Final Project - LU with OpenMP Tasks **Yu Pei**
CS 594 Date: 05/03/2017

Abstract

LU is a widely used matrix factorization algorithm that factors a matrix as the product of a lower triangular matrix and an upper triangular matrix and in order to achieve numerical stability, we usually perform row pivoting. But to achieve good performance especially on multicore computers, blocking (as in LAPACK) or tiling (as in PLASMA) is needed to exploit cache hierarchy as well as the available cores. Since OpenMP 4.0, tasking with task dependency becomes the standard and can be used to build linear algebra algorithms. The asynchronous job scheduling scheme is a big improvement over the bulk synchronous parallel (BSP) paradigm and can obtain finer grain parallelism and better performance. Here we implemented LU factorization with OpenMP tasking and compared our implementation with MKL LAPACK and PLASMA implementation, experiments showed that we can achieve comparable performance.

Introduction

LU factorization can be viewed as the matrix form of Gaussian elimination to solve a large dense system of linear equations. And it is the core of many computations, for example, the inverse of a matrix, linear regression etc. Much progress have been made in order to speed up the LU factorization [1, 2] on a shared memory system. One of them, PLASMA, has been actively optimizing this routine on multi-core machines with tile data layout and dynamic job scheduling framework.

The difficulty of implementing LU when compared with, for example Cholesky factorization is that in order to maintain numerical stability, partial pivoting needs to be performed but that incurs a lot of data movement overhead. In addition, the panel factorization step poses as a major bottleneck as well [1].

In recent years, due to the prevalence of multi-core processors and heterogeneous architectures, the idea of superscalar scheduling is in the mainstream again, where the runtime schedules tasks by resolving data hazards in real time. Many runtime systems have been developed that utilize this technique, among them are OMPSs, StarPU and QUARK [4,5,6]. From OpenMP 4.0, the standard adopted the superscalar scheduling model and this provides a great opportunity for numerical linear algebra libraries that previously used proprietary scheduling system to adopt the standard instead. With dynamic scheduling, a higher degree of parallelism is possible thus obtaining better performance.

In this report, we developed the LU factorization program with OpenMP tasking based on tile data layout, similar to the current PLASMA library implementation. We document the issues we encountered in this process and finally we performed performance comparison with MKL LAPACK and PLASMA implementation. A few suggestions for future improvement are proposed in the end.

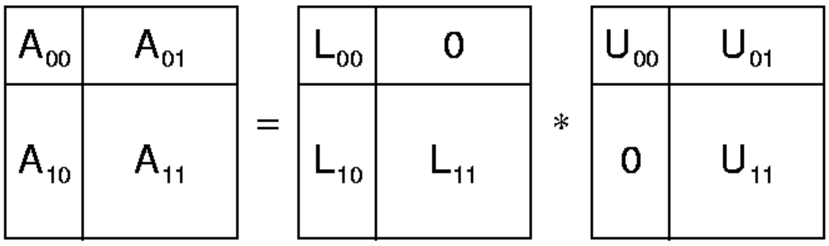
Background

LU Factorization

There are many different variants of the algorithm, depending on the sequence of the computation. Here we describe the block LU factorization and how can we solve it recursively. Suppose an $M*N$ matrix is partitioned as shown in the following figure, and we seek to factorize $A = LU$, then we can write:

$$L_{00}U_{00} = A_{00} \tag{1}$$
$$L_{10}U_{00} = A_{10} \tag{2}$$
$$L_{00}U_{01} = A_{01} \tag{3}$$
$$L_{10}U_{01} + L_{11}U_{11} = A_{11} \tag{4}$$

Figure 1: Block LU factorization of the partitioned matrix A



Where A_{00} is $r*r$, A_{01} is $r*(N-r)$, A_{10} is $(M-r)*r$ and A_{11} is $(M-r)*(N-r)$. L_{00} and L_{11} are lower triangular matrices with 1s on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

Equations 1 and 2 taken together perform an LU factorization on the first $M*r$ panel of A (i.e., A_{00} and A_{10}). Once this is completed, the matrices L_{00}, L_{10} and U_{00} are known, and the lower triangular system in Eq. 3 can be solved to give U_{01}. Finally, we rearrange Eq. 4 as,

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \tag{5}$$

From this equation we see that the problem of finding L_{11} and U_{11} reduces to finding the LU factorization of the $(M-r)*(N-r)$ matrix A'_{11}. This can be done by applying the steps outlined above to A'_{11} instead of to A. Repeating these steps until we obtain the LU factorization of the original $M*N$ matrix A. For an in-place algorithm, A is overwritten by L and U - the 1s on the diagonal of L do not need to be stored explicitly. Similarly, when A is updated by Eq. 5 this may also be done in place (copied from Netlib notes).

OpenMP

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) that supports multi-platform shared memory multiprocessing pro- gramming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

The first OpenMP specification, Fortran version 1.0, was published in Oc- tober 1997. C/C++ version 1.0 was published in October 1998. Fortran version 2.0 was published in November 2000, and C/C++ version 2.0 in March 2002. Version 2.5 combined Fortran and C/C++ interfaces and was published in May 2005. Version 3.0 introduced the concept of tasks, and was published in May 2008. The critical development, from the standpoint of this work, was the introduction of task superscalar scheduling in OpenMP 4.0, published in July 2013.

Equally important is the support for the standard in compilers. The GNU compiler suite has been on the forefront of adopting the standard. Support for OpenMP 2.5 was added in GCC 4.2 (May 2007), support for OpenMP 3.0 was added in GCC 4.4 (April 2009), and support for OpenMP 4.0 was added in GCC 4.9 (April 2014).

The **#pragma omp task depend** clause can be used to inform the compiler that the following code block is to be executed as a task (Figure 2), i.e., dynamically scheduled at runtime, and that it depends on earlier tasks that use the data referenced by the depend clause.

Figure 2: OpenMP #depend prgama example

```
#pragma omp task depend(in:A[0:nA],B[0:nB]) depend(inout:C[0:nC])
call_some_function( A, B, C )
```

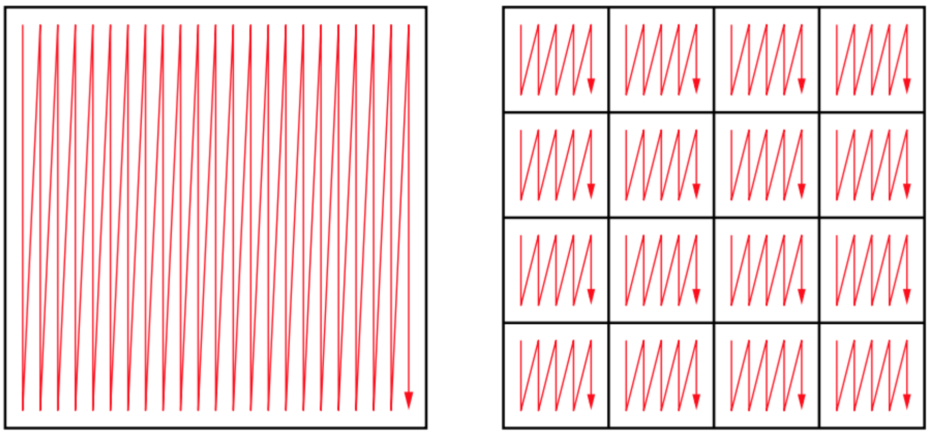
- The **in** dependence makes the task a descendant of any previously inserted task that lists the data items (i.e., A or B) in its **out** or **inout** dependency list.
- The **inout** or **out** dependence makes the task a descendant of any previously inserted task that lists the data item (i.e., C) in its **in**, **inout** or **out** dependency list.

The OpenMP runtime provides the capability to schedule the tasks at runtime, while avoiding data hazards by keeping track of dependencies. A task is not scheduled until its dependencies are fulfilled, and then, when a task completes, it fulfills dependencies for other tasks. This type of scheduling is more complex than the simpler scheduling of OpenMP 3.0 tasking, and the overhead associated with keeping track of progress may not be negligible[3].

Implementation

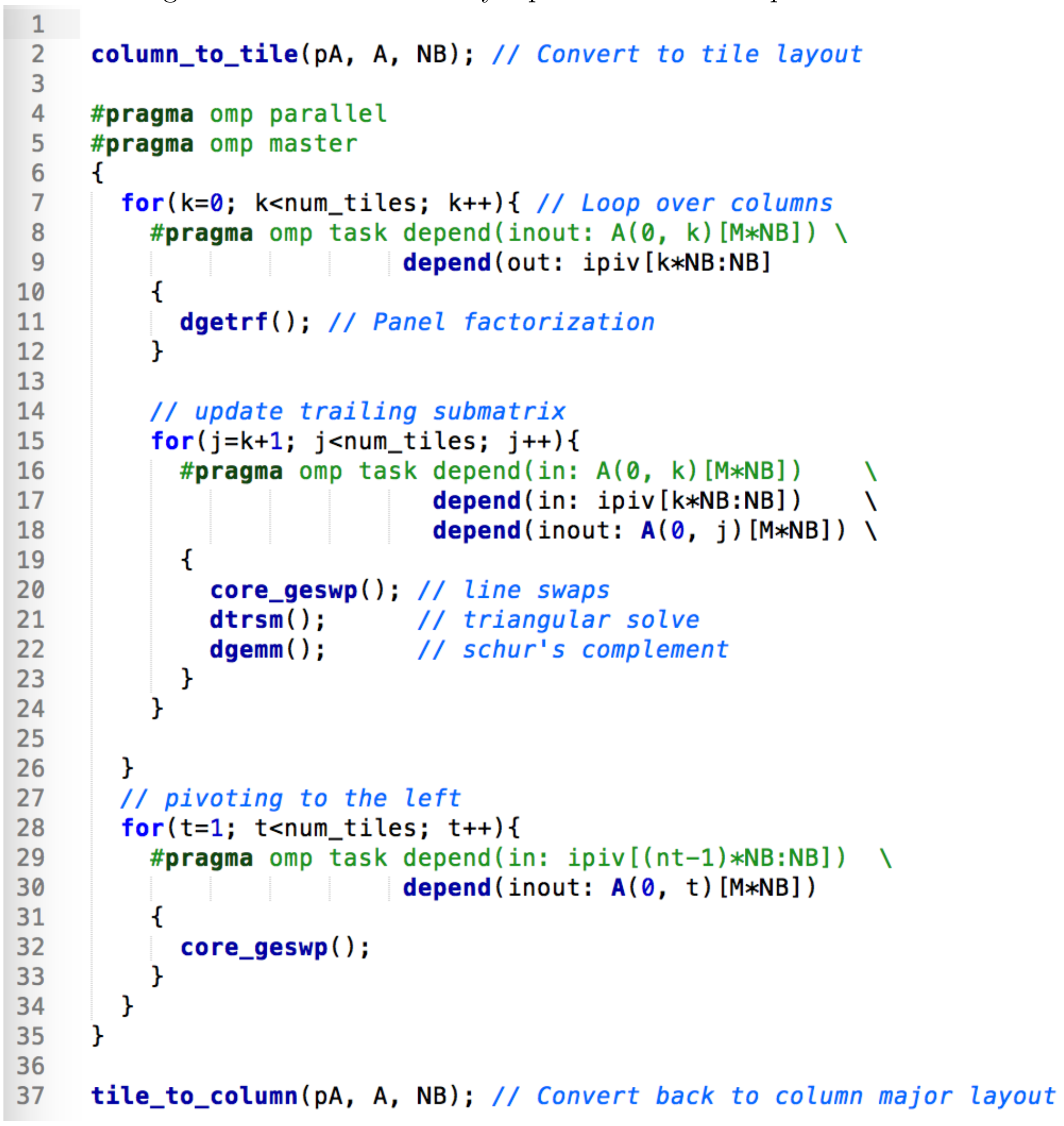
The main structure of the implementation follows closely the one in PLASMA, namely a right looking variant that operates on tile data layout. The difference between the canonical column major layout and the tile data layout is depicted in Figure 3. In the tile layout, each tile's data are in column major and the data within a tile are contiguous, and tiles are stored in column major layout.

Figure 3: Column major data layout and tile layout



The main framework is shown as the pseudo code in Figure 4. We have functions to convert between the two different data layouts (we assume user input is in column major layout). To create multiple threads and generate tasks, we enclose the main for loop within OpenMP parallel and master pragma. This way only the master thread will generate the tasks and multiple threads can pick up the tasks for execution.

Figure 4: Pseudo code of my OpenMP task LU implementation



```

column_to_tile(pA, A, NB); // Convert to tile layout

#pragma omp parallel
#pragma omp master
{
  for(k=0; k<num_tiles; k++){ // Loop over columns
    #pragma omp task depend(inout: A(0, k)[M*NB]) \
                     depend(out: ipiv[k*NB:NB]
    {
      dgetrf(); // Panel factorization
    }

    // update trailing submatrix
    for(j=k+1; j<num_tiles; j++){
      #pragma omp task depend(in: A(0, k)[M*NB])    \
                       depend(in: ipiv[k*NB:NB])    \
                       depend(inout: A(0, j)[M*NB]) \
      {
        core_geswp(); // line swaps
        dtrsm();      // triangular solve
        dgemm();      // schur's complement
      }
    }

  }
  // pivoting to the left
  for(t=1; t<num_tiles; t++){
    #pragma omp task depend(in: ipiv[(nt-1)*NB:NB])  \
                     depend(inout: A(0, t)[M*NB])
    {
      core_geswp();
    }
  }
}

tile_to_column(pA, A, NB); // Convert back to column major layout
```

The main loop iterates over all the columns, and the procedure follows the steps described in the background section: first we factorize the current panel (diagonal and all the tiles below it). Using notations from Figure 1, we will have L_{00} and L_{10} and we can solve equation equation 3 to obtain U_{01}. Since we perform row pivoting as well, we need to apply the row pivoting to all the trailing submatrix. Finally we perform matrix matrix multiply and update the subdiagonal matrix A_{11} and move to the next iteration. Because of the tile data layout, each column can be operated on independently and in parallel across threads.

After all the panels have been factorized, we need to apply the pivoting sequence to the left hand side of the matrix in the end and this can be completed for each column independently as well.

Based on the sequential workflow, we can derive the data dependency among tasks. Here we divided the work into three tasks: the panel factorization, the merged task that consist of line swaps, triangular solve and schur's complement update, and finally the task to pivot to the left. The trailing submatrix update can not be started until the current panel has been factorized, and the update also needs the pivoting sequence for line swaps. The trailing matrix update will take the panel as input and generate tasks for each remaining column, and the column next to the current panel will be very important, since the next panel factorization have to wait until this update have finished to start. Pivoting to the left in our implementation will need to wait for all the panels have been factorized and we have obtained the entire pivoting sequence. Only then, can each task do the line swap for a column independently from other columns.

There are a few difference in our implementation from the one in PLASMA in terms of dependency specification. OpenMP uses bracket with ':' to specify the offset from the pointer starting point, and the length of the memory region. Since the panel height is constantly decreasing, PLASMA specify the column in a finer grain fashion, here we specify the entire column instead, although we only use a portion of it. The benefit of this is that the pointer used in panel and in trailing update will be the same and is easy to guarantee correctness, since we are not sure how OpenMP use the length specified to manage dependency.

To avoid handling the corner cases, we assume the tile size can evenly divide the matrix size, and matrix row is greater than or equal to the number of columns. For panel factorization on tile data, we simply convert the tile layout back to column major layout, call LAPACK LU factorization then convert back to tile layout. Line swap routine is based on the BLAS routine.

Results and Discussion

Hardware/Software

The experiments were run on a machine node containing 2 sockets with 12 Intel Haswell threads on each socket (Xeon E5-2680 v3 2.60 GHz) for a total of 24 threads. The peak double precision performance of one node is 960 GFlop/s. ICC compiler 16.0.3 and the corresponding OpenMP and MKL math library were used for optimized BLAS operations. Timing is done via OpenMP timing function. We have validated the correctness of our result with LAPACK result.

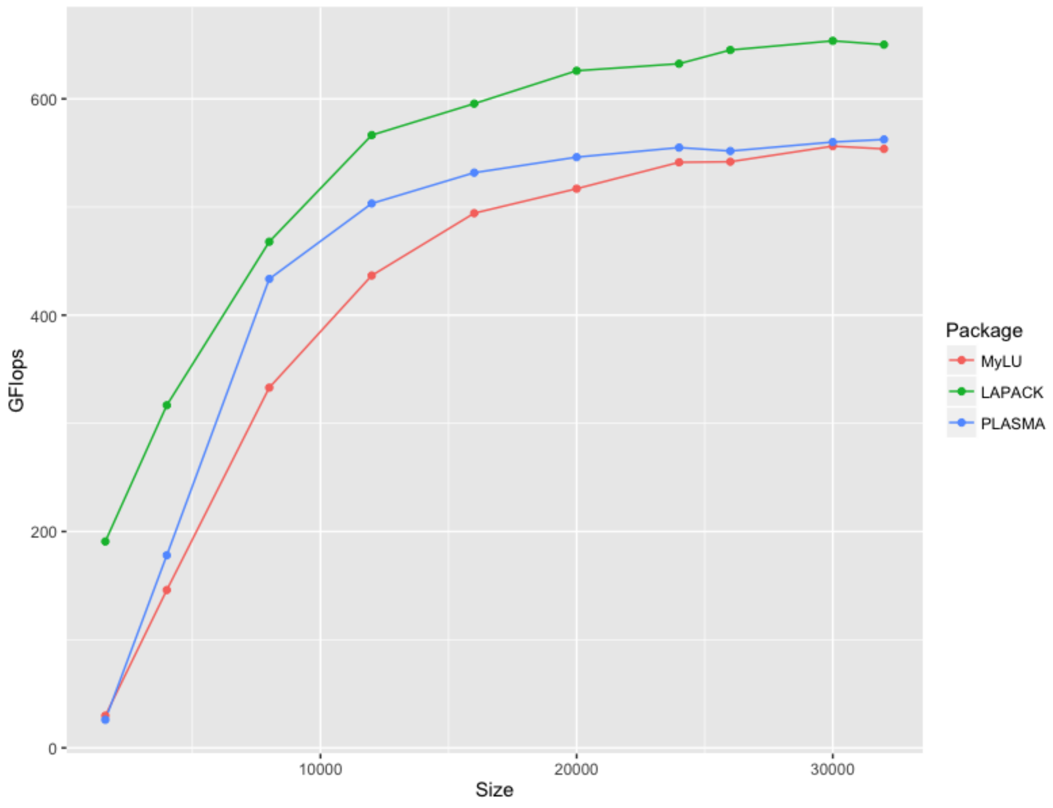
Tile size we used in the experiments is fixed to be 200 in our code as well as in PLASMA, and PLASMA panel block size is set to 40, with 5 parallel threads.

Performance with varying matrix sizes

The first measurement we did is to measure the maximum performance achievable with all 24 threads. GFlops is calculated based on $\frac{2*N^3}{3}$ total floating point operations. We varied the size of the matrix from 1600 to 32000 (square matrices with entries initialized to random numbers between 0 to 10). Both LAPACK and PLASMA measurements are for the whole procedure, **for my implementation, the timing excluded the translation time between the two layouts**.

We can see from Figure 5 that MKL LAPACK is always faster than both PLASMA and our implementation, which is consistent with previous result [3]. But PLASMA can saturate faster than our implementation, likely due to the faster panel factorization that can shorten the critical execution path.

Figure 5: Double precision LU with row pivoting performance for different matrix sizes



Performance with varying thread counts

Next we focus on the scalability of tasks LU regarding the number of thread used and the result is shown in Figure 6. We can see that our code (by extension, PLASMA code) can achieve good scalability and LAPACK can scale well as well.

Execution Traces

PLASMA library has capability to generate execution traces similar to the one shown in Figure 7 (Top trace is for LU with sequential panel, bottom one for parallel panel factorization). Unfortunately I didn't have enough time to get that to work on my program, and I didn't try some other tools that can work on OpenMP programs[7]. The information can provide guidance for our code optimization.

Figure 6: Double precision LU with row pivoting performance for different thread counts

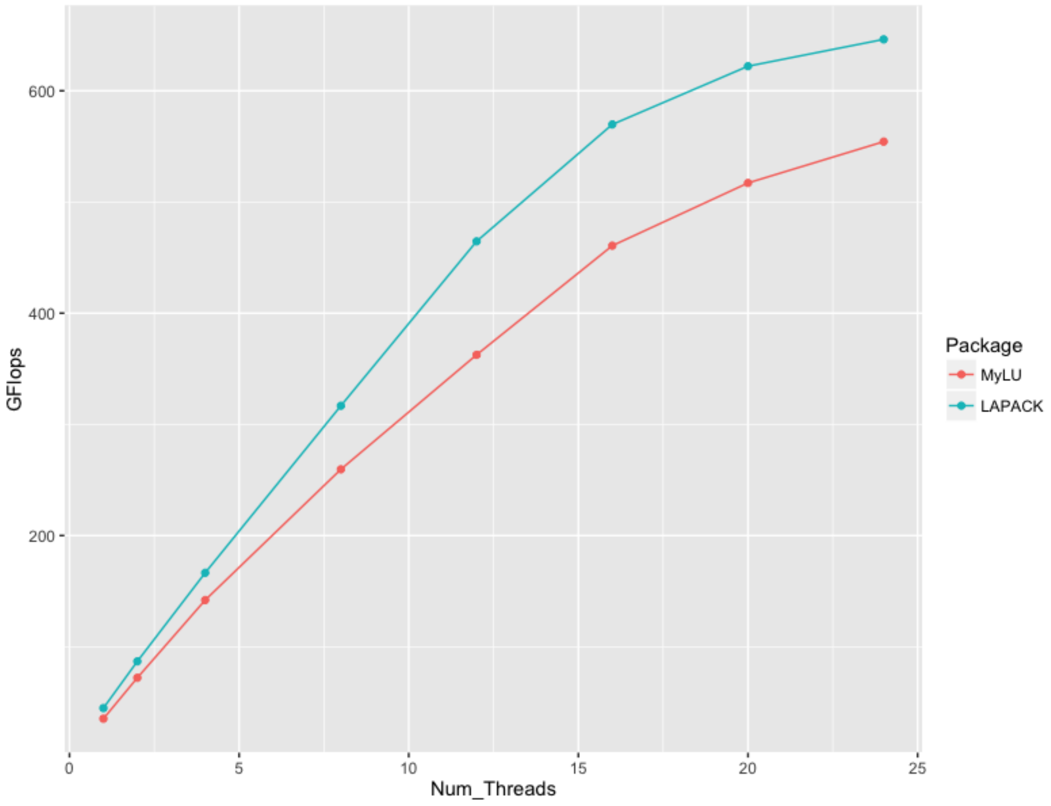
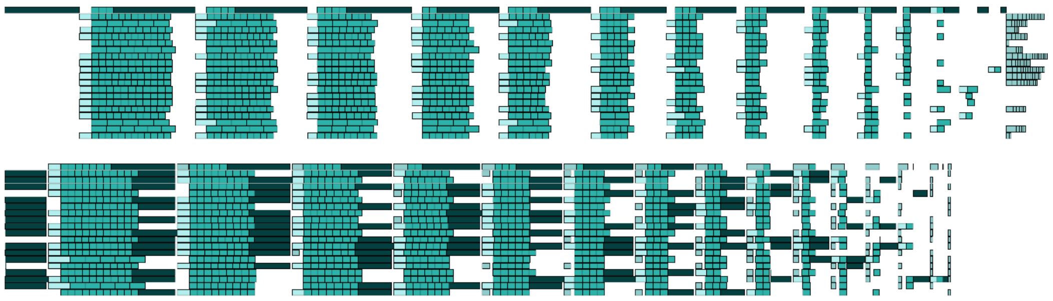


Figure 7: Example execution trace from a previous result



Conclusion and Future Work

In this project, we experienced with task scheduling, specifically the OpenMP task framework and used it to implement the classical LU matrix factorization very similar to the version in PLASMA library. There were some issues regarding dependency specification during the implementation and we solved the problem by expanding and unifying the pointers starting position. Comparable performance results are obtained for our implementation when compared with PLASMA, although still fall behind MKL LAPACK implementation.

For future work, there are several points that have been tested and can improve performance:

- Parallel panel factorization. Since we know that panel factorization is in the critical path and it is very important that we make that as fast as possible, introducing BLAS 3 operations and multiple threads will yield sizable speedup. PLASMA has the routine that implements this strategy but due to time limitation I did not implement it in my version[1].

- Speedup with accelerators. A large portion of the work lies in the matrix matrix multiplication and this can be significantly accelerated with GPU for example. Incorporating GPU can achieve much higher computing power but synchronization between host and device needs to be carefully managed[2].
- Assigning task priority. OpenMP provides a way to hint to the runtime the priority of the tasks. Giving tasks on the critical path should be able to improve the performance, and tracing graph can help in that aspect.

References

[1] Jack Dongarra, Mathieu Faverge, Hatem Ltaief and Piotr Luszcsek. *Achieving Numerical Accuracy and High Performance using Recursive Tile LU Factorization.* LAPACK Working Note 259.

[2] Jakub Kurzak, Piotr Luszczek, Mathieu Faverge and Jack Dongarra. *LU Factorization with Partial Pivoting for a Multi-CPU, Multi-GPU Shared Memory System.* LAPACK Working Note 266.

[3] Asim YarKhan, Jakub Kurzak, Piotr Luszczek and Jack Dongarra. *Porting the PLASMA Numerical Library to the OpenMP Standard.* Int J Parallel Prog (2017) 45: 612. doi:10.1007/s10766-016-0441-6.

[4] Duran, A., Ayguad, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J. *OMPSS: a proposal for programming heterogeneous multi-core architectures.* Parallel Process. Lett. 21(02),173193 (2011)

[5] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A. *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.* Concurr. Comput.: Pract. Exp. 23(2), 187198 (2011)

[6] YarKhan, A. *Dynamic Task Execution on Shared and Distributed Memory Architectures.* PhD thesis, University of Tennessee (2012)

[7] Drebes A., Brjon JB., Pop A., Heydemann K., Cohen A. *Language-Centric Performance Analysis of OpenMP Programs with Aftermath.* (2016) In: Maruyama N., de Supinski B., Wahib M. (eds) OpenMP: Memory, Devices, and Tasks. IWOMP 2016. Lecture Notes in Computer Science, vol 9903. Springer, Cham