Performance of Multicore LUP Decomposition

Nathan Beckmann Silas Boyd-Wickizer

May 13, 2010

Abstract

This paper evaluates the performance of four parallel LUP decomposition implementations. The implementations vary widely: two are implemented in Cilk++, but use different underlying algorithms to perform LUP decomposition; one written in Fortran; and the other is written in C++ with pthreads. This diversity allows us to evaluate which implementation techniques and properties are important to achieve high performance on modern multicore systems. The result of our evaluation indicates the performance of parallelized LUP decomposition is dependent on efficient use of on-chip and off-chip memory resources. Furthermore, dynamic scheduling is an important factor in achieving good performance in multiprogrammed environments.

1 Introduction

Multicore systems offer abundant compute cycles along with slow off-chip DRAM access. To avoid the cost of reading from DRAM, hardware architects provide fast on-chip caches. A key challenge is designing an algorithm so that its implementation reads from and writes to local on-chip caches. This challenge is especially difficult when the algorithm's implementation operates on an amount of data that exceeds the size of on-chip caches. One such application is LU decomposition with paritial pivoting (or simply LUP decomposition), which is an application commonly used in numerical analysis. This paper reports on our experience optimizing and evaluating the performance of four multicore LUP decomposition applications.

LUP decomposition decomposes a matrix into the product of a Lower triangular matrix and Upper triangular matrix, and produces a Permutation matrix. One way to achieve good performance with a large input matrix is to divide the input matrix into small blocks that fit in on-chip caches, then operate on each block in turn. This provides good performance because each operation on a block typically reads the block multiple times, but the penalty for reading the block from DRAM is only incurred once.

In principle the blocking approach sounds simple. The memory systems of modern multicore systems, however, have numerous complexities that must be considered to achieve good performance. For example, caches hold memory at the granularity of cache lines, which are typically at least 64 bytes. If a LUP application fails to partition a matrix along cache line boundaries the result might be that multiple cores write to the same cache line, causing the cache line to "bounce" between on-chip caches. Writing to a cache line in a local cache can be as much as $100\times$ faster than writing to a cache line that has bounced into another cache.

Multiprogrammed environments create additional complexities for LUP decomposition applications. In an multiprogrammed environment, a LUP application must share the CPU cycles and caches with other applications. A common result for LUP applications that assume exclusive access to CPU resources is poor performance caused by cache misses or load imbalance.

The main contribution of this paper is a performance evaluation of four LUP decomposition implementations, two written in Cilk++, one written in C++, and the PLASMA library [4], which is an "industrial grade" LUP application written in Fortran. Our results demonstrate the following: (1) the performance of our fastest Cilk++ application is competitive with PLASMA on a number of architectures; (2) careful matrix layout that accounts for hardware complexities is important for achieving good performance; (3) PLASMA uses more memory than our Cilk++ and C++ applications; (4) the relatively poor performance of one Cilk++ application and the C++ application is, at least in part, due to poor utilization of on-chip caches; and (5) compared to PLASMA and the C++ application, the performance of Cilk++ applications degrade gradually in a multiprogrammed environment as more processes compete for CPU resources.

The rest of this paper is organized as follows. Section 2 details the design of the two LUP decomposition algorithms used in the LUP implementation we evaluate. Section 3 describes the four LUP implementations. Section 4 analyzes experimental results from benchmarking all four implementations. Section 5 discusses future work and Section 6 concludes.

2 Algorithm Design

We evaluate the performance of LUP decomposition applications, each of which is based on one of two algorithms: the *right-looking* algorithm and the *recursively-partitioned* algorithm. The input to both algorithms is a $n \times m$ matrix A, where $n \geq m$, and the result is a

lower triangular matrix L, an upper triangular matrix U, and a permutation matrix P, such that $PA = LU$. This section provides a brief overview of both algorithms[1] and describes how we parallelize both algorithms.

2.1 Right-looking

The right-looking algorithm is used for LUP decomposition in popular linear algebra libraries, such as PLASMA and LAPACK [3]. The algorithm works iteratively, decomposing r columns and r rows of A at every iteration. At the start of the kth iteration

$$PA = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

where A_{22} is an r order square matrix, A_{32} is a $(n - kr) \times r$ matrix, and A_{23} is a $r \times (m - kr)$ matrix. The right-looking algorithm has decomposed the first $(k-1)r$ rows and $(k-1)r$ columns of A. The kth iteration performs the following steps:

1. Decompose

$$P_2 \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22}$$

If $r = 1$, perform pivoting and scaling, otherwise decompose using the right-looking algorithm with $r = 1$.

2. Permute

$$P_2 \begin{bmatrix} A_{23} \\ A_{32} \end{bmatrix} \to \begin{bmatrix} A_{23} \\ A_{32} \end{bmatrix}.$$

3. Permute

$$P_2 \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \to \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix}.$$

4. Solve the triangular system for U_{23}

$$\begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{23} = A_{23}.$$

5. Update $A_{33} - L_{32}U_{23} \to A_{33}$.

6. Repeat Step 1 until $kr = m$.

[1] We derived these descriptions from [6].

2.2 Recursively-partitioned

The recursively-partitioned algorithm was originally described by Toledo in [6]. Toledo demonstrated that the recursively-partitioned algorithm generates asymptotically less memory traffic than the right-looking algorithm, even when the block size r of the right-looking algorithm is tuned to an optimal value. The algorithm works recursively, processing the block matrix

$$PA = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{11} is an order $m/2$ square matrix. The algorithm performs the following steps:

1. If $m = b$, where b is a base case, factor

$$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}$$

using the right-looking algorithm with r = 1 and return.

2. Else, recursively factor

$$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}.$$

3. Permute

$$P_1 \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \to \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}.$$

4. Solve the triangular system for U_{12}

$$A_{12} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{12}.$$

5. Update $A_{22} - L_{21}U_{12} \to A_{22}$.

6. Recursively factor $P_2A_{22} = L_{22}U_{22}$.

7. Permute $P_2L_{21} \to L_{21}$.

2.3 Parallelizing

We parallelize the right-looking and recursively-partitioned algorithms using parallel algorithms to perform matrix multiplication, forward substitution to solve triangular systems, pivoting, and scaling. There are opportunities to achieve more parallelism, for example by performing Step 1 in parallel with Step 5 of the right-looking algorithm after Step 5 updates the first r rows and r columns of A_{33}. However, our empirical results indicate that it is more important to optimize Step 5 of both right-looking and recursively-partitioned, because it has the most parallelism.

3 IMPLEMENTATION

This paper reports on four implementations of LUP decomposition. We wrote one version of the right-looking algorithm in Cilk++ (referred to as Cilk++ right), wrote another version of the right-looking algorithm in C++ using pthreads (referred to as pthreaded right), ported Kuszmaul's implementation of the recursively-partitioned algorithm [5] from Cilk-5 to Cilk++ (referred to as Cilk++ recursive), and use PLASMA's LUP decomposition, which is written in Fortran, without any modifications.

All four implementations operate on A in place, replacing the lower triangular of A with L and the upper triangular of A with U, and return an array permutations that, when applied to an identity matrix, produce a permutation matrix P. All matrices use a row major layout with an optimization described in Section 4.2. We use 64×64 as our base case value in Cilk++ right, Cilk++ recursive, and pthreaded right. All four implementations use the Goto BLAS [2] linear algebra library for performing matrix multiplication. Table 1 gives the breakdown of the number of lines of code for each implementation.

The parallelized Cilk++ matrix multiplication, pivoting, and scaling functions are all written as divide-and-conquer algorithms. Forward substitution is written using `cilk_for`. We found that the matrix multiplication algorithm presented in [1] performed poorly because most the of matrix multiplication computations involve non-square matrices. The result of using the algorithm in [1] was that the base case essentially multiplied a row vector and a column vector. For large matrices A, reading the column vector resulted in poor cache performance and low parallelism. We use an alternative matrix multiplication algorithm provided by Kuszmaul, which recursively divides matrices along the longest dimension until the algorithm reaches a base case. With this algorithm, the base case multiplies matrices that have closer to square dimensions.

The pthreaded implementation of right-looking works by creating P worker threads, partitioning the matrix A into blocks, and statically assigning each worker thread to distinct set of blocks. Every thread updates only the blocks of A they are assigned to. For example, when executing Step 5 of the right-looking algorithm, threads update only the blocks of A_{33} that the schedule assigns to them.

A "master" thread is responsible for controlling the execution of LUP decomposition. The master thread signals the other threads to perform the matrix operations by writing to a shared memory location, which the other threads poll. For example, to execute Step 5 of the right-looking algorithm, the master thread signals the other threads to update A_{33} by subtracting $L_{32}U_{23}$. Threads will execute only the multiplications necessary to update the blocks of the matrix they are assigned to. The result of this design is that the implementation of right-looking algorithm is almost identical to the Cilk++ implementation, except with different calls to matrix functions.

Implementation	LUP	Matrix
Cilk++ right	121	257
Cilk++ recursive	111	238
Pthreaded right	134	934
PLASMA	143	269

Table 1: A breakdown of the number of lines of code for the four implementations of LUP decomposition. The "LUP" column refers to the number of lines of code to implement the algorithms described in Section 2. The "Matrix" column refers to the number of lines of code to implement parallel matrix operations.

4 EVALUATION

We evaluated the four implementations of LUP to explore what factors had the greatest impact on performance. We ran all experiments on matrices composed of 8-byte double precision floating point values. To see the impact of different machine architectures, we benchmarked on three 64-bit machine architectures. These revealed significant performance heterogeneity even using the same ISA. Cache behavior was found to be a dominant factor in performance for certain matrix sizes, and a simple fix improved performance considerably. Because of this, we believe memory performance to be the dominant factor in LUP performance on multicore architectures.

With good cache behavior, Cilk++ implementations were competitive with PLASMA. Both right-looking implementations suffered performance problems on all but one architecture, which we believe to be related to memory performance. The one architecture that performs well is the newest, and has a significantly improved memory architecture.

4.1 Architectural Heterogeneity

Name	Configuration
AMD16	Quad-quad AMD Opteron 8350 at 2.0 GHz, 64 GB RAM.
Intel16	Quad-quad Intel Xeon E7340 at 2.0 GHz, 16 GB RAM.
Intel8	Dual-quad Intel Xeon E5530 at 2.4 GHz, 12 GB RAM.

Table 2: Machine configurations used in benchmarks.

Table 2 shows the machine configurations used in these studies. A *Xen* suffix indicates the machine was running a Xen-modified Linux kernel. This was found to exhibit strange behavior in some cases, so may impact perfor-

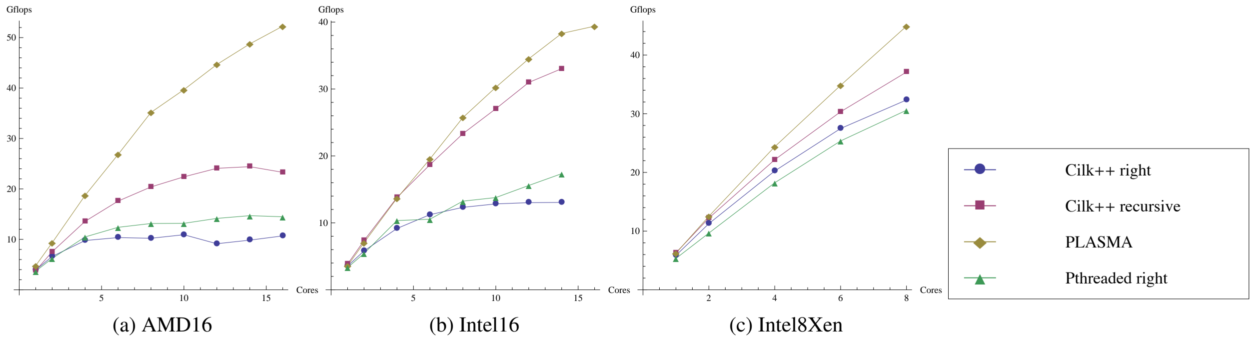


Figure 1: Performance and scaling of implementations on each machine configuration for a 8192×8192 matrix. Gigaflops achieved by each implementation is plotted against the number of cores used.

Implementation	LUP performance (Gflops)			
	AMD16	Intel16	Intel16Xen	Intel8Xen
Cilk++ recursive	17.2	19.6	17.4	32.5
Cilk++ right	7.72	8.53	7.38	23.2
Pthreaded right	12.5	11.2	10.8	22.1
PLASMA	28.7	21.5	20.6	31.1

Table 3: Summary of performance on 4096×4096 matrices running on 8 cores.

mance in other ways. All tests were run in *dom0* (not inside a virtual machine).

Table 3 summarizes the performance of each implementation on each machine configuration. These results are for a 4096×4096 matrix, which is smaller than the matrices used to gather other results in this section. LUP was ran on 8 cores for each configuration so that comparisons could be drawn with Intel8Xen.

Figure 1 shows the performance and scaling of the implementations on each machine configuration. These results are for a 8192×8192 matrix. Figure 1 and Table 3 show significant performance differences for each configuration. PLASMA performs the best on AMD16, but Cilk++ recursive is competitive on all other configurations. Intel16 and Intel16Xen perform similarly at 8 cores, with small degradation on the Xen configuration. On AMD16 and both Intel16 configurations, the right-looking implementations perform poorly. On Intel8Xen, however, they perform much better (although still significantly worse than Cilk++ recursive and PLASMA).

Finally, Figure 2 shows the odd scaling problems with Xen kernels alluded to earlier. PLASMA performance flat-lines going from 10 to 11 cores on Intel16. We are not sure what causes this, but it raises the possibility of other performance interference from Xen kernels. The only hint we have is that PLASMA uses more memory than our implementations. Table 4 shows the maximum amount of memory each implementation uses to decompose a 8192×8192 matrix. We measured memory usage using statistics provided by Linux. We suspect that PLASMA's poor performance is caused when PLASMA uses more

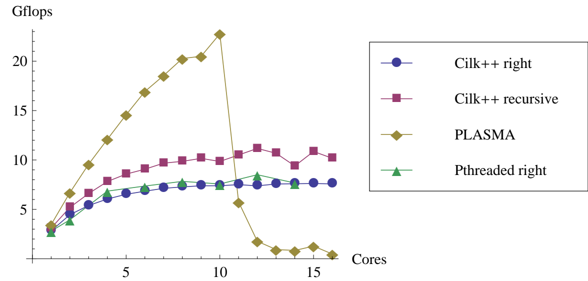


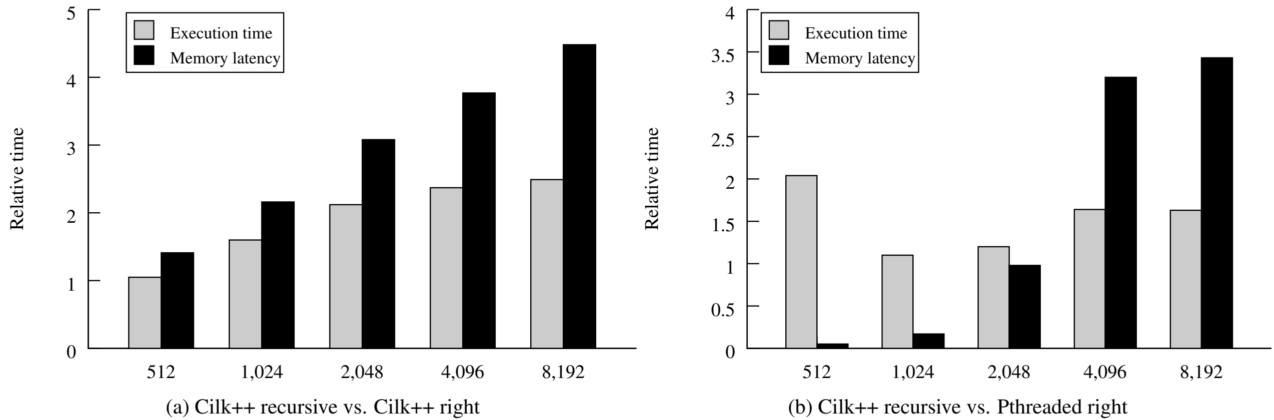
Figure 2: Performance impact from Xen kernels on Intel16.

DRAM than the Xen kernel has allocated, and Xen begins paging memory to disk.

Implementation	Maximum memory usage
Cilk++ recursive	1415 Mbytes
Cilk++ right	855 Mbytes
Pthreaded right	816 Mbytes
PLASMA	2158 Mbytes

Table 4: The maximum memory each LUP implementation uses to decompose a 8192×8192 matrix. The amount of memory required to hold a 8192×8192 matrix of double precision floats is 512 Mbytes.

We believe memory performance is the major limiting factor on performance for the right-looking implementations. Figure 3 shows performance of Cilk++ recursive compared to the right-looking implementations on



(a) Cilk++ recursive vs. Cilk++ right

(b) Cilk++ recursive vs. Pthreaded right

Figure 3: Execution time and memory latency of right-looking implementations relative to Cilk++ recursive. Each value is calculated by dividing the total for the right-looking implementation by the total for Cilk++ recursive. Value of 1 indicates equal values.

AMD16. This figure plots the relative load latency and execution time for different matrix sizes. Memory latency is counted using hardware performance counters as the total number of pipeline stalls waiting for memory divided by the number of loads. Notice how for large matrices, memory latency increases and execution time correspondingly increases. There are some constant overheads for small matrices that make this untrue in Figure 3b, but we believe this would correct itself for larger matrices. This figure does not prove a direct connection, but given other observations we believe it shows memory and caching to be the critical performance issue.

4.2 Cache Effects

We discovered while gathering results that performance for matrices whose row size was a power of 2 suffered significant performance degradation. This was due to conflict misses in the processor cache – elements of a single row would map to the same cache set, causing cache misses every time a worker accessed an element.

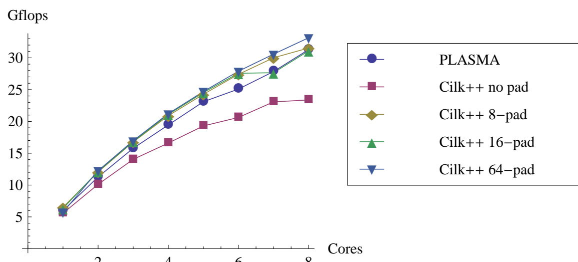


Figure 4: Performance of Cilk++ recursive compared to PLASMA with different padding sizes. Plot "i-pad" indicates that i padding elements were added to each row of the matrix.

We implemented a simple fix by padding the matrix with a single cache line per row, as needed. Figure 4 shows the results of this optimization. With no padding ("Cilk++ no-pad"), performance of Cilk++ recursive is significantly worse than PLASMA. With padding, performance of the two are indistinguishable, with Cilk++ recursive showing slightly better and less variable performance. Slight performance differences are observed for changing the padding value, but this is a second-order effect.

4.3 Parallelism

Table 5 shows parallelism values as reported by cilkview for both Cilk++ implementations. As the matrix size increases, so do the parallelism metrics for each implementation. Note, however, that for larger matrices Cilk++ recursive achieves much higher parallelism. Additionally, the burdened parallelism of Cilk++ recursive is much closer to its unburdened parallelism than Cilk++ right. This indicates that we should not expect the right-looking implementation to scale up to a large number of cores. However, we do not believe parallelism is the fundamental issue limiting the performance of right looking algorithms. With parallelism of 57.3 for 8192×8192 matrices, we would expect to see scaling up to at least four cores. However, even with low core count Cilk++ recursive scales much better, particularly on AMD16 and Intel16. This, along with the improved performance on Intel8Xen, makes us believe that memory architecture is a more important issue.

4.4 Scheduling

One of the major features of Cilk++ is its distributed, dynamic scheduler. Numeric algorithms such as LUP have well understood computation patterns, and static scheduling works well. Therefore, one would not expect LUP

Matrix Size	Cilk++ recursive		Cilk++ right	
	Parallelism	Burdened Parallelism	Parallelism	Burdened Parallelism
2048 × 2048	15.8	15.5	16.0	12.2
4096 × 4096	38.1	37.4	34.6	26.0
8192 × 8192	92.6	91.1	72.8	57.3

Table 5: Parallelism numbers given by cilkview for different matrix sizes.

decomposition to benefit much from Cilk++'s dynamic scheduler. In fact, one could expect performance degradation, as the dynamic scheduler moves computation away from the core that has cached the data.

There are, however, many plausible environments that reintroduce the need for dynamic scheduling. For example, a chip could have non-uniform frequency scaling due to temperature problems. Another example is when the system is running multiple programs that are contending on the CPU. This study explores the impact of a multiprogrammed environment on the performance of each algorithm.

Our methodology is to run multiple copies of each algorithm simultaneously, and see what impact this has on their performance. The "background job" is configured to take up different numbers of cores on the machine, going from no cores to the full chip. One interesting direction of future work would be to extend this with targeted workloads running as the background jobs to see which system factors are limiting performance.

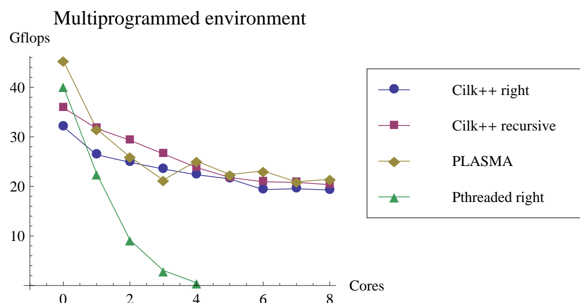


Figure 5: Performance of each implementation in a multiprogrammed environment. Performance is plotted on the y-axis in Gigaflops, while the x-axis plots the number of cores consumed by a background job.

Figure 5 shows the results on Intel8Xen. The Cilk++ implementations show modest, gradual performance degradation as more cores are consumed in the system. The degradation for PLASMA and pthreaded right are more severe. Pthreaded right shows extreme performance degradation – in fact, no points are plotted after 4 cores because tests would not finish in reasonable amounts of time. The degradation of PLASMA is much less pronounced, but is nevertheless a greater percentage of its original run-time. The performance of PLASMA and the Cilk++ implementations is roughly equivalent as more cores are contended.

Curiously, we observed greater performance degradation for PLASMA with smaller matrices (4096 × 4096), with Cilk++ implementations non-trivially outperforming PLASMA. This, along with its modest performance degradation in Figure 5, leads us to speculate that PLASMA may perform some coarse-grain dynamic scheduling itself. However, we do not understand the PLASMA code base sufficiently to definitely say whether or not this is the case.

5 Future work

Our evaluation indicates Cilk++ is a suitable language for writing high performance LUP decomposition applications. A possible direction for future work is implementing and evaluating Cilk++ applications similar to LUP decomposition, such as QR factorization.

Another interesting direction for future work is developing and evaluating matrix layouts for LUP decomposition. For example, a blocked matrix, where each block is contiguous array in memory, might provide better performance than a row major matrix with padding. The reason for this is that some hardware prefetchers, such as the unit-stride prefetch unit in AMD Opterons, are optimized for contiguous arrays. Using a matrix layout that is conducive to hardware prefetching would likely improve the memory bandwidth that the LUP application achieves. Our evaluation indicates that the performance of LUP applications is, at least in part, bottlenecked by memory bandwidth, so this could be an important optimization.

6 Conclusion

This paper presents the evaluation of four LUP decomposition implementations. We demonstrate that matrix layout is an important factor for achieving good performance, and that dynamic scheduling is necessary in a multiprogrammed environment. Furthermore, LUP decomposition written in Cilk++ competes with PLASMA, consumes less memory, and provides better performance in a multiprogrammed environment. These results suggest that Cilk++ is an appropriate language for writing high performance LUP decomposition applications and similar applications, such as QR decomposition.

References

[1] Concepts in multicore programming, lab 1a: Cilk++ warm-up, May 2010. http://stellar.mit.edu/S/course/6/sp10/6.884/homework/assignment1/.

[2] Gotoblas, May 2010. http://www.tacc.utexas.edu/tacc-projects/.

[3] Linear algebra package, May 2010. http://www.netlib.org/lapack/.

[4] The parallel linear algebra for scalable multicore architectures, May 2010. http://icl.cs.utk.edu/plasma/.

[5] B. C. Kuszmaul. Cilk provides the "best overall productivity" for high performance computing: (and won the hpc challenge award to prove it). In *SPAA*, pages 299–300, 2007.

[6] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.