Evaluation of Sparse LU Factorization and Triangular Solution on Multicore Platforms*

Xiaoye S. Li

Lawrence Berkeley National Laboratory, MS 50F-1650,
One Cyclotron Road, Berkeley, CA 94720, USA.
Tel: (510) 486-6684. Fax: (510) 486-5812, Email: 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 include both pthreads and MPI implementations in this study, and found that the pthreads implementation consistently delivers good performance and 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 the case of multicore architecture, high performance is achieved by replicating the execution units on a single die while keeping the clock rate (hence power consumption) relatively low. In the case of multithreaded architecture, high throughput is achieved by providing multiple sets of hardware thread contexts for each FPU and simultaneously executing multiple streams of instructions without relying on speculation. Multithreading can effectively hide instruction and cache latency. 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 several kernel algorithms associated with sparse direct solvers on a couple of leading CMP systems. Direct solvers based on matrix factorizations are among the most reliable methods or preconditioners for solving

* This research was supported 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.

sparse linear and eigen systems. They are often the computational bottlenecks in large-scale computer modeling codes. Over the past decade or so, we have been developing new algorithms to exploit advanced high-performance, large-scale parallel computers. Our algorithm research has led to the software package called SuperLU [4, 7], which is widely used in research and industry.

Previously, Williams et al. [14] performed extensive study and optimization of sparse matrix-vector multiply (SpMV) on several leading CMP systems. In SpMV, the matrix A needs to be read only once, hence the ratio of flops to memory accesses is $\mathcal{O}(1)$. The operation is largely memory-bound, since there is hardly any reuse, They develoepd various blocking strategies exploiting different hardware components, including thread blocking, register blocking, cache and local store blocking, and TLB blocking. In contrast, the factorization algorithm need to access the matrix A, the L and U factors multiple times, which exhibit higher reuse. But the ratio of flops to memory accesses changes throughout the elimination process. In early stages of elimination, the factors are sparser and workload is memory-bound. Whereas in later stages, the factors are denser; level 3 BLAS are appropriate and hence the workload is compute-bound. The change of arithmetic density makes it harder to develop uniform strategies for performance optimization.

Our goal of this study is two-fold. Firstly, 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.

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 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 HPC 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 only use one SMP node in this study.

Table 1 summarizes the key architectural features of the three systems used in this study. Figure 1 shows the simplified block diagrams of the two CMP systems. The sources come from [11, 12, 14].

Several characteristics in Table 1 are worth noting. Compared to the IBM SMP node, the cores-to-DRAM bandwidths are considerably lower on the multicore systems. In addition, the write bandwidth on both multicore systems is only half of the read bandwidth. The byte-per-flop ratio shows the balance of the memory bandwidth versus floating-point speed, with larger ratio indicating higher bandwidth relative to core speed. The CMP systems are clearly worse than conventional SMPs in this regard, implying that algorithms demanding larger memory bandwidth will be penalized in performance. The conventional SMPs usually have more complex designs and consume more power, see the last line in the table. The quoted number for p575 was measured while running a full computational workload, which is much lower than the manufacturer's peak power rating [12].

Table 1. Summary of the experimental machines.

Systems	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 [12])

3 Overview of the Algorithms and Implementations

Over the years, we have been deveoping the SuperLU suite of libraries, with different variants of sparse Gaussian elimination targeted for shared or distributed

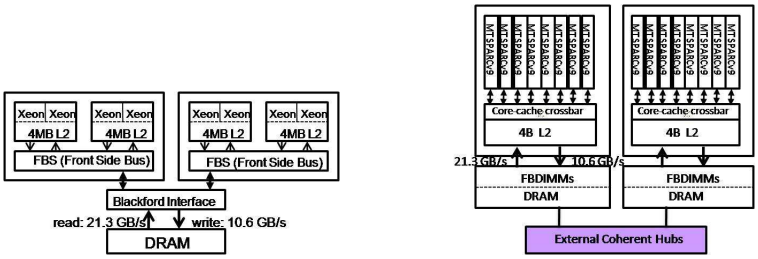

Fig. 1. High level block diagrams of Intel Clovertown (left) and Sun VictoriaFalls (right).

memory high performance machines [7]. Below, we briefly summarize the algorithmic and implementational features of the two variants used in this study.

3.1 Factorization in `SuperLU_MT` using Pthreads or OpenMP

The initial target platforms of `SuperLU_MT` were the SMPs of modest size (e.g., 32 processors). It was first developed with Pthreads, and recently, we have added OpenMP support. The earlier tests on a number of commercially popular SMPs, such as Sun, DEC Alpha, SGI Origin, and Cray C90/J90, demonstrated excellent speedups [3, 6]. Pleasantly, we will show that this code is equally suitable for the modern multicore machines. The algorithmic features and parallelization techniques are outlined below:

- Use panel-based *left-looking* factorization, with partial pivoting and possibly with diagonal preference to better preserve sparsity. Use supernode-panel update kernel to effectively use Level 3 BLAS.
- Use an asychronous and barrier-free dynamic algorithm to schedule both coarse-grain and fine-grain parallel tasks to 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-grain task is to factorize the independent panels in the disjoint subtrees, while the fine-grain 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 2 illustrates the left-looking factorization scheme, and the dynamic scheduling method using the elimination tree.

3.2 Factorization in `SuperLU_DIST` using MPI

The target platforms of this code are the massively parallel distributed memory computers [8]. Previously, we tested this code on a number of HPC platforms, including Cray T3E, IBM SP, and various Linux clusters. Good scalability was

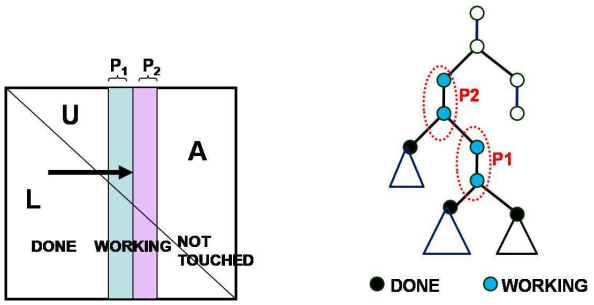

Fig. 2. Panel-based left-looking algorithm in superlumt.

demonstrated up to one thousand processors. In order to address the scalability issues, the parallel algorithm is significantly different from that in `SuperLU_MT`. The main differences are in pivoting strategy and matrix distribution, which are summarized below.

- Use block-based *right-looking* factorization, which comprises abundance of parallelism during the block outer-product updates to the trailing submatrices. According to the supernode partition, perform a two-dimensional (nonuniform) block-cyclic matrix-to-processor mapping. Use the elimination DAGs to identify task and block dependencies, and a look-ahead mechanism to better overlap communication with computation and shorten the critical path.
- Before factorization, pre-permute the rows of the matrix so that the diagonal has entries of large magnitude, using a weighted bipartite matching algorithm from MC64 [5]. During factorization, allow single precision perturbation to the small diagonal entries.

Figure 3 illustrates the 2D block-cyclic partition and distribution for a sparse matrix.

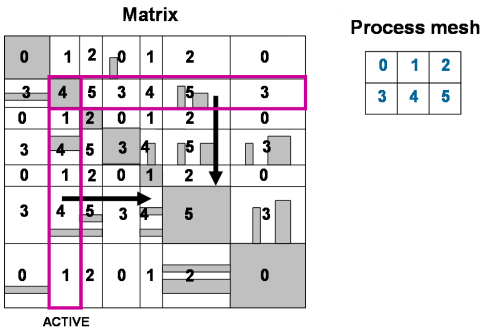

Fig. 3. Block-based right-looking algorithm in `SuperLU_DIST`.

3.3 A note on SuperLU_MT — symmetric mode

In order to conduct direct comparision between SuperLU_MT and SuperLU_DIST, we have added a new algorithmic choice for SuperLU_MT, which is called *symmetric mode*. As is known, with partial pivoting (SuperLU_MT), it is better to use A^TA-based column ordering strategy to preserve sparsity, since pattern-wise, the Cholesky factor of A^TA upper bounds the LU factors in the decomposition $PA = LU$, for any row permutation P. However, in the case of satic pivoting where pivots are selected before hand (SuperLU_DIST), no row interchanges are made during factorization, then the A^TA-based upper bound becomes too loose. Therefore, we can use a tighter upper bound based on $A + A^T$. The difference in the amount of fill using an A^TA-based sparsity ordering or an $(A + A^T)$-based one can be more than a factor of two. The new symmetric mode option in SuperLU_MT contains an algorithm that is similar to the one in SuperLU_DIST — it uses MC64 to perform static numerical pivoting, an $(A+A^T)$-based symmetric sparsity ordering, and single precision diagonal perturbations when needed.

All our experimental results used the symmetric mode in SuperLU_MT. Thus, the amount of fill and number of floating-point operations are roughly the same with both solvers.

3.4 Triangular solution in SuperLU_DIST

The triangular solution phase in SuperLU_MT is not yet parallel, therefore we will evaluate only the parallel algorithm in SuperLU_DIST. In $Lx = b$, where L is a lower triangular matrix, the i-th solution component is computed as

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij} \cdot x_j}{L_{ii}} \ .$$

Therefore, computation of x_i needs some or all of the previous solution components $x_j, j < i$, depending on the sparsity pattern 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 the current implementation of SuperLU_DIST, the parallel triangular algorithm uses the same 2D block-cyclic distribution as used in the factorization phase. Figure 4 illustrate such a distribution and the solution process. The processes owning the diagonal blocks (called *diagonal processes*) are responsible for computing the corresponding blocks of the x components. When x_j is needed in $L_{ij} \cdot x_j$, and the owners of x_j and L_{ij} are different, x_j's processor needs to send it to the processor of L_{ij}, see ① in Figure 4. In case of ②, no communication is needed because both x_j and L_{ij} reside on the same processor, i.e. processor 1. After receiving the needed x_j entries, each processor proceeds with local summation, i.e., step ③ in Figure 4. Finally, the local sums are sent to the diagonal processor which performs the division, see ④ in the figure.

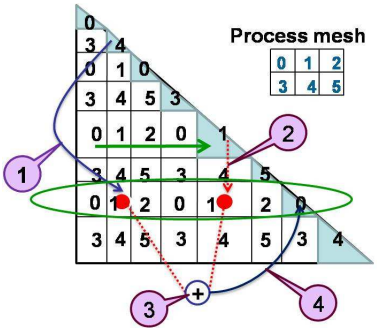

Fig. 4. Block-based triangular solution.

3.5 Entire solvers

Since the numerical pivoting methods are different in the two solvers — partial pivoting in `SuperLU_MT` and static pivoting in `SuperLU_DIST`, the high level structure of the two codes are different. In case of partial pivoting, the fills are generated dynamically, so the symbolic factorization step cannot be separated from numerical factorization. Whereas with static pivoting, we can separate symbolic and numerical factorization steps. Figure 5 summarizes the major steps of the two solvers and highlights the differences between them.

`SuperLU_MT`

1. Sparsity ordering
2. Factorization ... interleaveing:
   2.1) partial pivoting
   2.2) symbolic factorization
   2.3) numerical factorization (BLAS 2.5)
3. Triangular solution

`SuperLU_DIST`

1. Static pivoting
2. Sparsity ordering
3. Symbolic factorization
4. Numerical factorization (BLAS 3)
5. Triangular solution

Fig. 5. Major steps in the entire solvers.

4 Experimental results

Table 2 presents the characteristics of our benchmarking matrices, which are available from the University of Florida Sparse Matrix Collection [2]. In the following subsections, we will present the parallel runtimes and analysis. We benchmarked the Pthreads version of `SuperLU_MT`, and `SuperLU_DIST` using MPICH [9].

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

4.1 Characterization from hardware performance counters

Given that the memory system performance plays increasingly significant role on the CMP architectures and with the sparse matrix algorithms, it would be more relevant to quantify the memory access patterns of the different algorithms than to count the flops alone. For simpler kernels like SpMV, it is possible to count manually. For complex codes, we need to resort to performance analysis tools.

The first tool we used is PAPI [10] which provides an API to access machines' hardware counters. As a first cut, we examine the load and store instruction counts, which are independent of the cache memory organization. Table 3 compares the counts of the left-looking (`SuperLU_MT`) and right-looking (`SuperLU_DIST`) factorization algorithms. It is clear that right-looking algorithm incurs many more load and store instructions, typically an order of magnitude more.

Table 3. Factorization load and store instruction counts (billions), reported by PAPI.

	LOAD		STORE	
	`SuperLU_MT`	`SuperLU_DIST`	`SuperLU_MT`	`SuperLU_DIST`
g7jac200	1.2	27.7	0.3	8.2
stomach	0.8	52.0	0.3	10.8
torso1	9.1	17.9	2.8	4.5
twotone	1.2	18.6	0.2	8.4

Although the load/store instruction count indicates the superiority of the left-looking algorithm in the sense of program's static behavior, we are also interested in the temporal behavior of the codes while running on an actual machine. Unfortunately, the two multicore machines do not yet have proper PAPI support for such study. So we used the CrayPat performance tool provided on the Cray XT systems [1]. CrayPat uses PAPI's counters to collect raw data, and then computes a variety of derived quantities which are easy to understand. The machine we used is the Cray XT4 installed at NERSC. Each node consists of a 2.6 GHz dual-core AMD Opteron processor, sharing 4 GBytes of memory. Each core has a 64KB L1 data cache of and a 1MB L2 cache. The L2 cache is

a victim cache which holds only the cache lines evicted from L1, whereas most data loaded from memory go directly to L1. We used only one core to run the codes and collected data shown in Table 4, and the data are for the entire solvers. We report two metrics: "Mem-to-D1" measures the amount of data transferred between memory and L1 data cache, and "L2-to-Mem" measures the data traffic between L2 and memory. In both metrics, we see that `SuperLU_DIST` requires considerably larger amount of data transfer.

Table 4. The solvers' memory traffic (billion bytes) reported by CrayPat.

	Mem-to-D1		L2-to-Mem	
	`SuperLU_MT`	`SuperLU_DIST`	`SuperLU_MT`	`SuperLU_DIST`
g7jac200	10.7	17.5	3.9	15.1
stomach	25.1	24.9	16.0	16.3
torso1	3.0	7.4	4.7	11.6
twotone	1.9	7.8	1.2	7.0

Lastly, we examine the flop-to-load (store) ratio in the triangular soution phase. We compare the metric for the two distinct stages of the solver, one is "ordering + factor" and the other is "tri-solve". In Table 5, we report the respective ratios of flop-to-load and flop-to-store. As can be seen, in both metrics, the triangular solution phase has much smaller flop density, sometimes can be more than an order of mangnitude lower than the other part of the solver.

Table 5. Ratio of flops over load or store instructions in the triangular solution of `SuperLU_DIST`, compared with the rest of the program.

	LOAD		STORE	
	ordering + factor	tri-solve	ordering + factor	tri-solve
g7jac200	0.86	0.14	2.89	0.24
stomach	1.35	0.24	6.49	0.47
torso1	0.75	0.21	2.95	0.35
twotone	0.30	0.06	0.67	0.09

4.2 Runtime

In the factorization codes of both solvers, the BLAS routines could take more than 30-40% of the time. Therefore the BLAS speed is a key performance bound. In sparse codes, the matrix size for BLAS calls is usually small. The kernel in `SuperLU_MT` is "BLAS 2.5", where we perform multiple DGEMV calls with different vectors while keeping the matrix in cache. Therefore, we usually keep the matrix size bounded by 200×100. The kernel in `SuperLU_DIST` is BLAS 3

(mostly DGEMM). In order to maintain good load balance, we use even smaller block sizes, such as 50×50. In Figure 6, we plot the performance (Gflops rate) of DGEMV and DGEMM on Clovertown and VictoriaFalls. In each case, we used the vendor's high performance mathematical libraries — Intel's MKL and Sun's SunPerf.

Recall that each core processor of VictoriaFalls is hardware-multithreaded. But without explicit parallelization (e.g., threading) at the software level, DGEMM only achieves less than one-third of the peak performance. That is, a single threaded program is incapable of fully utilizing the resources provided by a multithreading architecture.

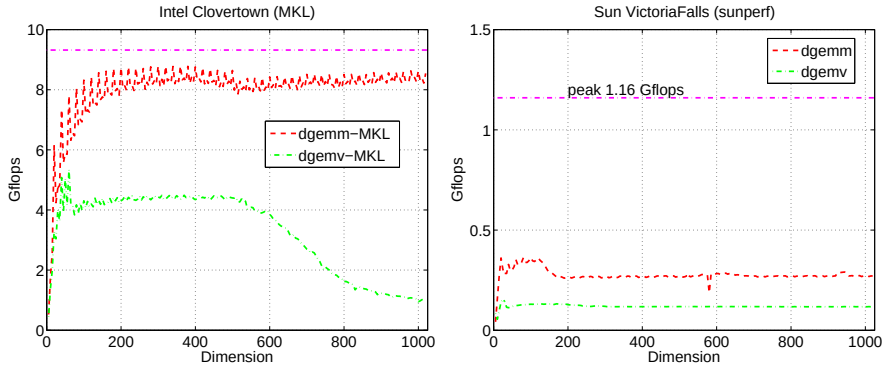

Fig. 6. BLAS performance on Intel Clovertown (left) and Sun VictoriaFalls (right). The top dashed line shows the core's peak performance.

Table 6 shows the parallel factorization times of the two solvers on Intel Clovertown. *For fair comparison, the time includes both symbolic and numerical factorization, because it is not possible to separate these two steps in* `SuperLU_MT`*, see Figure 5.*

First we note that MPICH can be configured with either `ch_shmem` device for shared memory processors, or `ch_p4` device for communication through sockets on distributed memory machines. We first used the default `ch_p4` configuration on the Clovertown cluster, and found that the code slowed down significantly beyond two or four cores. After we switched to `ch_shmem` setup, we obtained respectable speedup. Therefore, for a large distributed system comprising many-core chips, it is imperative to be able to configure MPICH in a *hybrid* device mode — `ch_shmem` mode within socket and `ch_p4` mode across sockets. Currently, the hybrid mode is not avaialbe.

Secondly, 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. We see that this is true only with two matrices, g7jac200 and stomach, which have relatively denser L and U factors (the fill ra-

tios 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, see the measures presented in Section 4.1.

Thirdly, we examine the speedups of the two codes. The last column of Table 6 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 [3]. 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.

In addition, 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 6. 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	

Table 7 shows the parallel factorization times on the Sun VictoriaFalls. Recall that this system has 16 eight-way harware-threaded cores, and altogether we can have up to 128 threads. The single-thread performance of SuperLU_DIST is usually better than that with 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-grain task parallelism supported by MPI programming does not match the fine-grain multithreading architecture — MPICH often

crashes when more than 16 tasks are generated. The Pthreads programming is much more robust, and SuperLU_MT can effectively use 64 threads. Similar to Clovertown, 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 case, SuperLU_MT achieves a factor of 2 more speedup than SuperLU_DIST.

We see that SuperLU_MT achieves nearly perfect speedups for the first four to eight threads. This may be related to the Sun Solaris' round-robin scheduling policy which schedules multisocket first, then multicore, then multithreads [13]. With this order, the first few threads are spread across different sockets, and do not have much memory bus contention.

Table 7. Factorization time in seconds on Sun VictoriaFalls. "f" indicates that an MPI failure occurred.

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			

We now evaluate performance of the parallel triangular solution algorithm in SuperLU_DIST. We compare the eight-core Clovertown with the eight-processor Power5 SMP node. The parallel runtimes are tabulated in Table 8. The columns labeled "Current" correspond to the current implementation, and the columns labeled "Improved" refer to the new implementation as a result of this study.

It is very disapointing that on the Clovertown, the current code runs much more slower with more cores involved. A similar trend was also observed on the VictoriaFalls. After we profiled various parts of the code, we found that the slowdown is due to many calls of MPI_Reduce; in fact, on eight cores, MPI_Reduce can take over 75% of the time. Consider one block row of the L matrix, as circled in Figure 4, the diagonal process 0 needs to know which off-diagonal processes (1 and 2) will have sum contributions to be sent to process 0. To compute this count, every process holds a 0/1 flag depending whether this process has nonzero blocks. Then, all the processes in each process row perform an MPI_Reduce (by SUM) over the flags, with diagonal process being the root. 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 or 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 restpectable improvement, from 63% to 84%.

Note that the Clovertown time still does not scale as well as the Power5 time. Further investigation is needed in the future.

Table 8. `SuperLU_DIST` triangular solution time in seconds on Intel Clovertown and IBM 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

5 Final remarks

We performed preliminary study of the SuperLU sparse direct solvers on representative multicore achitectures. Using the performance analysis tools such as PAPI and CrayPat, we gave quantitative measures of both static and temporal memory access bahavior. We found that the left-looking factorization incurs much less memory traffic than the right-looking one, therefore, it performs better on the CMP systems with limited memory bandwidth. We believe this performance characteristics is very likely associated with the other right-looking algorithm variants, such as a multifrontal algorithm. We also quantified that the arithmetic density of the triangular solution algorithm can be over an order of magnitude lower than the preprocessing and factorization algorithms. The Pthreads code is usually more robust and delivers consistently better performance than the MPI code, particularly on a multicore+multithreading architecture, such as Sun VictoriaFalls. These suggest that it will be beneficial to use hybrid programming model, to design hybrid algorithms, and to provide hybrid device mode for MPICH.

In the future, we plan to continue using the performance tools to refine our understanding of multicore scaling, and find ways to enhance performance.

Acknowledgments

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. We are grateful to John Shalf, Rich Vuduc and Sam Williams for their help in using these machines and understanding the architectural features.

References

1. CrayPatCray Performance Analysis Tools. `http://docs.cray.com/books/S-2376-41/S-2376-41.pdf`.
2. Timothy A. Davis. University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices`.
3. James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.
4. James W. Demmel, John R. Gilbert, and Xiaoye S. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: September 2007.
5. Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Analysis and Applications*, 22(4):973–996, 2001.
6. Xiaoye S. Li. Sparse Gaussian elimination on high performance computers. Technical Report UCB//CSD-96-919, Computer Science Division, U.C. Berkeley, September 1996. Ph.D dissertation.
7. Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Mathematical Software*, 31(3):302–325, September 2005.
8. 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.
9. MPICH - A Portable Implementation of MPI. `http://www-unix.mcs.anl.gov/mpi/mpich1/`.
10. PAPI - Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.
11. Stephen Phillips. Victoriafalls: Scaling highly-threaded processor cores. In *HOT CHIPS 19: A Symposium on High Performance Chips*, Stanford, California, August 19-21, 2007.
12. John Shalf. Private communications.
13. Samuel Williams. Private communications.
14. Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing (SC)*, Reno, California, November 10-16, 2007.