requires different ad-hoc data structures, we plan to develop a new simulation module in order to compare the two strategies before engaging in the costly implementation.

5. Related Work

Performance modeling efforts range from developing synthetic benchmarking suites for measuring raw performance of the machines' individual components to developing high level tools and application-specific cost models for performance prediction [11,4,10,12].

Our micro-benchmarking approaches to measuring the raw latencies and bandwidths of the memory and interconnect systems are based on the widely adopted methodology of Saavedra et al. [9]. By systematically accessing memory locations with varying range and stride, it is possible to derive the latency, line size and associativity of each cache level. Similarly is done in STREAM [7] and MultiMAPS [10], with the difference that our micro-benchmarks combine multiple tests in order to estimate a larger set of parameters. Our MPI ping-pong benchmarking is similar to IMB [4], but we consciously perform the ping-pong tests with single pair and all pairs, as well as intra-node and inter-node. This covers most scenarios of the point-to-point communications occurred in real applications.

A major difference between our framework and many others appears in the way how the application performance profile is collected. A popular approach is trace-based, such as [12], which captures the addresses of the program during execution, and feed the trace to the memory simulator to derive runtime with different cache configurations. The drawbacks of this approach are that it can only predict performance for the existing applications and analyze codes that are purely memory-bound. We chose to use simulation-based approach at the application level mainly because we would like to use this framework for faster prototyping of new algorithms, in addition to analyzing existing implementations. Writing the simulation code with different algorithm choices is much easier than implementing the actual algorithms (see Algorithm 1), since we can avoid dealing with the details of the complicated sparse matrix data structures. Furthermore, an important characteristic of sparse matrix factorization algorithms is that they consist of a mixture of memory-bound and CPU-bound phases, hence any method simply based on memory-bound characteristic would likely overestimate runtime. That is why we developed a separate benchmark model for the BLAS kernels, which captures most of the CPU-bound phases. Thus, our combined memory simulator and BLAS benchmarks can predict performance of this workload mixture more accurately. Our simulation framework is also more flexible in that once we create an application module, the inputs (e.g., different sparse matrices) and the processor configurations are arguments of a simulation run, whereas a trace-based method needs a re-run of the code when the input changes.

Previously, Grigori et al. made a first attempt for developing a realistic simulation model to study the sparse factorization algorithm implemented in SuperLU_DIST [3]. Their framework is also simulation-based, containing memory simulator, detailed models of BLAS kernels, and a communication bandwidth model for varying message sizes. The model has greatly improved the prediction accuracy, but it is tailored for a specific algorithm-implementation and requires estimation of the instructions involved in the BLAS routines. Therefore, it is very difficult to adapt to different algorithms and to achieve our Goal 2).

6. Conclusions

Performance models based on simulations are very useful in several cases: to enable rapid prototyping of new algorithms, to evaluate systems design, and to model performance

that heavily depends on complex characteristics of the input. With PMTOOLS we modeled parallel sparse LU factorization and we observed errors within 15% in most cases and an average of less than 7% error across two different architectures for up to 64 processors. More importantly, the model proved to be useful for our goal of new algorithm design.

Our results indicate that the approach is promising and is applicable in modeling different and perhaps more complex applications. While the achieved accuracy in the work presented seemed sufficient for our goals, it exposed some weaknesses of the framework, namely the inaccuracy that might arise in presence of contention. To improve the robustness of our framework we plan to model the effect of resources contention as it is becoming a crucial aspect of multi-core systems. To this end we will investigate two types of contention: contention of memory resources, and contention of network resources.

Acknowledgement

This research was supported in part by the NSF contract ACI0326013, and in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. 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] P. Cicotti, X. S. Li, and Scott B. Baden. LUSim: A framework for simulation-based performance modeling and prediction of parallel sparse LU factorization. Technical Report LBNL-196E, Lawrence Berkeley National Laboratory, May 2008.

[2] Timothy A. Davis. University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices`.

[3] 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.

[4] Intel® MPI Benchmarks 3.2. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[5] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, 1991.

[6] Xiaoye S. Li and James 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.

[7] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

[8] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. *SIAM J. Scientific Computing*, 15(6):1413–1439, November 1994.

[9] Rafael H. Saavedra and Alan J. Smith. Measuring cache and tlb performance and their effect on benchmark run times. Technical report no. usc-cs-93-546, University of Southern California, 1993.

[10] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for application performance modeling and prediction. In *Supercomputing (SC02)*, Baltimore, MD, November 16-22, 2002.

[11] E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, Oct. 2004.

[12] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Supercomputing (SC07)*, Reno, California, November 10-16, 2007.