1) Problem statement and description

Prerequisite definitions:

An n × n real symmetric matrix M is positive definite if $z^T M z > 0$ for all non-zero vectors z with real entries (i.e z belongs to Real space R^n), where z^T denotes the transpose of z.

Statement:

Decomposition of a real, symmetric, positive definite matrix into a product of lower triangular matrix with strictly positive diagonal entries and its conjugate transpose. A complex conjugate of a real matrix is the transpose of the matrix. (The problem is addressed in real space R only, however in general these matrices can be extended to complex space also)

Cholesky decomposition can be used to solve linear system of equations in the same way as LU decomposition is used. It also has applications in the problems of Non-linear optimization, Monte-Carlo simulation, Kalman filters, Computational fluid dynamics etc.

2) Sequential Algorithm for Cholesky decomposition

The sequential algorithm described below does in place modifications to form a lower triangular component of LL^T factorization of matrix A of n by n size.

Pseudo Code:

```
FOR k = 1 to n
    A[k][k] = squareRoot(A[k][k]);
    FOR i=k+1 to n
        A[i][k] = A[i][k]/A[k][k];
    END
    FOR j = k + 1 to n
        FOR i=j to n
            A[i][j] = A[i][j] – A[i][k]*A[j][k]
        END
    END
END
```

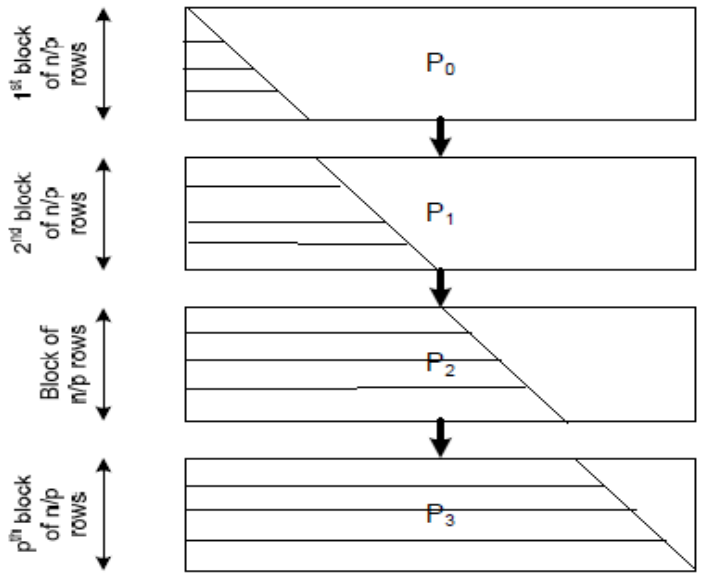
3. Parallel algorithm for Cholesky decomposition

a) Inter processor connectivity:

If the input matrix dimensions are n rows and n coloums, and number of processors lauched are p, then each processor has a share of n/p rows for it's own processing. All the p processors are connected in the form of a 1D torus.

Each processor's source processor id is one lesser than its own and the destination is one higher that its own.

b) Parallel Algorithm: Matrix is row partitioned as shown below. Eg. p=4;n=16.



Step (1): Distribute the actual data among p processors into m/p blocks.

Step (2): Each processor, based on their id, determines the number of recieve and send transactions in which they need to participate during the computation. Communications are initiated by the processor with id=0 once it computes the first diagonal element in the whole matrix

Step (3): For ith recieve from its previous processor, the processor updates the corresponding ith coloumn in its own data and forwards the recieved data.

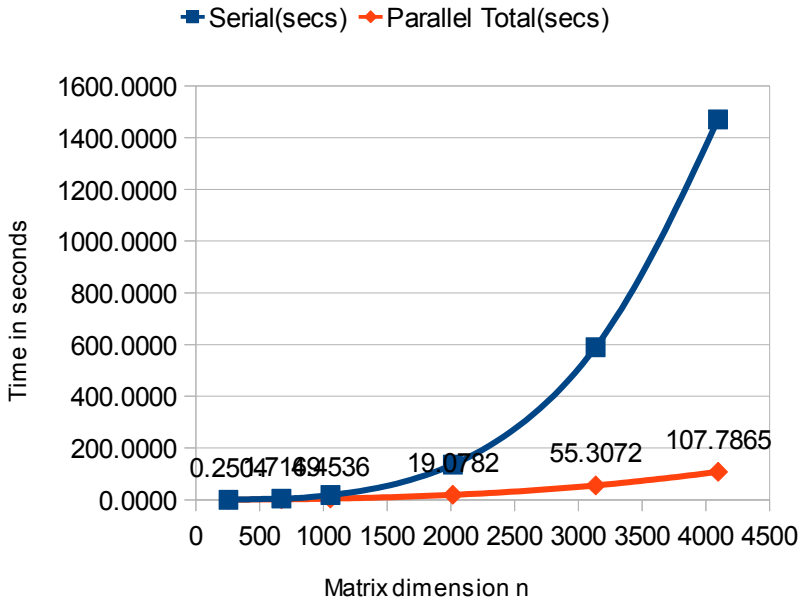
Step (4): If the processor's all recieve transactions have been completed, the processor will repeat steps (2), (4) and (5) considering its own rows as virtual recieves and there by process the remaining coloumns in its own data. In other words, step(4) replaces step(3) in each processor computation from now on.

c) Step(5): Forward data from every recieve(includes the virtual recieve from self) transaction to the other neighbour.

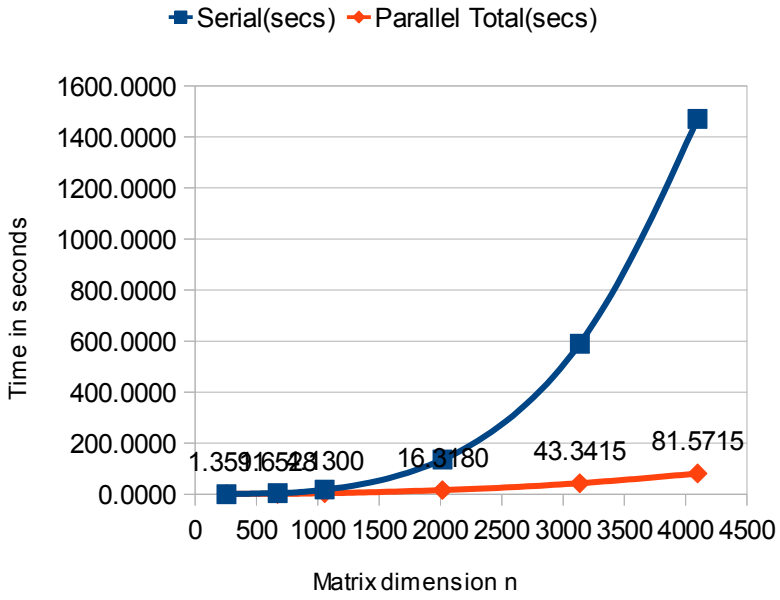
4. Sample runs and results

Below listed are a sample of results provided for matrix sizes ranging from 256-by-256 to 4096-by-4096. Lauching more number of processors for small matrix dimensions is costlier because more amount of time needs to be spent on communicating where as an equivalent amount of extra work could be done by the same processor instead of transferring work. This can be inferred from the increasing parallel execution time for matrix 256-by-256 as we move from p=8 to p=32.

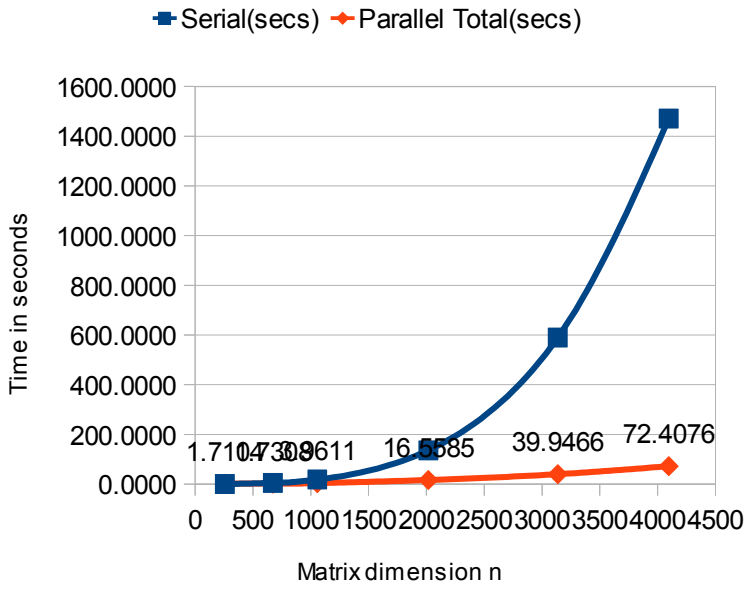
Matrix Dimensions	# of processes	Serial(secs)	Parallel Total(secs)	Parallel Transfer(secs)	Parallel Compute(secs)	Speed up	Efficiency
256by256	8	0.2798	0.2504	0.2366	0.0138	1.1176	0.1397
672by672	8	4.6758	1.7169	1.4830	0.2339	2.7234	0.3404
1056by1056	8	18.4209	4.4536	3.5530	0.9006	4.1362	0.5170
2016by2016	8	135.8836	19.0782	12.7549	6.3233	7.1225	0.8903
3136by3136	8	589.8952	55.3072	31.3699	23.9373	10.6658	1.3332
4096by4096	8	1470.4754	107.7865	53.8617	53.9248	13.6425	1.7053



Matrix Dimensions	# of processes	Serial(secs)	Parallel Total(secs)	Parallel Transfer(secs)	Parallel Compute(secs)	Speed up	Efficiency
256by256	32	0.2798	1.7104	1.6968	0.0136	0.1636	0.0051
672by672	32	4.6758	1.7308	1.6598	0.0710	2.7015	0.0844
1056by1056	32	18.4209	3.9611	3.7188	0.2423	4.6505	0.1453
2016by2016	32	135.8836	16.5585	14.9538	1.6047	8.2063	0.2564
3136by3136	32	589.8952	39.9466	33.9194	6.0273	14.7671	0.4615
4096by4096	32	1470.4754	72.4076	58.9734	13.4342	20.3083	0.6346



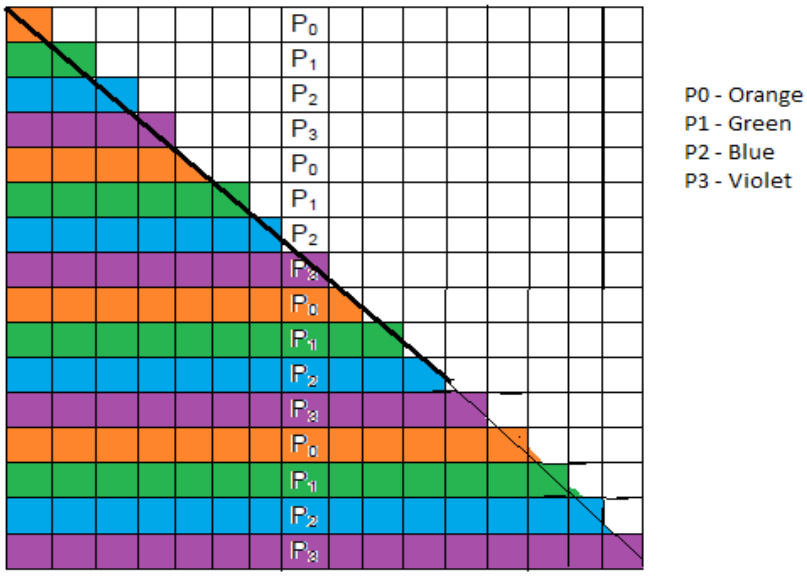
Matrix Dimensions	# of processes	Serial(secs)	Parallel Total(secs)	Parallel Transfer(secs)	Parallel Compute(secs)	Speed up	Efficiency
256by256	16	0.2798	1.3591	1.3515	0.0076	0.2059	0.0129
672by672	16	4.6758	1.6528	1.5339	0.1188	2.8291	0.1768
1056by1056	16	18.4209	4.1300	3.6763	0.4538	4.4602	0.2788
2016by2016	16	135.8836	16.3180	13.1621	3.1559	8.3272	0.5205
3136by3136	16	589.8952	43.3415	31.4352	11.9064	13.6104	0.8506
4096by4096	16	1470.4754	81.5715	54.8107	26.7608	18.0268	1.1267



5. Possible Improvements over current implementation

An immediate enhancement could be block partitioning the matrix and working over a p-by-p two dimensional mesh. Although coloumn partitioning can also be implemented, it would be approximately of same perforamce speedup since same of number of operations are performed. Current algorithm can also be modified to take rows in cyclic pattern to have better load balance between the processors since the work reduces as we progress from P0 to Pn-1.

A more optimized but complex approach can be row partitioning in cyclic manner as shown below for n=16 i.e 16-by-16 matrix and p=4.



Cyclic row partitioning example with 4 processors for a 16 by 16 positive definite matrix

Darkened Line covers the diagonals

6. Conclusion

Compared against the single core my parallel implementation have showed decent amount of performance speedup. Super-linear speedup is not feasible due to the inherent data dependency in the nature of the problem itself. However, better matrix partitioning strategies like row cyclic partitioning can give much higher performace speedups.

References:

- http://en.wikipedia.org/wiki/Cholesky_decomposition - Sequential Implementation algorithm has been synthesized from Wikipedia.