Evaluation of SuperLU on multicore architectures

Evaluation of SuperLU on multicore architectures

Xiaoye S. Li
Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
E-mail: xsli@lbl.gov

Abstract. The Chip Multiprocessor (CMP) will be the basic building block for computer systems ranging from laptops to supercomputers. New software developments at all levels are needed to fully utilize these systems. In this work, we evaluate performance of different high-performance sparse LU factorization and triangular solution algorithms on several representative multicore machines. We included both Pthreads and MPI implementations in this study and found that the Pthreads implementation consistently delivers good performance and that a left-looking algorithm is usually superior.

1. Introduction

The Chip Multiprocessor (CMP) systems will be the basic building blocks for computers ranging from laptops to supercomputers. Compared to the superscalar microprocessors exploiting high degree of instruction level parallelism, the CMP designs represent a paradigm shift that strikes better trade-offs between performance (parallelism) and energy efficiency. In theory, the CMPs can often be programmed the same way as the conventional SMPs, but the CMPs have lower memory bandwidth and abundance of fine-grained parallelism. Given the diversity of CMP designs, it is necessary, albeit difficult, to develop new software strategies at the system level as well as the application level in order to fully utilize the hardware resources.

In this paper, we study the factorization and triangular solution kernels in the sparse direct solver SuperLU [1] on two leading CMP systems. Our goal of this study is twofold. First, we would like to evaluate performance of the existing implementations on the new CMP architectures, and secondly, we would like to identify the inefficiencies in the algorithms and/or implementations and the ways to improve them for the new architectures.

2. Experimental machines

Our testing systems include an Intel Colvertown, a Sun VictoriaFalls, and an IBM Power5. The last one contains a conventional SMP node. Table 1 summarizes the key architectural features of the three systems used in this study. The sources come from [2, 3, 4].

The Intel Colvertown consists of two sockets, each with two pairs of dual-core Xeon chips (Core2Duo), with total eight processors (Dell PowerEdge 1950 dual-socket). Each core runs at 2.33 GHz with a peak performance of 9.3 Gflops (4 flops per cycle) and has a private 32 KB L1 cache. Each chip (two cores) share a 4 MB L2 cache. Each socket has access to a Front Side Bus (FSB) delivering 10.6 GB/s. The two independent FSBs are connected to the memory controller which interfaces to the DRAM channels, delivering 21.3 GB/s read memory bandwidth and 10.6 GB/s write bandwidth.

The dual-chip Sun VictoriaFalls contains 16 SPARCv9 cores, in which each CMP is a Niagara2 chip with 8 cores. Each core runs at 1.16 GHz with a peak performance of 1.16 Gflops, and has a private 8 KB L1 cache. All eight cores share a 4 MB L2 cache. In addition, each core

Table 1. Summary of the experimental machines.

	Intel Colvertown	Sun VictoriaFalls	IBM Power5 (575)
Core type	superscalar (4)	multithreaded (8)	superscalar (4)
Clock (GHz)	2.3	1.16	1.9
L1 Dcache	32 KB	8 KB	32 KB
DP Gflops	9.3	1.16	7.6
# Sockets	2	2	8
# cores/socket	4	8	1
L2 cache	4 MB/2-cores (16 MB)	4 MB/socket (8 MB)	1.92 MB /core (32 MB L3$/node)
DP Gflops	74.7	18.7	60.8
DRAM GB/s read	21.3	42.6	200
write	10.6	21.3	–
Byte/flop ratio	0.29	0.44	3.29
Power/socket (Watts)	160 (max)	84 (max)	500 (measured [3])

supports eight hardware threads, and the entire dual-chip system provides a total of 128 threads. The two sockets are interconnected via External Coherence Hubs (ECH). There are altogether 8 FBDIMM memory channels, delivering the aggregate DRAM bandwidth of 42.6 GB/s for read and 21.3 GB/s for write.

The IBM p575 Power5 is a scalable distributed-memory high-performance computing system consisting of conventional SMP nodes. The entire system (bassi at NERSC) has 111 compute nodes, each of which has 8 Power5 processors running at 1.9 GHz and has a shared-memory pool of 32 GBytes. Each processor has a peak performance of 7.6 GFlops (4 flops per cycle) and has a private 32 KB L1 cache. We use only one SMP node in this study.

3. Overview of the algorithms and implementations

3.1. Factorization in `SuperLU_MT` *using Pthreads or OpenMP*

`SuperLU_MT` [8] was first developed with Pthreads, targeted for the SMPs of modest size (e.g., 32 processors). Recently, we have added OpenMP support. The factorization uses a panel-based *left-looking* algorithm, with partial pivoting and possibly with diagonal preference to better preserve sparsity. The kernel is based on supernode-panel update, which invokes multiple calls to BLAS 2, effectively achieving BLAS 2.5 speed. Parallelization uses an asychronous and barrier-free dynamic algorithm to schedule both coarse-grained and fine-grained parallel tasks and achieve a high level of concurrency. A globally shared task queue is used to store the ready panels in the column elimination tree; and whenever a thread becomes free, it obtains a ready panel from the task queue. The coarse-grained task is to factorize the independent panels in the disjoint subtrees, while the fine-grained task is to update panels by previously computed supernodes. The scheduler facilitates the smooth transition between the two types of tasks, and maintains load balance dynamically. Figure 1(a) illustrates the left-looking factorization scheme and the dynamic scheduling method using the elimination tree.

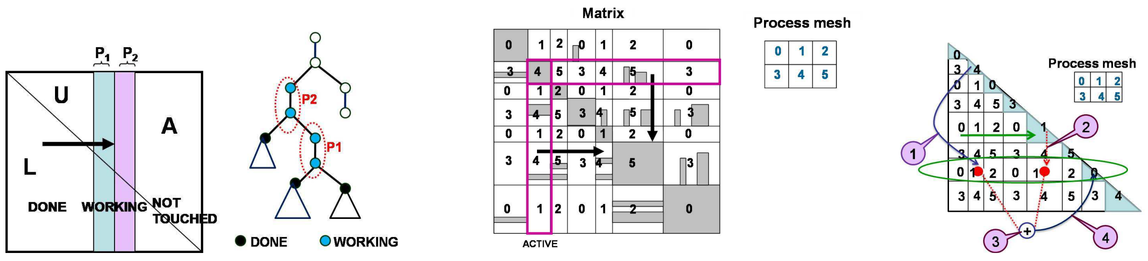
3.2. Factorization in `SuperLU_DIST` *using MPI*

In order to address the scalability issues, the parallel algorithm in `SuperLU_DIST` [5] is significantly different from that in `SuperLU_MT`. Using the supernode partition, we perform a two-dimensional (nonuniform) block-cyclic matrix-to-processor mapping. The factorization uses a block-based *right-looking* algorithm which comprises abundance of parallelism during the block

Table 2. Properties of the test matrices. Minimum degree algorithm was applied to the structure of $|A| + |A|^T$. "fill-ratio" denotes the ratio of number of nonzeros in $L + U$ over that in A; "Mean S-node" refers to an average number of columns in a supernode.

	Application	Dimension	Nonzeros in A	Fill-ratio	Mean S-node
g7jac200	economic model	59,310	837,936	40.2	1.9
stomach	duodenum model	213,360	3,021,648	45.4	4.0
torso1	2D model of torso	116,158	8,516,500	3.1	4.0
twotone	nonlinear anal. circuit	120,750	1,224,224	9.3	2.3

outer-product updates to the Schur complements. We use the elimination DAGs to identify block dependencies, and a look-ahead scheme to overlap communication with computation on the critical path. Figure 1(b) illustrates the 2D block-cyclic partition and distribution for a sparse matrix.



(a) 1D left-looking factorization. (b) 2D right-looking factorization. (c) 2D triangular solve.

Figure 1. Illustration of the parallel algorithms in `SuperLU_MT` (a) and `SuperLU_DIST` (b, c).

3.3. Triangular solution in `SuperLU_DIST`

The triangular solution phase in `SuperLU_MT` is not yet parallel; therefore we evaluate only the parallel algorithm in `SuperLU_DIST`. When solving $Lx = b$, where L is a lower triangular matrix, the ith solution component is computed as $x_i = (b_i - \sum_{j=1}^{i-1} L_{ij} \cdot x_j)/L_{ii}$. Therefore, computation of x_i needs some of the previous solution components $x_j, j < i$, depending on the sparsity structure of the i-th row of L. This sequentiality often poses scaling hurdle for a parallel algorithm. Another hurdle to achieve good performance is the much lower arithmetic density as measured by flops per byte of DRAM access or communication, compared to factorization. In `SuperLU_DIST`, the parallel triangular algorithm uses the same 2D block-cyclic distribution as used in the factorization phase. Figure 1(c) illustrates such a distribution and the solution procedure. The processes owning the diagonal blocks (called *diagonal processes*) are responsible for computing the corresponding blocks of the x components. Consider one block row of the L matrix, as circled in figure 1(b), computation of the corresponding x entry needs to proceed following the steps ①, ②, ③ , and ④. Communication is necessary when the data reside on different processes, as in steps ① and ④.

4. Experimental results

Table 2 presents the characteristics of our benchmarking matrices, which are available from the University of Florida Sparse Matrix Collection [6]. We benchmarked the Pthreads version of `SuperLU_MT`, and `SuperLU_DIST` using MPICH [7].

Table 3 shows the parallel factorization times of the two solvers on the Clovertown. The time includes both symbolic and numerical factorization. First, when we used MPICH configuration with `ch_p4` device for communication through sockets, the code slowed down significantly beyond two or four cores. After we switched to `ch_shmem` setup, we obtained respectable speedup. So

Table 3. Factorization time in seconds on Intel Clovertown.

Matrix	Threads or Tasks		1	2	4	8	Speedup
g7jac200	`SuperLU_MT`		32.78	17.91	12.41	10.60	3.1
	`SuperLU_DIST`	ch_shmem	28.10	15.95	11.06	7.57	3.9
		ch_p4	28.62	22.98	56.31	62.39	
	speedup ratio (`_MT/_DIST`)		1.00	1.03	1.01	0.80	
stomach	`SuperLU_MT`		64.38	37.15	20.39	17.24	3.7
	`SuperLU_DIST`	ch_shmem	43.45	25.91	15.81	13.64	3.4
		ch_p4	44.28	27.84	210.99	264.58	
	speedup ratio (`_MT/_DIST`)		1.00	0.99	1.10	1.10	
torso1	`SuperLU_MT`		9.43	4.92	2.87	2.20	4.3
	`SuperLU_DIST`	ch_shmem	9.43	5.83	4.55	4.76	2.2
		ch_p4	9.62	7.23	54.77	76.32	
	speedup ratio (`_MT/_DIST`)		1.00	1.12	1.49	1.99	
twotone	`SuperLU_MT`		6.80	4.05	2.32	1.83	3.9
	`SuperLU_DIST`	ch_shmem	18.08	10.17	7.55	7.21	2.1
		ch_p4	18.34	12.19	47.30	60.99	
	speedup ratio (`_MT/_DIST`)		1.00	0.95	2.26	1.86	

for a large distributed system comprising manycore chips, it is imperative to be able to configure MPICH in a *hybrid* device mode — `ch_shmem` within socket and `ch_p4` across sockets. Currently, this hybrid mode is not avaialbe. Second, we examine the single core performance. We would expect that `SuperLU_DIST` outperforms `SuperLU_MT`, because the former uses BLAS 3, whereas the latter uses only BLAS 2.5. This is true only with two matrices, g7jac200 and stomach, which have relatively denser L and U factors (the fill ratios are over 40; see table 2), and hence BLAS 3 plays a larger role. For sparser problems, the algorithms are memory-bound. We believe the worse performance of `SuperLU_DIST` is mainly due to more memory traffic of the right-looking algorithm, especially more memory write operations. Third, we examine the speedups of the two codes. The last column of table 3 shows the speedup obtained when creating eight threads or MPI tasks. The best speedup is 4.3 and is less than what we observed on conventional SMP processors [8]. After performing code profiling, we found that the overhead of the scheduling algorithm using the shared task queue and the synchronization cost using mutexes (locks) are quite small. Further study is needed to understand where the time goes. Lastly, `SuperLU_MT` usually achieves more speedup than `SuperLU_DIST`. This can be seen in the row "speedup ratio (`_MT/_DIST`)" associated with each matrix. In some cases, `SuperLU_MT` achieves a factor of two more speedup than `SuperLU_DIST`.

Table 4 shows the parallel factorization times on the Sun VictoriaFalls. The single-thread performance of `SuperLU_DIST` is usually better than that of `SuperLU_MT`. This is probably because the machine has a higher byte-to-flop ratio (see table 1) compared to Clovertown, hence it does not penalize an algorithm that is memory-bandwidth demanding, such as the right-looking algorithm in `SuperLU_DIST`. However, the coarse-grained task parallelism supported by MPI programming does not match the fine-grained multithreading architecture; MPICH uften crashes when more than 16 tasks are generated. The Pthreads program is much more robust, and `SuperLU_MT` can effectively use 64 threads. Similar to the Clovertown, `SuperLU_MT` usually achieves more speedup than `SuperLU_DIST`. In some case, `SuperLU_MT` achieves a factor of 2 more speedup than `SuperLU_DIST`.

For the parallel triangular solution, we compare the eight-core Clovertown with the eight-processor Power5 SMP node. The parallel runt imes are tabulated in table 5. The columns

Table 4. Factorization time in seconds on Sun VictoriaFalls ("f" indicates an MPI failure).

Matrix	Threads or Tasks	1	2	4	8	16	32	64	128
g7jac200	`SuperLU_MT`	480.84	244.24	126.16	68.93	40.22	28.47	**23.95**	24.80
	`SuperLU_DIST`	283.44	153.18	83.09	49.20	31.70	f	f	f
	speedup ratio (`_MT/_DIST`)	1.00	1.06	1.09	1.15	1.24			
stomach	`SuperLU_MT`	1212.97	620.58	319.85	168.04	90.01	56.51	**53.54**	62.37
	`SuperLU_DIST`	598.49	329.28	183.90	116.22	85.56	f	f	f
	speedup ratio (`_MT/_DIST`)	1.00	1.06	1.13	1.33	1.79			
torso1	`SuperLU_MT`	201.05	102.09	52.51	27.41	15.16	11.56	**10.23**	11.34
	`SuperLU_DIST`	101.68	58.25	32.53	21.83	17.06	f	f	f
	speedup ratio (`_MT/_DIST`)	1.00	1.12	1.18	1.46	2.01			
twotone	`SuperLU_MT`	113.12	60.09	31.50	17.18	11.17	8.17	**7.26**	7.90
	`SuperLU_DIST`	135.43	78.44	46.64	30.01	18.49	f	f	f
	speedup ratio (`_MT/_DIST`)	1.00	1.08	1.19	1.38	1.26			

Table 5. `SuperLU_DIST` triangular solution time in seconds on Clovertown and Power5.

		Current				**Improved**			
Matrix	Tasks	1	2	4	8	1	2	4	8
g7jac200	Clovertown	0.39	0.79	0.76	2.94	0.30	0.28	0.29	0.44
	Power5	0.61	0.68	0.46	0.39	0.43	0.39	0.28	0.22
stomach	Clovertown	0.93	1.21	3.79	6.74	0.77	0.74	0.53	0.90
	Power5	1.24	1.29	0.86	0.75	0.92	0.77	0.59	0.46
torso1	Clovertown	0.28	0.52	1.98	3.22	0.21	0.29	0.32	0.45
	Power5	0.31	0.41	0.27	0.24	0.22	0.24	0.18	0.13
twotone	Clovertown	0.46	1.51	4.42	7.52	0.32	0.44	0.47	0.80
	Power5	0.71	0.97	0.69	0.58	0.44	0.52	0.44	0.34

labeled "Current" correspond to the current released code, and the columns labeled "Improved" refer to the new implementation as a result of this study. As seen in the table, the current code runs much more slower with more cores involved on the Clovertown. A similar trend was also observed on the VictoriaFalls. After profiling various parts of the code, we found that the slowdown is due to many calls of `MPI_Reduce`, which can take over 75% of the time using eight cores. In figure 1(b), each diagonal process needs to know which off-diagonal processes will have sum contributions to be sent to the diagonal process. To compute this count, every process holds a 0/1 flag indicating whether this process has nonzero blocks. Then all the processes in each row communicator perform an `MPI_Reduce` (by SUM) over the flags, with root being the diagonal process. Overall, each block row corresponds to one such reduction operation.

The improvement we have made is the following. Instead of performing many reductions with one integer, we allocate a flag array of integers, the size of which is the number of block rows owned by each process. Each entry is the flag associated with one block row. Then all the processes in the respective process row perform *only one* reduction operation on this flag array. This has greatly reduced the memory and communication latency cost. On eight-core Clovertown, the improvement is significant, ranging from 6- to 9-fold. Even on the conventional SMP node, such as eight-CPU Power5, we also obtained 63% to 84% improvement.

Acknowledgments

This research was supported in part by the Director, Office of Advanced Scientific Computing Research of the U.S. Department of Energy undercontract number DE-AC03-76SF00098. We used the multicore clusters with the PSI project and the RADlab at UC Berkeley, and the resources at the National Energy Research Scientific Computing Center.

References

[1] Li X S 2005 overview of SuperLU: Algorithms, implementation, and user interface *ACM Trans. Mathematical Software* **31(3)**:302–325
[2] Phillips S 2007 Victoriafalls: Scaling highly-threaded processor cores *HOT CHIPS 19: A Symposium on High Performance Chips* (Stanford, CA)
[3] Shalf J 2008 (Private communications)
[4] Williams S 2008 (Private communications)
[5] Li X S and Demmel J W 2003 SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems *ACM Trans. Math. Soft.* **29(2)**:110–140
[6] Davis T A 1997 *University of Florida Sparse Matrix Collection* http://www.cise.ufl.edu/research/sparse/matrices
[7] *MPICH – A portable implementation of MPI* http://www-unix.mcs.anl.gov/mpi/mpich1/
[8] Demmel J W, Gilbert J R, and Li X S 1999 An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Analysis and Applications* **20(4)**:915–952