Progress Report for Parallel LU factorization

June 27 – September 2, 2016

Stephen Wood

Ryan Glasby

Greg Peterson

Implementations of Chow and Patel’s PariLU algorithm from their 2015 paper, “Fine-grained Parallel Incomplete LU Factorization” are developed to maximize the performance on an Intel Xeon E5-2670 0 CPU. A basic implementation proved the algorithm to be stable and produced a baseline performance of 2 Gflops/s. The least efficient operations were identified by learning and utilizing Intel’s VTune and Vector Advisor. These operations were successively increased in their algorithmic intensity to achieve the best performance to date of 237 Gflops/s. To focus on the impacts of parallelization and vectorization strategies the algorithm has been applied to dense matrices as a precursor to sparse matrices. Convergence to a Gaussian Elimination factorization is utilized to assess the algorithmic impacts of implementation strategies and to separate this initial investigation from a particular solver utilizing the factorization.

Basic Implementation

Beginning with a basic implementation of Chow and Patel’s PariLU algorithm from their 2015 paper, “Fine-grained Parallel Incomplete LU Factorization”, I first confirmed the accuracy compared to a LU factorization obtained by Gaussian Elimination. I tested the convergence rate on 1 to 32 threads on an Intel Xeon E5-2670 0 CPU and found the asynchronous execution of the algorithm is stable and advantageous for more than 8 threads, see Figure 1. The performance of this basic implementation on the largest dense matrix (7680x7680) is less than 1% of the theoretical peak possible on the Xeon CPU, see Figure 2. Worryingly, the performance decreases as matric size increases for the basic implementation.



Figure : Convergence of basic implementation for a range of parallelism



Figure 2: Performance of basic implementation for a range of parallelism

DDOT Implementation

Examining the “hot spots” within the basic implementation execution using Intel’s VTune and the efficiency of the vectorization with Intel’s Vector Advisor I identified the slowest operations within the algorithm to be the dot product operations. Using Intel’s Math Kernel Library (MKL) double precision dot product (ddot) on vectors of L and U to replace the element by element operations increases the performance by ~20X for the largest matrix, see Figure 3. Convergence is not worsened by operating on vectors of L and U, see Figure 4.



Figure 3: Performance of MKL ddot() implementation for a range of parallelism



Figure 4: Convergence of MKL ddot() implementation for a range of parallelism

Tiled DDOT Implementation

The “hot spot” within the DDOT implementation had shifted to the updates of L and U. To better feed data to the performant ddot() operation I tested tiling the loop access through the matrix. I set the tile size based upon a rule of thumb, t <= sqrt(cache/2), for the L1 Cache (32 KB). The peak performance on the largest matrix increases by more than 10 Gflops/s, see Figure 5. Convergence is degraded by tiling access, see Figure 6. However, tiling access facilitates matrix-vector operations to update L and U which were implemented and examined next.



Figure 5: Performance of tiled ddot() implementation for a range of parallelism



Figure 6: Convergence of tiled ddot() implementation for a range of parallelism

Tiled DGEMV Implementation

The tiled access pattern enables a single column of U to read into memory and reused to update several entries in a column of L through a matrix-vector product. Performance of this implementation degrades relative to the tiled ddot() for the largest matrix size but improves for the 1920x1920 matrix, see Figure 7. Convergence improves by an average of two iterations relative to the tiled ddot(), see Figure 8.



Figure 7: Performance of Tiled MKL dgemv() implementation for a range of parallelism



Figure 8: Convergence of Tiled MKL dgemv() implementation for a range of parallelism

Tiled DGEMM Implementation

Storing the diagonal of the U matrix in a separate array and placing zeros on the diagonal enables matrix-matrix operations to be utilized to update entire tiles. The arithmetically intense matrix-matrix operation and tiled access pattern yields more than 50% of the peak performance, see Figure 9. Convergence remains stable, see Figure 10. Sweeping through tile sizes from 5 to 300 by 5 with 16 threads reveals that a tile size of 160 produces 237 Gflops/s, see Figure 11. Convergence remains stable across the range of tile sizes tested, see Figure 12.



Figure 9: Performance of Tiled MKL gemm() implementation for a range of parallelism



Figure 10: Convergence of Tiled MKL dgemm() implementation for a range of parallelism



Figure 11: Performance of the Tiled MKL dgemm() implementation a for range of tile sizes



Figure 12: Convergence of the Tiled MKL dgemm() implementation for a range of tile sizes

Remarks

A cache aware parallelization and vectorization strategy for the PariLU algorithm including the algorithmically intense matrix-matrix operation achieves 71% of peak performance on the Xeon CPU. Thread affinity settings are being tuned for increased performance. Rectangular Full Packed storage for dense triangular matrices remains to be examined.

Sparse and block sparse matrices common to engineering applications can often be permuted into banded matrices with small bandwidths. The implementation strategies examined so far appear promising for these engineering applications.