Dense Linear Algebra with OpenMP and MPI

ARHAM
BENGANI

KTH Industrial Engineering
and Management

DD2356 Methods in High Performance Computing Report
Stockholm, Sweden 2017

Gaussian Elimination - A short summary

Gaussian Elimination is an algorithm in linear algebra to solve linear equations. The gaussian elimination consists of two phases, the forward elimination and the backward substitution. The forward elimination consists of (n - 1) steps where n is the number of rows of the matrix A in the linear system that is defined by the equation Ax = b, where x is the unknown solution vector of size n and b is the vector of size n. After the forward elimination we will have the equation Ax = b transformed into the equation Ux = b' where U is the (n * n) matrix in upper triangular form. This transformation is carried out in n - 1 steps where in each step all the elements below the diagonal element in the respective column based on the number of the step is converted to 0. The conversion takes place by first dividing the corresponding element in the diagonal with the element at the position that needs to be converted to 0 and then multiplying the row consisting of the diagonal element with the result of the division. Thus now if we subtract the two rows i.e. the row containing the diagonal element after multiplication from the row containing the element that need to be converted to 0, we get the element to be 0 and a new corresponding values in the row. At the same time, we also perform the similar operation in the corresponding rows in the b vector. Thus after the 1st step all the elements below the diagonal element in the first column are converted to 0, after the second step all the elements below the diagonal element in the second column are converted to 0 and so on. Thus at the end of n - 1 steps we will have a matrix in the upper triangular form and also vector b with new set of values. This will be the end of the first phase of gaussian elimination. Now we need to perform the second phase i.e. the backward substitution to calculate the unknown solution vector x of the equation. We do this in the backward order i.e. the loop will run from n to 1. If we look at the form of the matrix U in the equation Ux = b' that we obtained after the completion of the forward elimination, we will see that upon solving the last row of the upper triangular form matrix and the last row of the vector b' for the unknown vector x, we can obtain the value of x[n] = b'[n] / U[n,n] [1]. Thus we have x[n] which can now be used in the matrix vector multiplication equation the previous row to solve the equation and obtain the value of x[n - 1]. Thus in this way by substituting the value computed in the subsequent row in the previous row we can iterate and solve for the value of x vector. Thus at the end of the second phase named backward elimination we will have the value or solution of the unknown solution vector. In this way over the course of these two steps we can solve the set of given linear equation.
We then implement the gaussian elimination by the pivot row method. A pivot element is needed when $a_{kk} = 0$ and when a_{kk} is very small which would induce an elimination factor, which is very large leading to imprecise computations. Pivoting strategies are used to find an appropriate pivot element. Typical strategies are column pivoting, row pivoting, and total pivoting [1].
In this method the basic structure and the steps involved in the process are same with that of the normal elimination explained above. The main difference is the determination of the pivot row. The additional steps include the modification in the forward elimination method. In the forward elimination method, we first calculate the maximum absolute value in the concerned column. The row with the maximum absolute value is the pivot row. Then we replace the row that consists the diagonal element with the pivot row and carry out the following steps of the forward elimination.

Comparison of row-cyclic and total cyclic data distribution

The following tables i.e. table 1 And 2 show the time measurement of the algorithms implemented in the step 2 and step 3 of the project assignment. The table under the **STEP 2** i.e. table 1 is the time measurement for the row-cyclic data distribution and the table under the **STEP 3** i.e. table 2 is the time measurement for the total cyclic data distribution. All time measurement is done in the unit millisecond (ms). The algorithm was run for different number of processors i.e. 4, 8 and 16 and different number of data block size i.e. 8, 16, 64, 256, 512 and 1024. Every combination was run for 5 times on one node of the cluster tegner at PDC Center for High Performance Computing. The average was then calculated and presented in the table at the end of the 5 computations for each combination.

STEP 2

Table 1.

No. of processor 4					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000315	0.000452	0.001283	0.016311	0.083334	0.401941
0.000305	0.000434	0.001322	0.015031	0.084885	0.400551
0.000315	0.00043	0.001195	0.022709	0.084648	0.404519
0.000351	0.000447	0.001279	0.01506	0.085556	0.401951
0.000308	0.000427	0.001388	0.015426	0.081147	0.407833
0.0003188	0.000438	0.0012934	0.0169074	0.083914	0.403359
No. of processor 8					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000466	0.000532	0.001581	0.012417	0.054516	0.227116
0.00039	0.00059	0.001428	0.011044	0.056478	0.227475
0.000485	0.000577	0.001669	0.013688	0.048389	0.226083
0.000494	0.000531	0.001479	0.013541	0.070282	0.222856
0.000462	0.000498	0.001643	0.00998	0.05587	0.220935
0.0004594	0.0005456	0.00156	0.012134	0.057107	0.224893
No. of processor 16					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.0005	0.000781	0.002221	0.010038	0.044659	0.136615
0.000524	0.000818	0.00231	0.010711	0.039814	0.137442
0.00057	0.00077	0.002535	0.011057	0.028875	0.137421
0.000628	0.000823	0.002458	0.011544	0.031107	0.137236
0.000571	0.000831	0.002185	0.010337	0.041202	0.137708
0.0005586	0.0008046	0.0023418	0.0107374	0.0371314	0.1372844

STEP 3

Table 2.

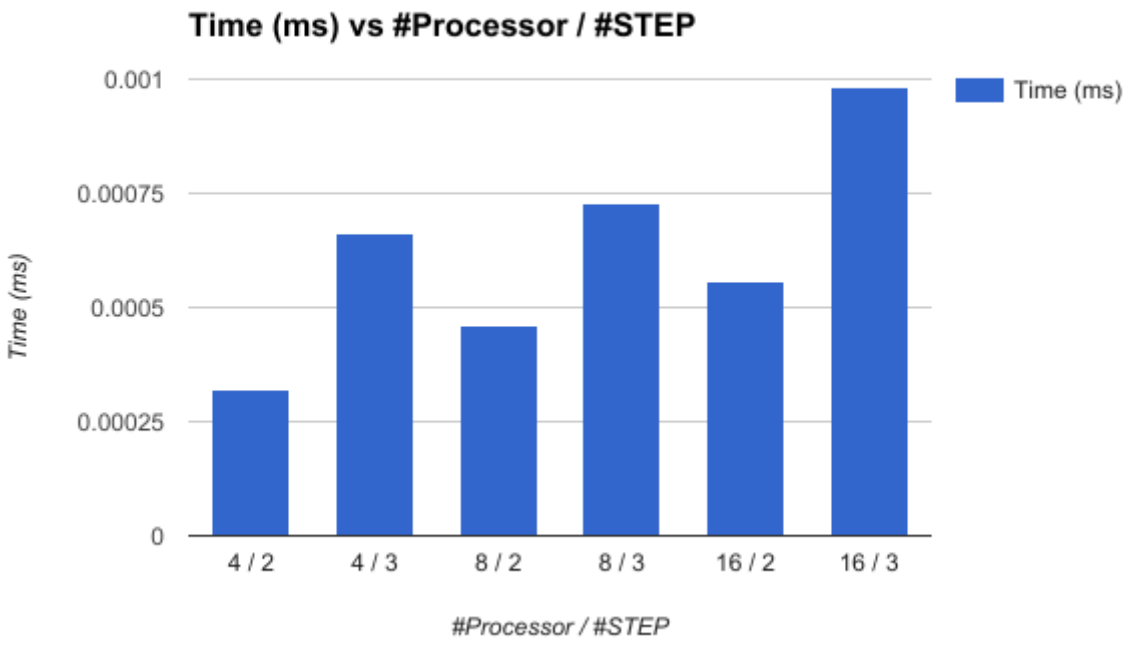
No. of processor 4					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000645	0.000952	0.006358	0.07169	0.30055	1.433485
0.000699	0.001048	0.005821	0.067076	0.265325	1.418062
0.000688	0.000933	0.006085	0.068687	0.313756	1.427033
0.000643	0.001151	0.006529	0.069361	0.268755	1.486862
0.000638	0.001006	0.006503	0.066265	0.264651	1.434588
0.0006626	0.001018	0.0062592	0.0686158	0.2826074	1.440006
No. of processor 8					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000715	0.001017	0.004876	0.06142	0.229806	0.862014
0.000743	0.001056	0.004774	0.056848	0.199077	0.8609
0.000753	0.001135	0.004892	0.065695	0.19856	0.884211
0.000729	0.001069	0.005046	0.05372	0.204839	0.883804
0.000707	0.001054	0.004929	0.065672	0.224674	0.871719
0.0007294	0.0010662	0.0049034	0.060671	0.2113912	0.8725296
No. of processor 16					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000958	0.001356	0.005455	0.058631	0.199441	0.685513
0.001003	0.001541	0.005388	0.04874	0.186378	0.690665
0.000951	0.001393	0.005468	0.056028	0.188844	0.686386
0.001003	0.001337	0.005459	0.050817	0.18161	0.684355
0.001	0.001411	0.005358	0.053669	0.186745	0.701445
0.000983	0.0014076	0.0054256	0.053577	0.1886036	0.6896728

The following tables and graphical representation compare the results of the time measurement for the algorithms implemented in step 2 and step 3. Each table compares the time measurement for each block size. The left hand side column shows the number of processor and the step number i.e 2 or 3 and the right hand side column shows the time measurement in seconds. The values in the right hand side column are the average values calculated in the table 1 and table 2. The information in each table is then graphically represented in the figure following each table.

MATRIX 0008

Table 3.

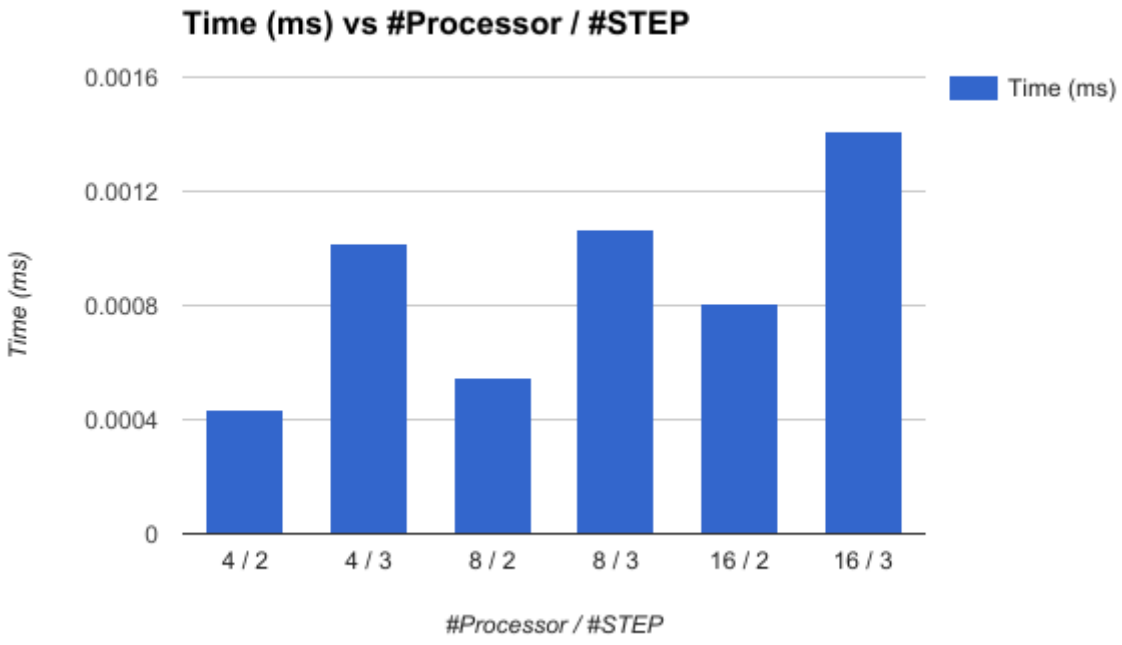
#Processor / #STEP	Time (ms)
4 / 2	0.0003188
4 / 3	0.0006626
8 / 2	0.0004594
8 / 3	0.0007294
16 / 2	0.0005586
16 / 3	0.000983



MATRIX 0016

Table 4.

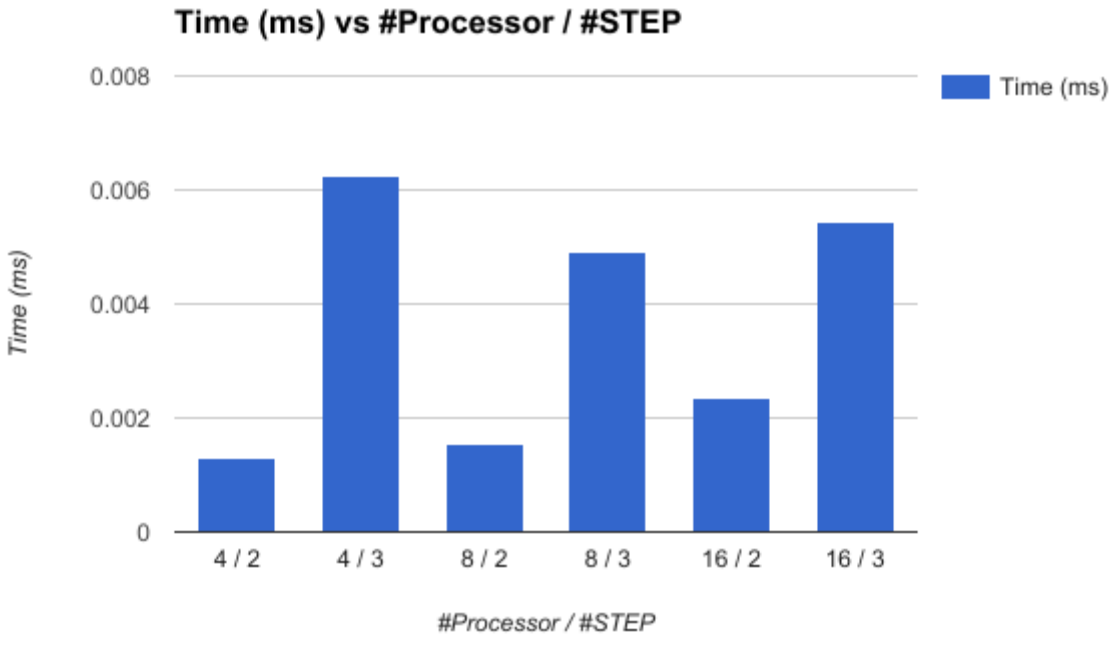
#Processor / #STEP	Time (ms)
4 / 2	0.000438
4 / 3	0.001018
8 / 2	0.0005456
8 / 3	0.0010662
16 / 2	0.0008046
16 / 3	0.0014076



MATRIX 0064

Table 5.

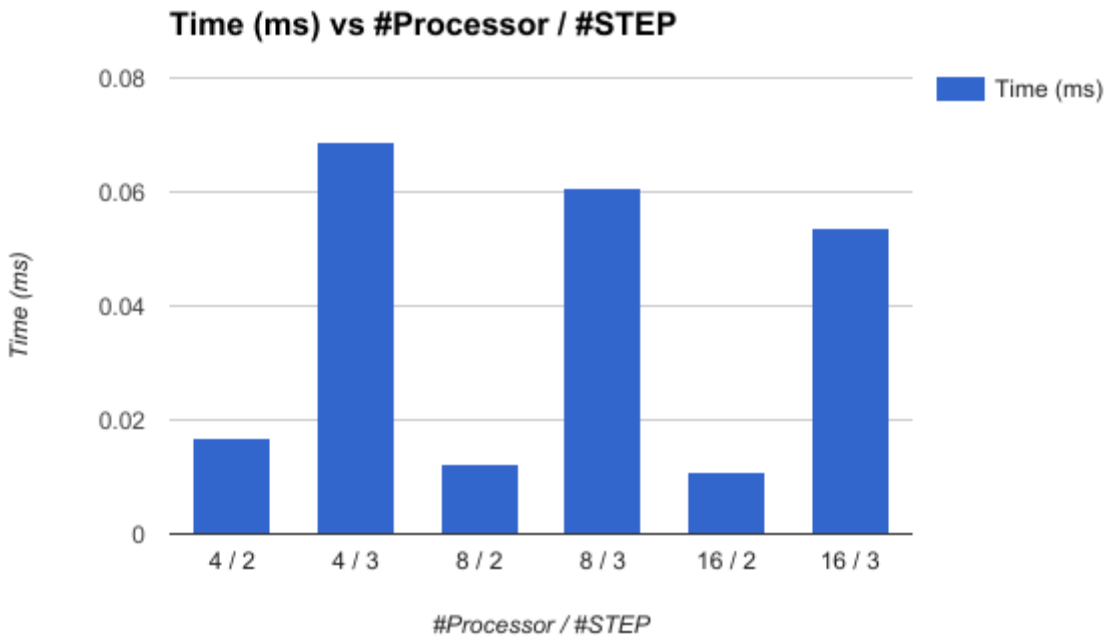
#Processor / #STEP	Time (ms)
4 / 2	0.0012934
4 / 3	0.0062592
8 / 2	0.00156
8 / 3	0.0049034
16 / 2	0.0023418
16 / 3	0.0054256



MATRIX 0256

Table 6.

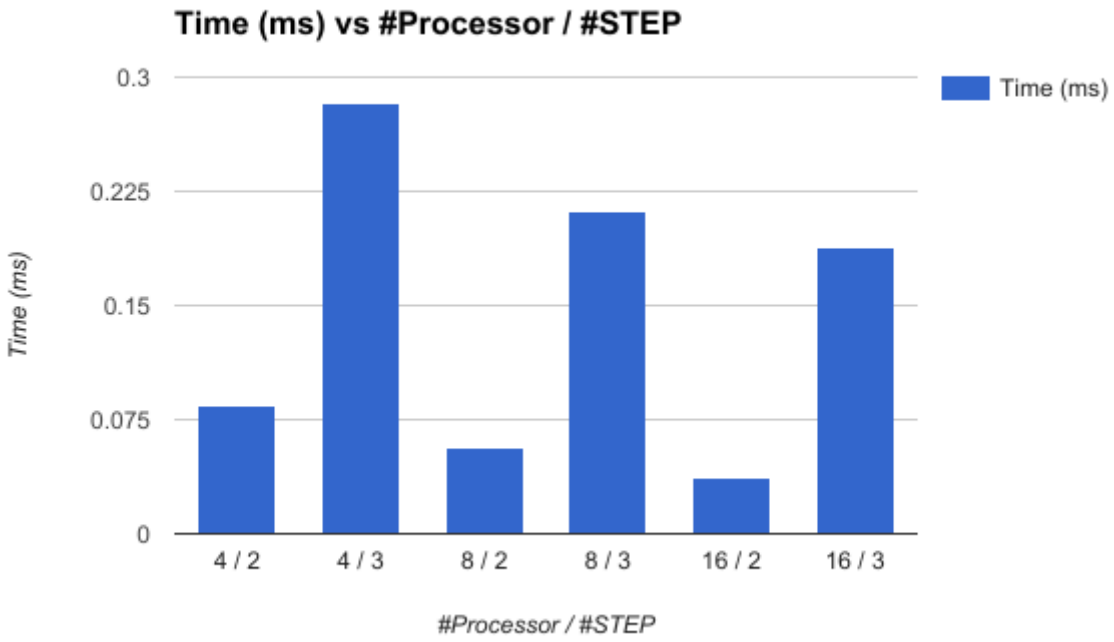
#Processor / #STEP	Time (ms)
4 / 2	0.0169074
4 / 3	0.0686158
8 / 2	0.012134
8 / 3	0.060671
16 / 2	0.0107374
16 / 3	0.053577



MATRIX 0512

Table 7.

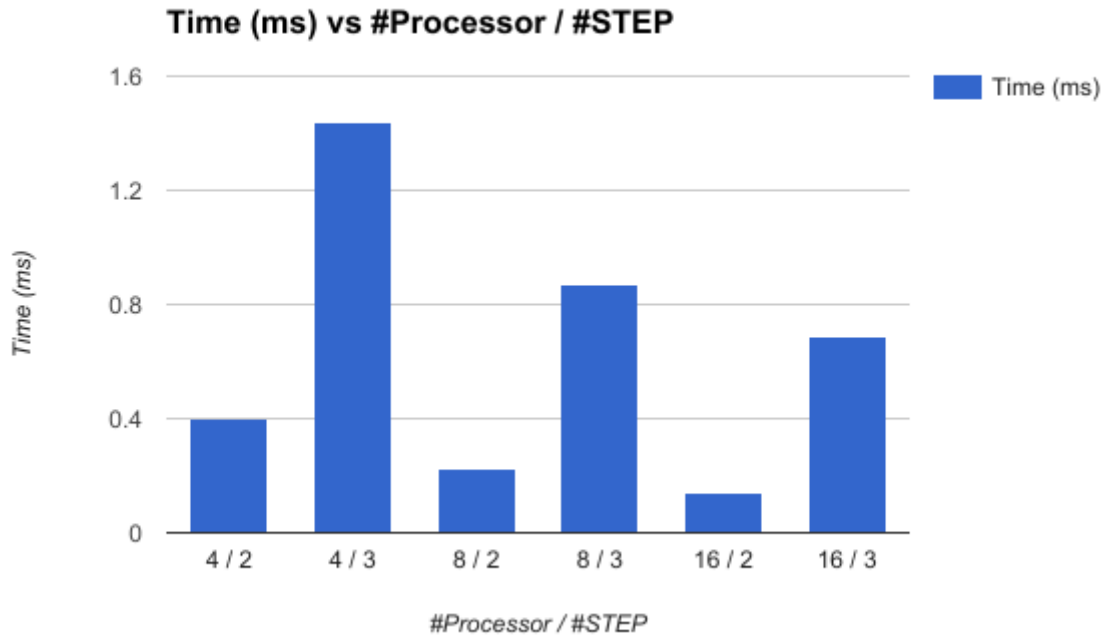
#Processor / #STEP	Time (ms)
4 / 2	0.083914
4 / 3	0.2826074
8 / 2	0.057107
8 / 3	0.2113912
16 / 2	0.0371314
16 / 3	0.1886036



MATRIX 1024

Table 8.

#Processor / #STEP	Time (ms)
4 / 2	0.403359
4 / 3	1.440006
8 / 2	0.224893
8 / 3	0.8725296
16 / 2	0.1372844
16 / 3	0.6896728



The time taken to implement the algorithm can be seen to increase with the increase in the matrix size. This result is as expected because the computation time and the data sharing through MPI commands naturally increases with the increase in the size of the data.
The performance of the algorithm can be seen to improve with the increase in the number of processors which provides for more resources for computation thus increasing the speedup. But this observation is only true for large data sets as can be seen from the results above. The results for eg. for the matrix size of 8 show that with the increase in the number of processors the performance gradually decreases. This is because the size of the data is not large enough to compensate for the communication time between the processes. Thus due to more processes the communication time is more compared to the computation time which leads to the decrease in the performance.
But the efficiency or the performance of the algorithm implemented in step 2 performs better than the efficiency of the algorithm implemented in step 3 i.e. row-cyclic performs better than the total cyclic.
As seen from the representation of data in the the graphs the performance of the step 2 algorithm is better than the performance of the step 3 algorithm. The reason for the obtained result can be explained with the increase in the communication time. In total cyclic data distribution each row is also distributed among different processors which leads to more communication to get access to data from the respective processors.
The way to get the most flexibility for load-balancing, and to have the highest degree of inter-process parallelism available, is a purely cyclic distribution. The cyclic decomposition is maximally good for parallelism and load-balancing, but it's terrible for data access; every neighbouring piece of data you'd want to be able to access to do linear algebra operations is off-processor. On the other hand, the block decomposition is maximally good for data access; you have as large a contiguous chunk of data as possible, so you can do matrix operations on nice big sub-matrices; but it's inflexible for parallelism and can cost in terms of load-balancing [2].

Gaussian elimination using OpenMP

The following table i.e. table 9 show the time measurement of the algorithm implemented in the step 5 of the project assignment. The table is the time measurement for the algorithm implemented for the shared memory systems using OpenMP. All time measurement is done in the unit millisecond (ms). The algorithm was run for different number of threads i.e. 8 and 16 and different number of data block size i.e. 8, 16, 64, 256, 512 and 1024. Every combination was run for 5 times on one node of the cluster tegner at PDC Center for High Performance Computing. The average was then calculated and presented in the table at the end of the 5 computations for each combination.

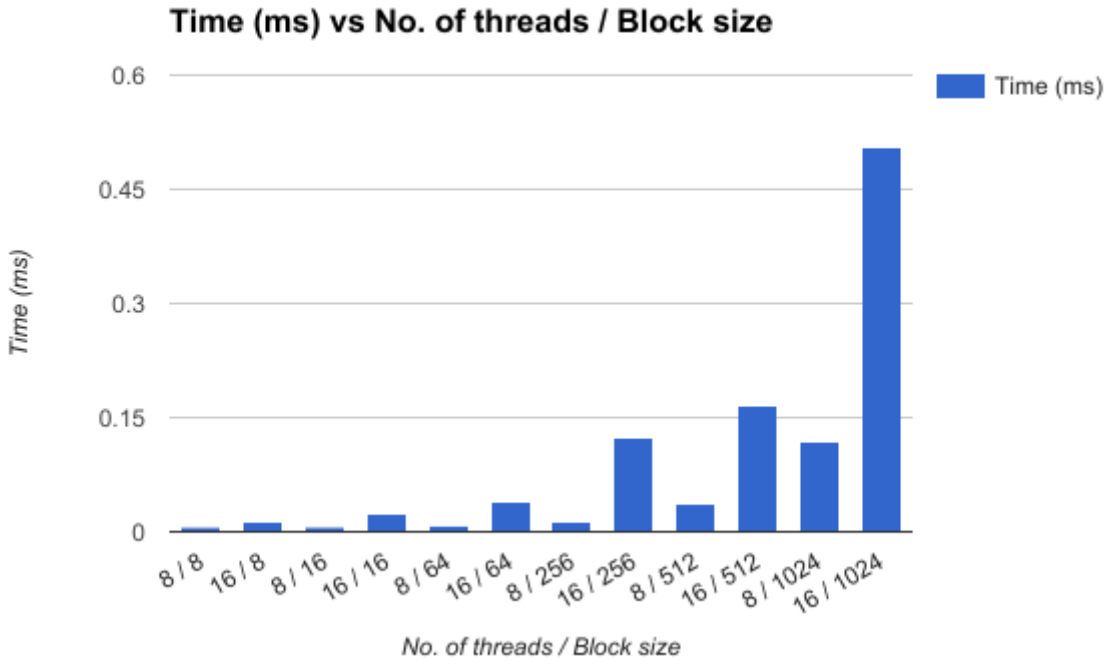
Table 9.

No. of threads 8					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.006683	0.006786	0.010093	0.014469	0.037572	0.117253
0.00475	0.005648	0.009157	0.01518	0.03972	0.119737
0.006635	0.009031	0.005867	0.014146	0.0324	0.100209
0.005049	0.004906	0.008254	0.011542	0.031946	0.149823
0.008777	0.006467	0.008957	0.016684	0.038024	0.100428
0.0063788	0.0065676	0.0084656	0.0144042	0.0359324	0.11749
No. of threads 16					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.008034	0.032827	0.057658	0.084322	0.226676	0.460411
0.015436	0.018093	0.019894	0.149999	0.135572	0.543046
0.007096	0.023126	0.022845	0.160744	0.135572	0.532768
0.01911	0.022902	0.047717	0.121882	0.215014	0.525214
0.014758	0.020569	0.054902	0.098573	0.122658	0.47145
0.0128868	0.0235034	0.0406032	0.123104	0.1670984	0.5065778

The following table shows the average that was calculated in table 9. The left hand side column shows the number of threads used i.e. either 8 or 16 and the respective block size.

Table 10.

No. of threads / Block size	**Time (ms)**
8 / 8	0.0063788
16 / 8	0.0128868
8 / 16	0.0065676
16 / 16	0.0235034
8 / 64	0.0084656
16 / 64	0.0406032
8 / 256	0.0144042
16 / 256	0.123104
8 / 512	0.0359324
16 / 512	0.1670984
8 / 1024	0.11749
16 / 1024	0.5065778



The implementation time increases as expected with the increase in the size of the input data. But the run time also increases or the performance decreases with the increase in the number of threads. This observation can be attributed to the concept of false sharing. In my program although I have used the methods to reduce false sharing but it's effect can still be clearly seen in the implementation of the algorithm.
False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in figure below. This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access [3].

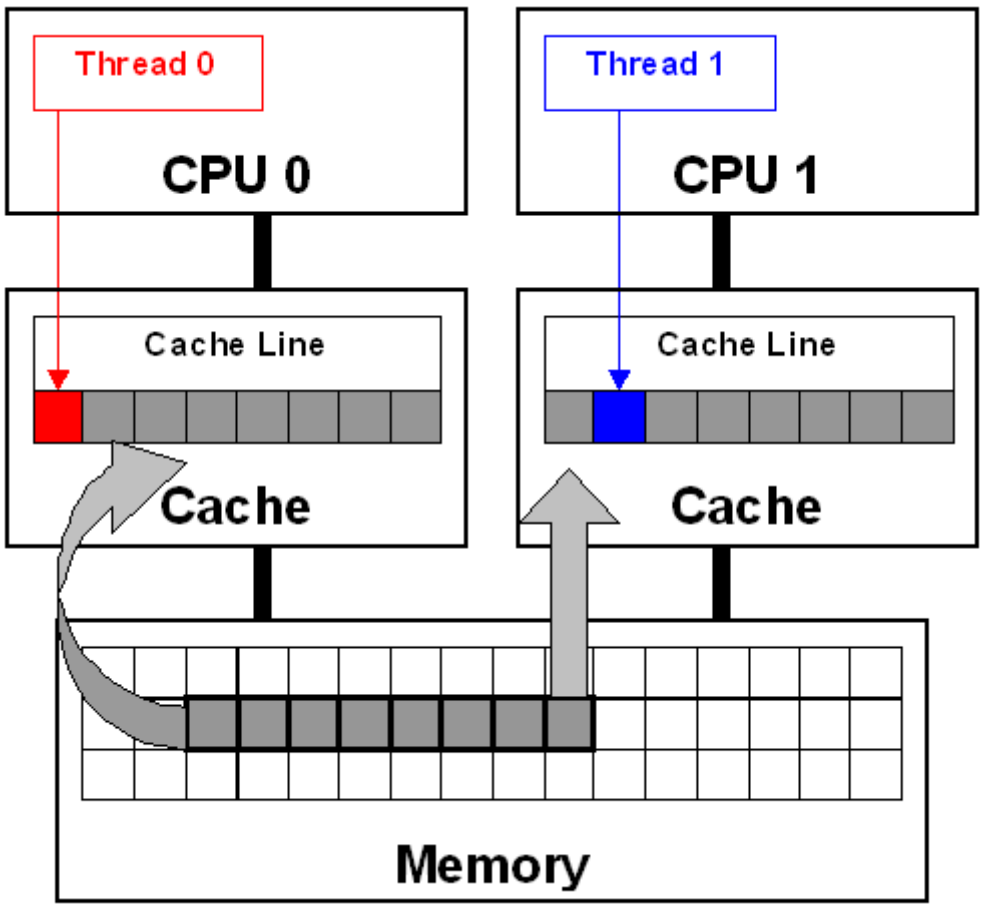


Figure 1. Source [3]

To obtain parallelism I parallelise the for loop in the function max_col_loc() and exchange_row() by using #pragma omp parallel and passing the private variable in both the respective cases to avoid false sharing. I also parallelise the loop used for calculating the elimination factors in the main() function again by passing the private variables. Finally I use #pragma omp parallel for reduction(+:sum) with reduction to parallelise the implementation of backward substitution.

Column-cyclic data distribution

In the column-cyclic data distribution the communication pattern had to be changed a bit compared to that of the row-cyclic data distribution. The pivot row was broadcasted to all the processes since different processes possess the different part of both the current row and the pivot row. The exchange of the pivot row and the current row was performed on the individual processes without the need for any broadcast. The elimination factors calculated on one processor for the respected step were broadcasted to all the other processes since all the processes will need the elimination factors. Then the local entries were computed on each individual processes depending on the part of the matrix they were processing on. At the end the broadcast operation was used to broadcast the sum and the x matrix during the backward substitution.

The following table i.e. table 11 show the time measurement of the algorithm implemented in the step 6 of the project assignment. The table is the time measurement for the algorithm implemented for the column-cyclic data distribution. All time measurement is done in the unit millisecond (ms). The algorithm was run for different number of processes i.e. 4, 8 and 16 and different number of data block size i.e. 8, 16, 64, 256, 512 and 1024. Every combination was run for 5 times on one node of the cluster tegner at PDC Center for High Performance Computing. The average was then calculated and presented in the table at the end of the 5 computations for each combination.

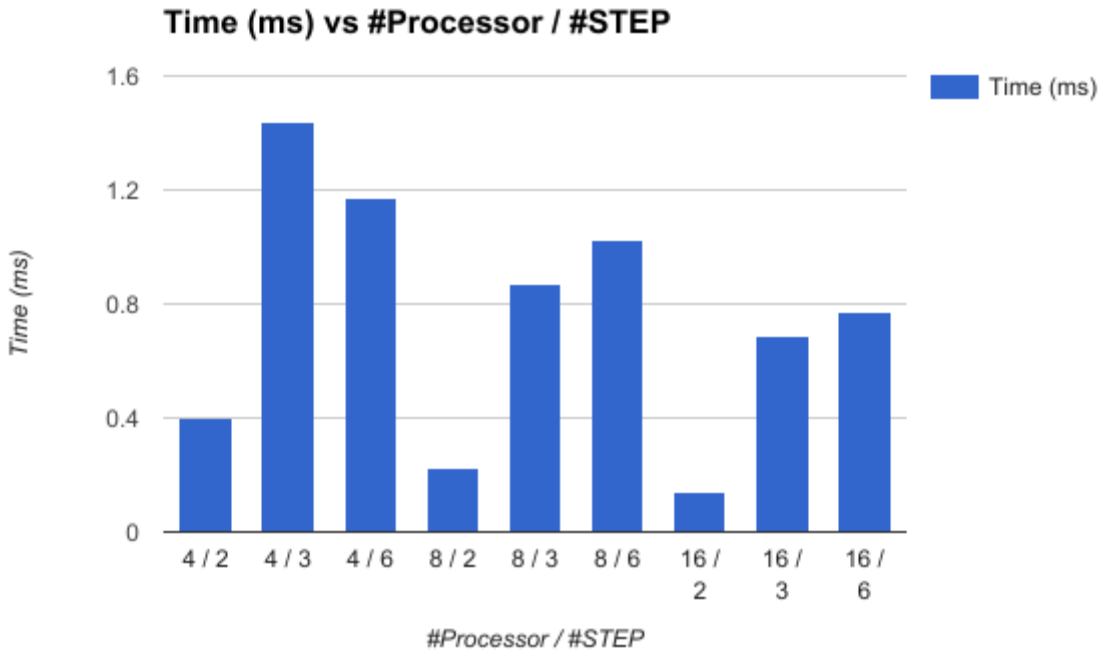
Table 11.

No. of processor 4					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000325	0.000452	0.003215	0.051024	0.233049	1.167955
0.000278	0.000451	0.003128	0.05171	0.231265	1.165967
0.000278	0.000444	0.003099	0.051221	0.230935	1.130489
0.000274	0.000455	0.004243	0.049692	0.230842	1.191261
0.000298	0.00044	0.003125	0.0543	0.22816	1.206001
0.0002906	0.0004484	0.003362	0.0515894	0.2308502	1.1723346
No. of processor 8					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000397	0.000641	0.003166	0.05463	0.182615	1.04483
0.000367	0.00073	0.003195	0.040993	0.202477	1.013898
0.000394	0.000678	0.003154	0.041203	0.196441	1.049217
0.000402	0.000628	0.002993	0.042724	0.168639	1.029698
0.000371	0.000707	0.003912	0.054076	0.172259	0.998962
0.0003862	0.0006768	0.003284	0.0467252	0.1844862	1.027321
No. of processor 16					
Block size 8	**Block size 16**	**Block size 64**	**Block size 256**	**Block size 512**	**Block size 1024**
0.000555	0.000814	0.00374	0.04537	0.168608	0.787554
0.000545	0.000955	0.003927	0.046666	0.166686	0.766623
0.000599	0.000816	0.003838	0.039936	0.168718	0.777294
0.000556	0.000861	0.003937	0.041732	0.171668	0.766035
0.000551	0.000989	0.003366	0.036619	0.164524	0.771628
0.0005612	0.000887	0.0037616	0.0420646	0.1680408	0.7738268

The following tables and graphical representation compare the results of the time measurement for the algorithms implemented in step 2, step 3 and step 6. The table compares the time measurement for 1024 block size. The left hand side column shows the number of processor and the step number i.e 2, 3 or 6 and the right hand side column shows the time measurement in seconds. The values in the right hand side column are the average values calculated in the table 1, table 2 and table 11. The information in the table is then graphically represented in the figure following each table.

Table 12.

#Processor / #STEP	Time (ms)
4 / 2	0.403359
4 / 3	1.440006
4 / 6	1.1723346
8 / 2	0.224893
8 / 3	0.8725296
8 / 6	1.027321
16 / 2	0.1372844
16 / 3	0.6896728
16 / 6	0.7738268



The performance in column-cyclic data distribution is declining for every varying number of processor. The data distribution is such that the load is not balanced properly and the example for the same can be the calculation of elimination factor. Only one process is performing the calculation while all the other processes are idle at that time not doing any activity. Thus due to this mis match of the data and the calculation pattern the performance takes a hit for the column-cyclic data distribution.

References

[1] T. Rauber, Parallel programming, 2nd ed. [Place of publication not identified]: Springer, 2014, pp. 418-420.
[2] J. Dursi, "Understanding Block and Block-Cyclic Matrix Distributions", www.stackoverflow.com, 2015. [Online]. Available: https://stackoverflow.com/questions/31076953/understanding-block-and-block-cyclic-matrix-distributions. [Accessed: 06- Jun- 2017].
[3]"Avoiding and Identifying False Sharing Among Threads | Intel® Software", Software.intel.com, 2017. [Online]. Available: https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads. [Accessed: 06- Jun- 2017].