Cholesky Decomposition

Amir Saadat[1] *CS 594 final report*

Abstract

Cholesky factorization is basically decomposition of a symmetric matrix into product of a triangular matrix L and its transpose ($A = L.\ L^T = U^T.U$). First an overview is given on different serial Cholesky methods which I have got familiar with, since all parallel algorithms is closely entangled to a serial Cholesky factorization. Row-wise and Column-wise Submatrix update method were chosen among serial methods to compare with LAPACK library *dpotrf_()* function. Then two methods are presented to perform Cholesky factorization in parallel; one in shared memory **OpenMP** system and the other in distributed memory message passing **MPI**. Based on arguments in reference papers, **Column Oriented Cholesky** was selected to focus on during parallelization of the algorithm using MKL_BLAS calls and data storage concept (Packed and Unpacked form). Also updating direction (Row-wise or Column-wise) was addressed and the results were compared with PLASMA in shared memory and scaLAPACK in distributed memory. In all cases performance and accuracy are considered and codes were precisely tested (and passed). Finally, I presented a discussion about the observed behaviors.

Introduction

Cholesky decomposition is a decomposition of a symmetric, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. It was discovered by André-Louis Cholesky for real matrices: $A=L.L^T$ where A is an nxn symmetric positive definite matrix

[1] PhD student- CBE department (asaadat@utk.edu)

and L is an nxn lower triangular matrix. When it is applicable, the Cholesky decomposition is a useful and fast method to solve linear system of equations. Linear system $Ax = b$ can then be solved by forward-substitution in system $Ly = b$, followed by back-substitution in upper triangular system $L^T x = y$ these substitutions will take about $O(n^2)$. Cholesky algorithm for nxn matrix perform about $O(n^3/3)$ flops therefore it's is roughly twice as efficient as the LU decomposition for solving systems of linear equations where $O(2n^3/3)$ flops should be performed. (Note that substitutions in order n^2 are negligible compare to n^3 for large matrices.)

Theory, Algorithms and Results (For Serial Cholesky)

There are various methods for calculating the Cholesky factorization:

1. *Row-Cholesky:*

In order to compute the ith row of the Cholesky factor, we require access to the previous *i-1* rows of L. Computationally, these i-1 rows are used to do a lower triangular solution to determine Lij, for $j=1,. \ . \ . \ ,i-1$. Then the diagonal element Lii can be obtained from these computed entries of the *ith* row. Depending on how the previous *i-1* rows are accessed, whether by row or by column, we have the *ijk* or the *ikj* forms of factorization. The *ith* task *Task_row(i)* depends on results from all the previous *i-1* tasks.

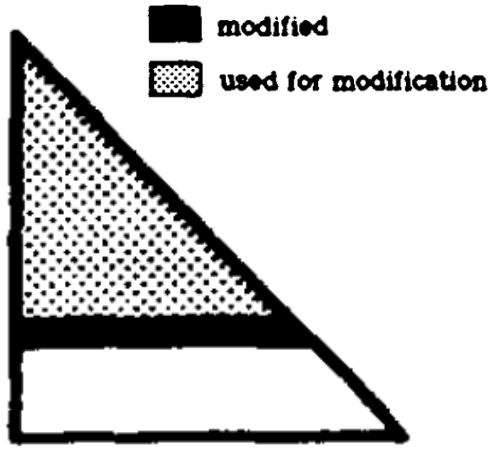

Figure 1. Schematic view of Row-Cholesky.

2. *Submatrix Cholesky:*

To apply the modification from column k of the Cholesky factor, we need to modify entries in the submatrix as given by the remaining *n-k* columns of the matrix. Figure 1 illustrates this this method.

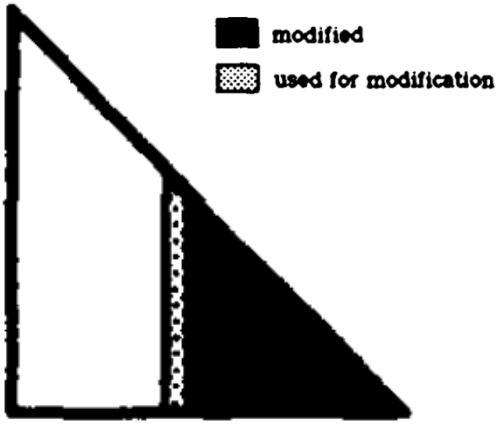

Figure 2. Schematic form of update in Submatrix update Cholesky.

This is the method that I used in my serial Cholesky investigation and comparison with LAPACK *dpotrf_()* function. I also introduced two types of data accessing, i.e. Row-wise and Column-wise. Here I will describe the method in detail for matrix of rank 4:

- For Column-wise Submatrix Cholesky we have:

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{21} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} * \begin{bmatrix} l_{11} & l_{21} & l_{31} & l_{41} \\ 0 & l_{22} & l_{32} & l_{42} \\ 0 & 0 & l_{33} & l_{43} \\ 0 & 0 & 0 & l_{44} \end{bmatrix} =$$

$$\begin{bmatrix} {l_{11}}^2 & l_{11}l_{21} & l_{11}l_{31} & l_{11}l_{41} \\ l_{11}l_{21} & {l_{21}}^2 + {l_{22}}^2 & l_{21}l_{31} + l_{22}l_{32} & l_{21}l_{41} + l_{22}l_{42} \\ l_{11}l_{31} & l_{21}l_{31} + l_{22}l_{32} & {l_{31}}^2 + {l_{32}}^2 + {l_{33}}^2 & l_{31}l_{41} + l_{32}l_{42} + l_{33}l_{43} \\ l_{11}l_{41} & l_{21}l_{41} + l_{22}l_{42} & l_{31}l_{41} + l_{32}l_{42} + l_{33}l_{43} & {l_{41}}^2 + {l_{42}}^2 + {l_{43}}^2 + {l_{44}}^2 \end{bmatrix} \quad (1)$$

Then first step is: $l_{11} = \sqrt{a_{11}}$

second step: $l_{j1} = {a_{j1}}/{l_{11}}$; $j = 1 + \{1,2,\ldots,n\}$

third step:

matrix constructed $l_{k1}l_{l1}$ where $\{ k = 1 + \{1,2,\ldots,n\} \,\&\, l = 1 + \{1,2,\ldots,n\}\}$:

$$\begin{bmatrix} {l_{21}}^2 & l_{21}l_{31} & l_{21}l_{41} \\ l_{21}l_{31} & {l_{31}}^2 & l_{31}l_{41} \\ l_{21}l_{41} & l_{31}l_{41} & {l_{41}}^2 \end{bmatrix}$$

subtracting from the Minor matrix of A:

$$\begin{bmatrix} a_{22} - {l_{21}}^2 & a_{32} - l_{21}l_{31} & a_{42} - l_{21}l_{41} \\ a_{32} - l_{21}l_{31} & a_{33} - {l_{31}}^2 & a_{43} - l_{31}l_{41} \\ a_{42} - l_{21}l_{41} & a_{43} - l_{31}l_{41} & a_{44} - {l_{41}}^2 \end{bmatrix}$$

Next step: is to continue doing the first three steps for $j = 2,\ldots,n$

Notes:

- ✓ Factor L is computed in place, overwriting lower triangle of A
- ✓ Pivoting is not required for numerical stability
- ✓ About $n^3/6$ multiplications and similar number of additions are required (about half as many as for LU)

- For Row-wise Submatrix Cholesky we have similar method but a_{ij} would be a_{ji}.
- Block Algorithm (for comparison with two former methods)

This is the method used by *dpotrf_()*. The bulk of the computation in the above algorithm is in the symmetric rank-1 update, which performs $o(n^2)$ operations on $O(n^2)$ data. It is this ratio of computation to data volume (requiring memory accesses) that stands in the way of high performance. To overcome this, we can reformulate the algorithm in terms of matrix-matrix multiplication (or rank-k updates) which have a more favorable computation to data volume ratio, allowing for more effective use of a microprocessor's cache memory. To increase

performance and better use of cache coherency, it's possible to devide matrix into blocks (*bxb* elements) and instead of single column, update a rectangular *(n-b)xb* matrix.

Results:

The program has been run in hydra machine. Codes and Makefile are stored in "Serial" subfolder. One thing to note is that the matrices were completely nonzero and random in order not to misinterpret the Performance data.

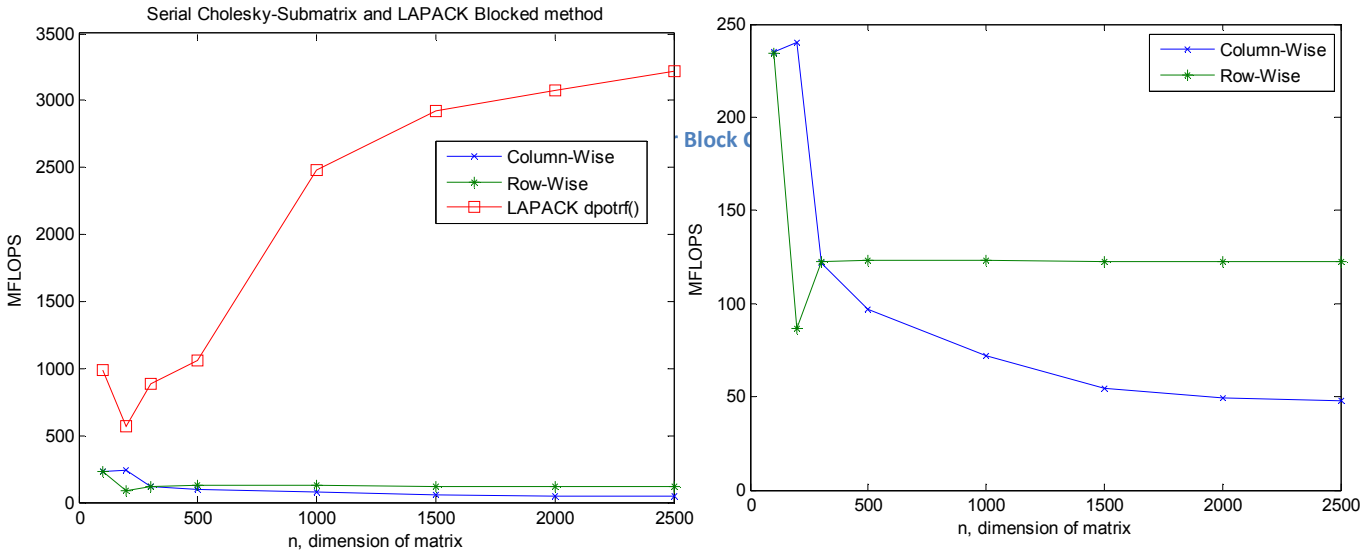

Figure 3. Serial Submatrix Update Cholesky method and LAPACK blocked dpotrf().

As it was expected, LAPACK is lot more efficient due to its Cache reuse and Blocked calculation method. Also, Row-wise Submatrix is better than Column-wise as I am coding in C++.

3. *Column Cholesky:*

To compute column j of the Cholesky factor, we require access to the rectangular submatrix enclosed (inclusively) by the *jth* row and *jth* column of *L*. This is the basic method I used for parallelizing Cholesky factorization. This method will be explained in more detail in parallel section.

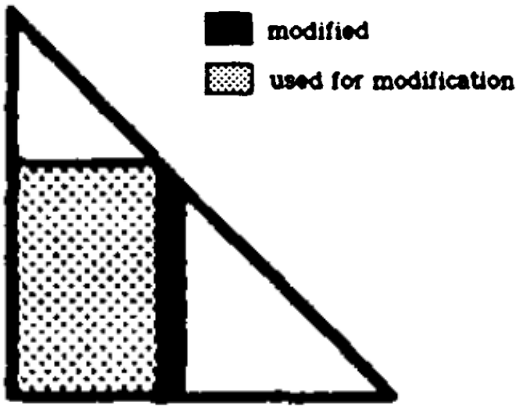

Figure 4. Schematic representation of Column Cholesky method.

Column Cholesky was introduced in Ref[1][2] as more potentially efficient Algorithm because of better load balance of Tasks. As it is understood from the task work distribution in Fig.5, In Col-

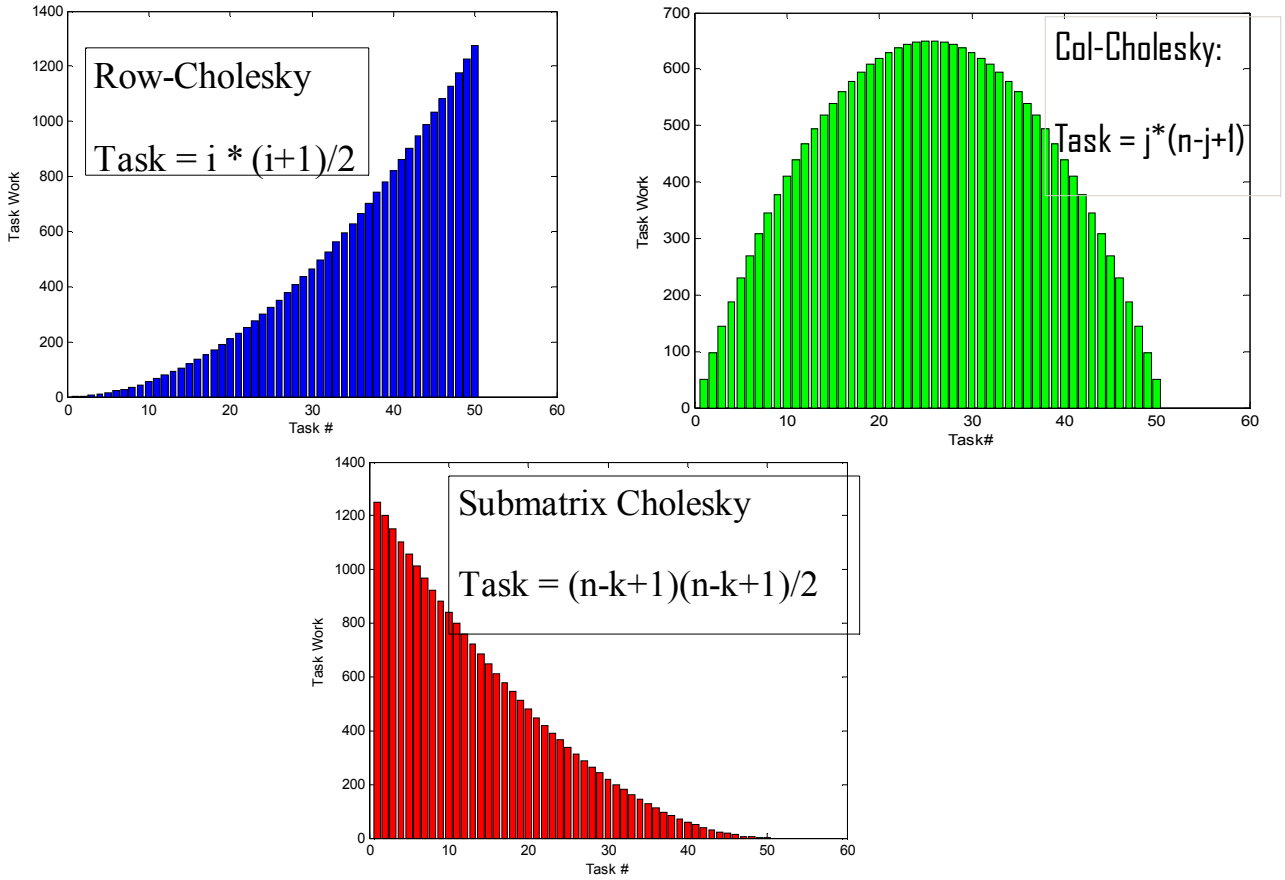

Figure 5. Task work distribution in different Cholesky methods.

[2] Ref[1]. A. George, M.T. Heath and J. Liu, Parallel Cholesky Factorization on Shared-Memory Multiprocessor, Linear Algebra and its Applications 77:165-187, 1986.

Cholesky, the distribution of work is better in the sense that processors from start to end are more likely to have something to do. Obviously, the integration of Task works is the same for all three.

- Array Storage method[3]:

As far as A is a symmetric matrix one can use only members below/above plus members on diagonal. This will be helpful when handling a huge matrix and also in high ranks it should be helpful to increase performance by increasing cache locality and not bringing useless elements of the other half to the cache each time one element is called from memory. There are two major ways to store a symmetric matrix in packed format, as shown above; namely first one Row-Packed and second one Column-Packed).

$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ a_{10}\ a_{20}\ *\ a_{11}\ a_{21}\ *\ *\ a_{22}$
$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ *\ *\ a_{10}\ a_{11}\ *\ a_{20}\ a_{21}\ a_{22}$

Figure 4. Data storage in Packed forms.

I have used the first alternative in doing Column Cholesky. As it is explained in Ref[2], one can easily do this convert indices as:

$$a_{i,j} \quad \rightarrow \quad a_{i,j*(2n-j-1)/2} \qquad (3)$$

Apparently, there are two advantages and one disadvantage in using this method.

Advantages:

- ✓ Changing the Accessing pattern from column major to row major.
- ✓ Decreasing the amount of cache misses and increasing cache reuse.

Disadvantage:

- Increase of floating point operations.

[3] Ref[2]. http://www.netlib.org/blas/blast-forum/chapter2.pdf

Theory, Algorithms and Results (For Parallel Cholesky with OpenMP)

For this section, I have used Ref[1] implemented an **OpenMP** based parallel Cholesky factorization. The algorithm here is Column Oriented Cholesky (As stated above). Particularly, each processor picks up a task *Task_col(j), 1 ≤ j ≤ n,* from a global task-queue, where the tasks are ordered on the basis of increasing column numbers. Thus, *Task_col(1)* appears before task *Task_col(2)* in the task-queue, which in turn appears before *Task_col(3),* and so on. Thus, the last task in the queue, and thus the last task to be performed, is *Task_col(n),* where n is the order of the matrix. The order of tasks in the queue is important and at the end of completion of *Task_col(j),* column *j* is the *j*th column for the Cholesky factor *L* of the original matrix. It could be presented as:

```
Cholesky_Parallel_OMP( ) {
  for j = 1 to n
  begin
    Pick up task Task_col(j) from task queue
  end
                              }
```

Figure 7. pseudo code for parallel OMP Cholesky.

Moreover, each task Task_col(j) is composed of a number of column modification operations. These operations are of two types:

1. A column j is modified by using data from all the preceding columns, where *k = 1, 2, ..., j-1*. For a given value of *k*, this can be denoted by *Col_mod(j,k)* and its pseudo-code isin Fig. 8.

The subtraction operation in *Col_mod()* is done by *cblas_daxpy()* function.

2. First, the square root of diagonal element is taken, then the elements of column j are divided by the square-root of the diagonal element on the same column. This can be denoted by *Col_div(j).* The division operation is done by *cblas_dscal().*

```
Col_mod( j, k ) {
        for i = j to n
         begin
                  A [i, j] = A [i, j] – A [j, k] * A [i, k]
        end
                  }
```

Figure 8. pseudo code for column modification function.

There is a fixed order for these operations. A *Col_div()* operation can only be carried out on column *j* only after the column *j* elements have been modified by data from all the preceding columns using *Col_mod()* operations. Moreover, a *Col_mod()* operation on a particular column can use a preceding column only when the latter is ready, that is after the *Col_mod* and *Col_div* operations on that column have already been performed. To indicate the status of the particular column, that is to indicate whether it can be used in *Col_mod* operations on succeeding columns, [] mentions the use of an array, ***ready[].*** This data structure has been used in my program too and plays an important role here.

A pseudo-code for the task *Task_col(j),* in terms of the *Col_mod* and *Col_div* operations:

```
Task_col( j ) {
        for k = 1 to j-1
        begin
                Wait until the ready[k] flag has been set
                Perform Col_mod(j,k) operation
        end
        Perform Col_div(j) operation
        Set the ready[j] flag
}
```

Figure 9. pseudo code for Task_col(j) function.

As seen from the above pseudo-code, in each task Tcol(j) a number of cmod operations are performed on a column at the end of which a cdiv operation is executed. The scheduling is better illustrated in Fig.1a. This figure illustrates how each task is further divided into the above defined *Col_mod* and *Col_div* subtasks, taking the specific example of a matrix of order $n = 5$. From the ordering shown in the figure, one can observe the potential parallelism in this algorithm. Consider a scenario for two processors, P1 and P2 , where each processor handles each time a single task *Task_col(j)*. Without loss of generality, we can assume that P1 starts working on *Task_col(1)* which contains only a single sub-task *Col_div(1).* During this time, processor P2 , which is currently idle, will pick up *Task_col(2)* from the queue. *Task_col(2)* consists of sub-tasks *Col_mod(2,1)* and *Col_div(2),* which need to be carried out in this order. For sub-task *Col_mod(2,1)* to be performed, column 1 needs to be ready first. Hence, P2 will be idle while P1 finishes its work on column 1. Once this

is done, P2 will resume execution. In the meantime, P1 can now pickup *Task_col(3)* from the queue. While P2 is still working on column 2, P1 can at least complete subtask *Col_mod(3,1)* as column 1 is complete. Once column 2 is ready, P1 can perform the rest of the task, *Task_col(3)*. During this time P2 , which is now idle will pick up *Task_col(4)* from the queue and, until *Task_col(3)* is fully completed, will perform sub-tasks *Col_mod(4,1)* and *Col_mod(4,2)*. This will continue until no more tasks remain to be performed, which in this case happens once P1 picks up the remaining task *Task_col(5),* as shown in Fig.1b. The authors of the original paper Ref[1] name this "self-scheduling".

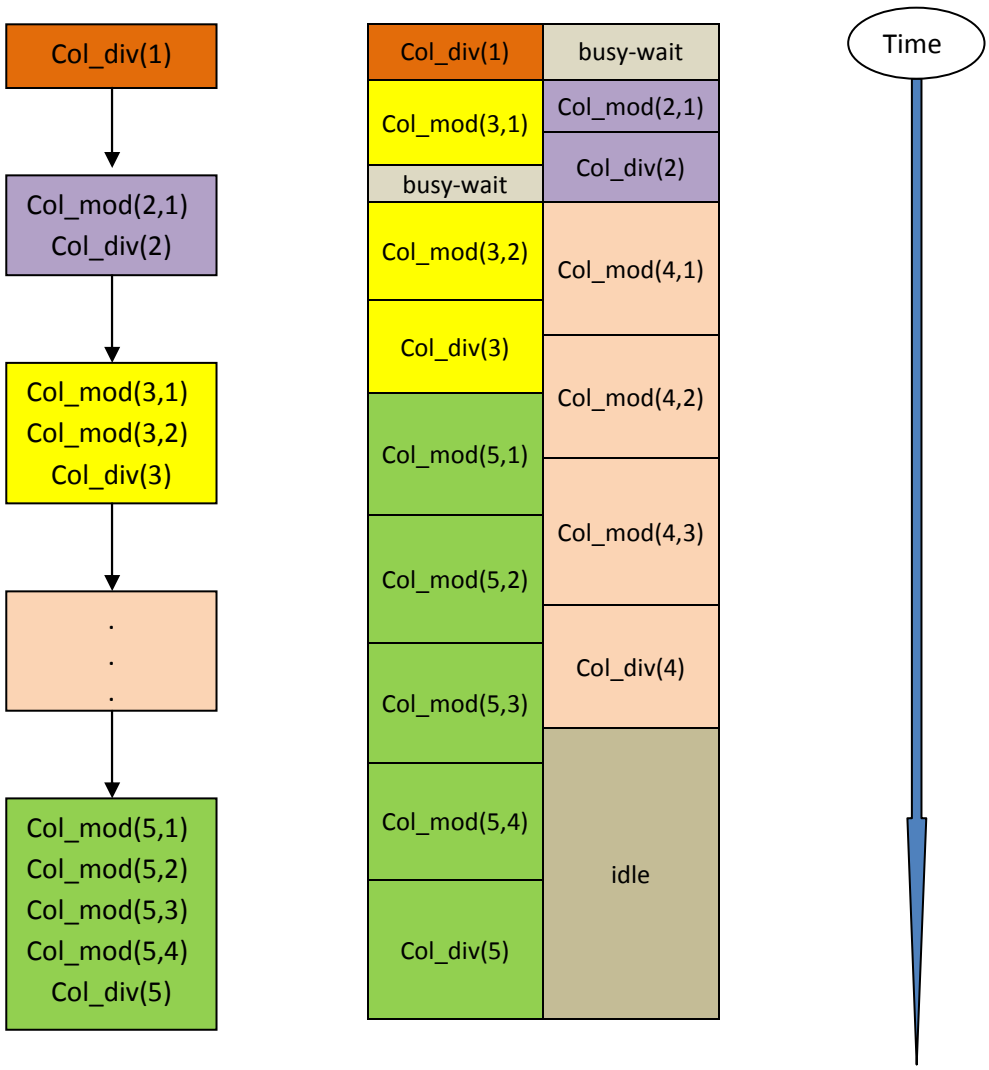

Figure 10. Schematic representation of parallel Column Cholesky for a matrix of rank 5.

The above algorithm has applied to both packed and unpacked states on group2 cluster.

Results:

Table 1. runtime and speedup for sequential and parallel Unpacked Stored Chlesky.

Order of matrix (n)	Sequential run-time (sec)	P = 2		p = 4		p = 8	
		Parallel run-time (sec)	Speedup	Parallel run-time (sec)	Speedup	Parallel run-time (sec)	Speedup
500	0.066575	0.573751	0.11	0.253307	0.26	0.180856	0.39
1500	2.741704	17.194054	0.15	5.325873	0.52	5.500979	0.5
2000	6.591150	15.413304	0.42	11.252013	0.58	17.690219	0.37
2500	14.660367	63.760863	0.23	22.191439	0.66	19.195476	0.76
3000	26.775183	44.543718	0.60	55.401160	0.48	29.904158	0.90
3500	40.449806	165.01936	0.24	53.391307	0.75	43.333751	0.93
5000	152.306137	345.0206	0.44	222.24456	0.68	121.26268	1.25

Table 2. runtime and speedup for sequential and parallel Packed Stored Cholesky.

Order of matrix (n)	Sequential run-time (sec)	P = 2		p = 4		p = 8		p = 8, PLASMA
		Parallel run-time (sec)	Speedup	Parallel run-time (sec)	Speedup	Parallel run-time (sec)	Speedup	Speedup
500	0.025132	0.072621	0.35	0.011709	2.14	0.005930	4.23	1.33
1500	0.504532	0.344995	1.46	0.168797	2.98	0.114359	4.41	5.01
2000	1.492682	0.885646	1.68	0.412234	3.62	0.277910	5.37	5.37
2500	3.784008	2.047774	1.85	1.042373	3.63	0.687795	5.50	6.09
3000	6.834918	3.703445	1.85	2.033217	3.36	1.268673	5.38	6.01
3500	8.773805	5.767410	1.52	3.036204	2.88	2.072095	4.23	6.21
5000	34.096489	16.31397	2.00	8.589718	3.96	5.906355	5.77	7.23

It is concluded from these data that packed storage has much better efficiency than unpacked in both sequential and parallel. The execution time increases by parallelizing unpacked, however, **for packed storage**, there is a significant improvement in execution time by parallelization.

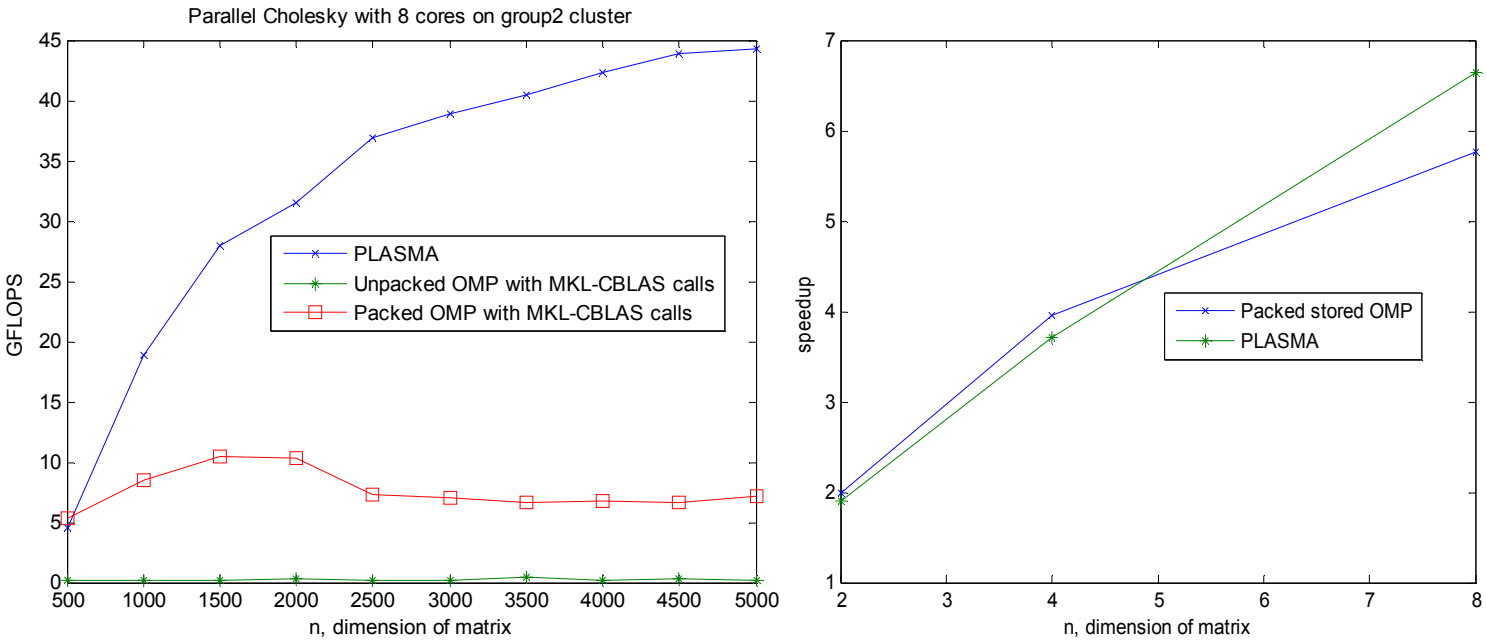

Figure 11. GFLOPS and Scalability (in n=5000) for PLASMA, packed and unpacked parallel Cholesky with OMP.

Theory, Algorithms and Results (For Parallel Cholesky with MPI)

The basic algorithm of this section is very similar to OMP section. The key features here is the fact that I used **1D Block Cyclic data distribution.** Then similar to OMP section, there should be a correct order of modification for columns. Each column should be modified based on those previous columns which have been modified. So the column in each process will need some modified columns of the same process and the other processes. Therefore, after each modification, the modified column is broadcast to all other processes (By non-blocking sends).

Moreover, during the modification procedure, the required previously modified columns would be received first and then used in modification. The ready flag here also plays an important role.

Things that I used to improve the performance:

- ✓ I stored the cyclic columns in Row-Major format.
- ✓ The modifications were only applied on necessary elements and not the whole column.
- ✓ Passing argument to function was managed to be done only on elements which should be changed or modified.

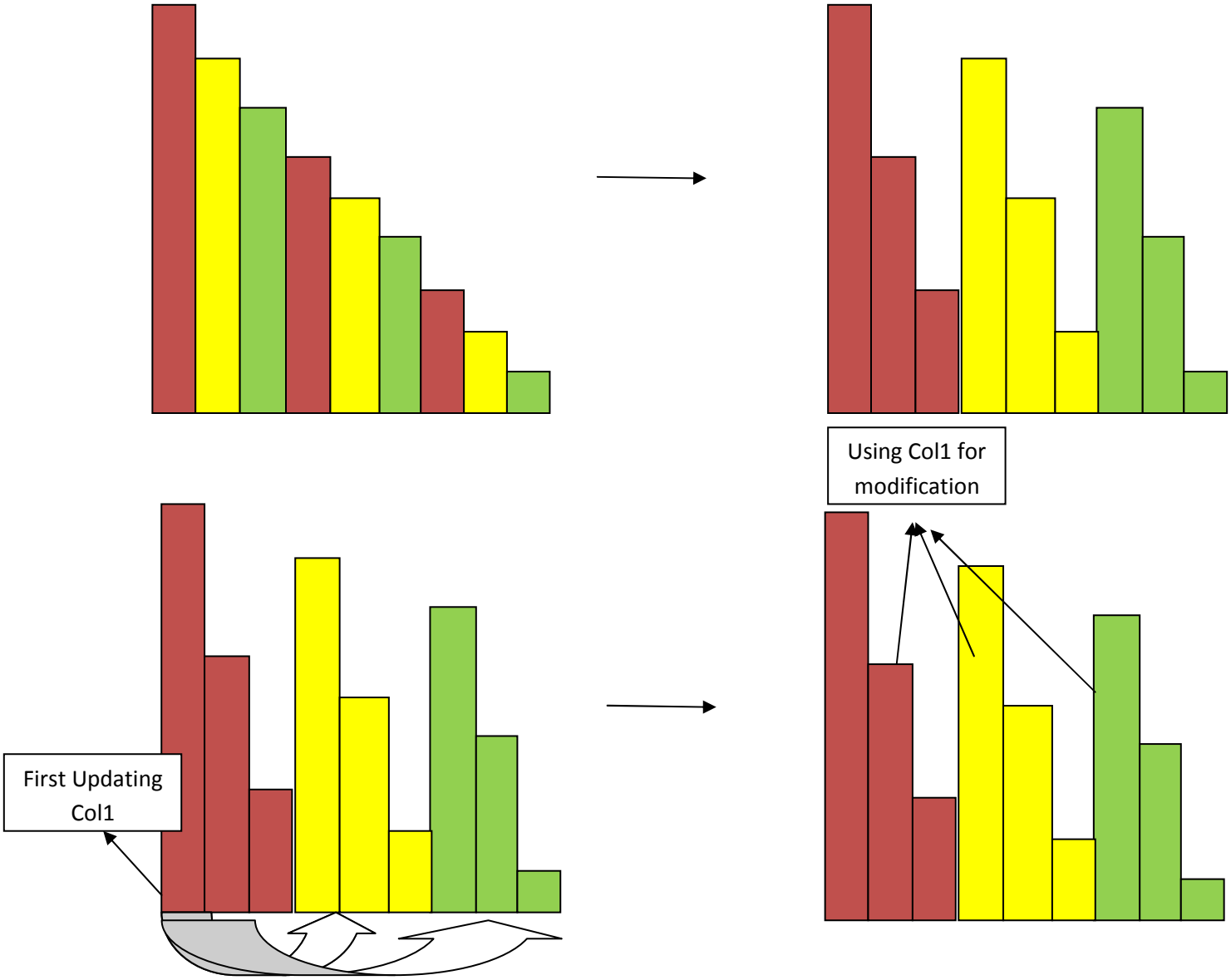

Figure 12. MPI parallel procedures.

Results:

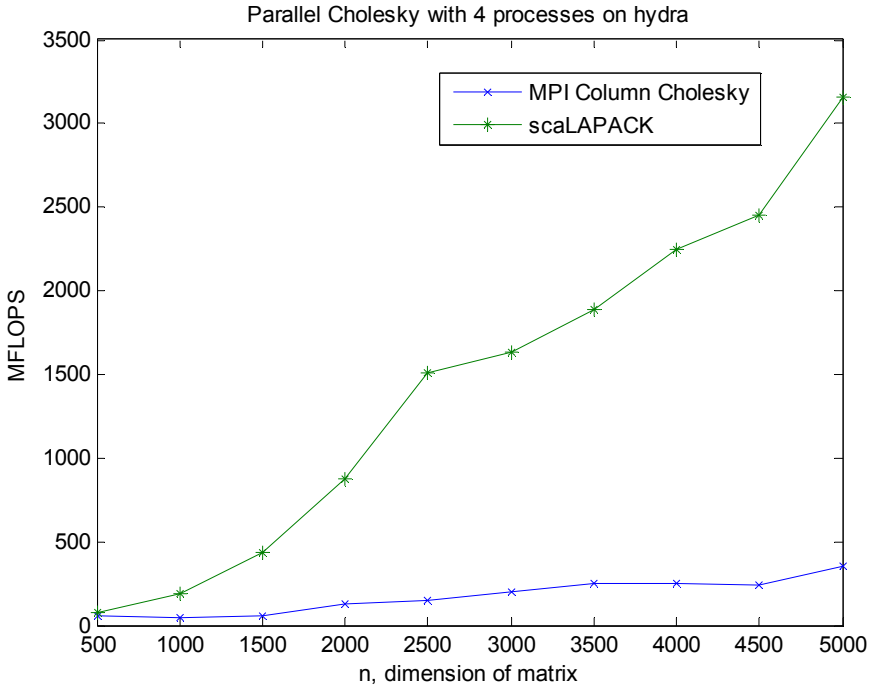

Figure 5. MFLOPS of Column Cholesky and scaLAPACK.

As expected scaLAPACK has much better efficiency and better speed up.

Table 3. execution time and speed up for MPI implementation and its comparison with scaLAPACK. (Speed up is compared to 4 processes)

Order of matrix (n)	P = 4; scaLAPACK	P = 16; scaLAPACK		p = 4; my MPI	p = 16; my MPI	
	Parallel run-time (sec)	Parallel run-time (sec)	Speedup	Parallel run-time (sec)	Parallel run-time (sec)	Speedup
500	0.579992	0.078547	7.38	0.720089	0.079353	9.07
1500	2.567762	0.392275	6.54	21.779398	3.608736	6.03
2000	3.032151	0.703602	4.3	20.221934	9.057050	2.23
2500	3.419241	1.238325	2.76	34.831245	31.785980	1.09
3000	5.510543	1.799746	3.06	46.172328	38.432590	1.20
3500	7.569723	2.540786	2.97	57.171893	86.442437	0.66
5000	13.184330	5.194248	2.54	118.392482	253.634777	0.46