A Note on Parallel Implementation of the Cholesky Factorization

Jaehoon Seol Robert van de Geijn

October 18, 1995

Abstract

In this paper, we revisit the parallel implementation of the Cholesky factorization on distributed memory parallel supercomputers. While traditional methods take their queue from the implementation of the LU factorization, which requires a right-looking variant, we show that the Cholesky factoration can be more naturally implemented using the left-looking variant. The resulting algorithm is simple, attains high performance, and is scalable. We report performance of our algorithm on a Intel Paragon supercomputer and compare that performance against the current implementation incorporated in the publicly available ScaLAPACK library.

1 Introduction

The Scalapack implementation of the Cholesky factorization is based on right looking algorithm [2]. But this right looking version of Parallel Cholesky Factorization needs symmetric rank update(PDSYRK) which broadcast a column of blocks across all columns of processes and then transpose that column blocks. This transpose operation is pretty expensive in a distributed memory wrap-mapped context and difficult to implement. Our implementation of the Cholesky factorization is based on left looking algorithm which will be described in later section. By using this left looking version, we can avoid transpose operation . So, we can expect better performance . And left looking version of Parallel Cholesky factorization is easier to implement.

2 Target Architectures

Our current implementation assumes a two dimensional physical mesh of processing nodes, with bidirectional links between nodes, and wormhole (cut-through) routing. Furthermore, we assume it is possible to model the time required for sending a message of length n bytes between any two nodes by

$$\alpha + n\beta$$

where α equals the latency of the network, and β is the time per item, in the absense of network conflicts. Communication is single ported, meaning a processor can only send to, or receive from, one other node at a given time. When two messages traverse the same physical link, we assume they time-share the bandwidth of that link.

In addition to these features, we also assume our target machines use hierarchical memory. This minimize data movement between memories of different levels when we use block-partitioned algorithm.But our test was done on a 4 node Intel Paragon.

3 Data Decomposition

The fundamental data object used in the ScaLAPACK library is the block-partitioned matrix. They have used block-cyclic method for distributing such a matrix over two-dimentional mesh of processors. In order to make our algorithm compatible with the ScaLAPACK library, we have used the same method. In this section, we describe the block cyclic decomposition which we have used.

In the block cyclic decomposition the mapping of the global index, m, can be expressed as $m \Rightarrow (p, b, i)$,where p is the process number, b is the block number, and i is the local index within block b to which m is mapped. Thus, if the block size is r, we can define the block cyclic data decomposition as

$$m \Rightarrow (\lfloor \frac{m mod T}{r} \rfloor, \lfloor \frac{m}{T} \rfloor, m mod r),$$

$T = rP$. It should be noted that this becomes cyclic decomposition if we take $r = 1$, and block decomposition if we take $r = \lceil \frac{M}{P} \rceil$. In the later case, there is a single block in each process .

4 The Sequential Algorithm

Cholesky factorization factors an $N * N$, symmetric ,positive-definete matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where$U$ is upper triangular). It is assumed that the lower triangular portion of A is stored in the lower triangule of a two-dimensional array and that the computed elements of L overwrite the given elements of A.

4.1 Right-looking variant

Partition matrices A and L so that

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right)$$

From this partitioning, we derive the equations

$$A_{11} = L_{11} L_{11}^T \tag{1}$$

$$A_{21} = L_{21} L_{11}^T \tag{2}$$

$$A_{22} = L_{21} L_{21}^T + L_{22} L_{22}^T \tag{3}$$

A resulting algorithm, which performs most computation using matrix-matrix operations, is given by

1. Factor $A_{11} \rightarrow L_{11}$.
2. Compute L_{21} from the triangular system $A_{21} = L_{21} L_{11}^T$.
3. Update $A_{22} \rightarrow A_{22} - L_{21} L_{12}^T$.
4. Continue recursively with the updated A_{22}.

4.2 Left-looking variant

To explain the alternative *left-looking* variant, consider the partitioning

$$A = \left(\begin{array}{c|c|c} \mathbf{A_{11}} & \star & \star \\ \hline \mathbf{A_{21}} & A_{22} & \star \\ \hline \mathbf{A_{31}} & A_{32} & A_{33} \end{array}\right) = \left(\begin{array}{c|c|c} \mathbf{L_{11}} & 0 & 0 \\ \hline \mathbf{L_{21}} & L_{22} & 0 \\ \hline \mathbf{L_{31}} & L_{32} & L_{33} \end{array}\right) \left(\begin{array}{c|c|c} \mathbf{L_{11}}^T & \mathbf{L_{21}}^T & \mathbf{L_{31}}^T \\ \hline 0 & L_{22}^T & L_{32}^T \\ \hline 0 & 0 & L_{33}^T \end{array}\right) \tag{4}$$

The assumption is that bold-face parts of the lower triangular matrix have already been computed, and have overwritten the corresponding parts of A. The rest of the matrix has not been updated at all, and the object of the next step is to compute the next parts of the lower triangular matrix, L_{22} and L_{32}, overwriting the corresponding parts of A. For the above equation, we derive

$$\left(\frac{A_{22}}{A_{32}}\right) = \left(\frac{L_{21}}{L_{31}}\right) L_{21}^T + \left(\frac{L_{22}}{L_{32}}\right) L_{22}^T$$

or

$$\begin{array}{rcl} \left(\frac{A_{22}}{A_{32}}\right) & \leftarrow & \left(\frac{A_{22}}{A_{32}}\right) - \left(\frac{L_{21}}{L_{31}}\right) L_{21}^T \\ A_{22} & \rightarrow & L_{22}L_{22}^T \\ A_{32} & = & L_{32}L_{22}^T \end{array}$$

The algorithm of left looking version of cholesky factorization can be given as follows using the above equations

1. Update the current panel according to Eqn. **??**.
2. Perform a Cholesky factorization of updated A_{22} (Eqn. **??**).
3. Now that L_{22} known, compute L_{32} from Eqn. **??**.
4. Continue recursively by repartitioning the matrix.

5 Parallel Implementation

In both our implementations, we will assume the following distribution of the matrix to nodes: For convenience, let $n = Nn_b$, where n_b is called the block size, and partition the matrix

$$A = \left(\begin{array}{c|c|c|c} \tilde{A}_{00} & \tilde{A}_{01} & \cdots & \tilde{A}_{0(N-1)} \\ \hline \tilde{A}_{10} & \tilde{A}_{11} & \cdots & \tilde{A}_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline \tilde{A}_{(N-1)0} & \tilde{A}_{(N-1)1} & \cdots & \tilde{A}_{(N-1)(N-1)} \end{array}\right)$$

where $\tilde{A}_{ij}$ is $n_b \times n_b$ and is assigned to node $\mathbf{P}_{(i \bmod r)(j \bmod c)}$.

5.1 Right-looking Variant

To derive a parallel implementation of the right-looking variant, we consider the operations given in Section **??** above. The following parallel implementation will implement the right-looking algorithm: For $k = 0, \ldots, N-1$,

1. Factor $\tilde{A}_{kk} \rightarrow \tilde{L}_{kk}\tilde{L}_{kk}^T$, overwriting the lower portion of $\tilde{A}_{kk}$.
2. Broadcast $\tilde{L}_{kk}$ within the column of nodes that owns $\tilde{A}_{kk}$.
3. All nodes in the column of nodes that owns $\tilde{A}_{kk}$ perform their part of solving

$$\begin{aligned}
\tilde{L}_{(k+1)k}\tilde{L}_{kk}^T &= A_{(k+1)k} && (5)\\
\tilde{L}_{(k+2)k}\tilde{L}_{kk}^T &= A_{(k+2)k} && (6)\\
\vdots \quad & \quad \vdots && (7)\\
\tilde{L}_{(Nk}\tilde{L}_{kk}^T &= A_{Nk} && (8)
\end{aligned}$$

4. To accomplish $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T$ we must
   - Distribute resulting $\tilde{L}_{ik}$, $i = k+1, \ldots, N$ within the row of nodes that holds all $\tilde{A}_{ij}$, i.e. broadcast $\tilde{L}_{ik}$ within rows.
   - Distribute resulting $\tilde{L}_{ik}$, $i = k+1, \ldots, N$ within the row of nodes that holds all $\tilde{A}_{ji}$, i.e. broadcast $\tilde{L}_{ik}$ within appropriate columns. It is this operation that is actually very tricky to implement.
5. Update local portions of $\tilde{A}_{ij} \leftarrow \tilde{A}_{ij} - L_{ik}L_{jk}^T$ in parallel.

5.2 Left-looking Variant

Jeahoon: Please rewrite the following in a similar style.

The parallel implementation of the corresponding left looking cholesky factorization can be given inductively using the following steps

1. The process P_i, which has the $n_b * n_b$ diagonal block A_{11} performs the Cholesky factorization of A_{11}.

$$A_{11} \longrightarrow L_{11}L_{11}^T$$

2. The lower triangular matrix L_{11} is broadcast columnwise by P_i down all rows in the current column of processes, which compute the

column of blocks of L_{i1} for $i = 2, 3$.

$$L_{i1} \longleftarrow A_{i1}(L_{11}^T)^{-1}$$

3. Broadcast the matrix L_{21}^T along the column and compute $L_{i1}L_{21}^T$ for $i = 2, 3$.

4. By doing the reduction to one along the row, we can find $L_{22}L_{22}^T$ and $L_{32}L_{22}^T$.

5.3 Estimated Run Time and Scalability

An important metric used to measure the performance of an parallel algorithm is parallel efficiency. Parallel effiency, $E(N, P)$, for a problem of size N on P processors is defined by

$$E(N, P) = \frac{1}{P}\frac{T_{seq}(N)}{T(N, P)}$$

where $T(N, P)$ is the runtime of parallel algorithm , and $T_{seq}(N)$ is the runtime of the best sequential algorithm. An implementation is said scalable if the parallel efficiency is an increasing function of $\frac{N}{P}$. In the case of dense matrix computations, $N = n^2$.

Let us consider what computation is needed to implement the parallel left looking version of Cholesky factorization. Before we analyze time complexity,we will make some simplifying assumption that the dimension n of the matrix A is an integer multiple of k,the size of the block. If we use minimum spanning tree(MST) Broadcast and Reduction-to-One and ignore the time cost caused by packing required before Broadcast and Reduction-to-One, the cost $T(n, rc, i)$ of our algorithm at ith iteration is given by

$$\begin{aligned}
T(n, rc, i) &= \frac{k^3}{3}\gamma && (9)\\
&+ \lceil logr \rceil(\alpha + \frac{k^2}{2}\beta) && (10)\\
&+ k^3\lceil \frac{\frac{n}{k} - i}{r} \rceil\gamma && (11)\\
&+ \lceil logr \rceil(\alpha + \lceil \frac{i}{c} \rceil k^2\beta) && (12)\\
&+ 2k^3\lceil \frac{\frac{n}{k} - i}{r} \rceil\lceil \frac{i}{c} \rceil\gamma && (13)
\end{aligned}$$

$$
\begin{aligned}
&+ \quad \lceil logc \rceil (\alpha + \lceil \frac{\frac{n}{k} - i}{r} \rceil k^2(\beta + \gamma)) && (14) \\
&+ \quad \lceil \frac{\frac{n}{k} - i}{r} \rceil k^2 \gamma && (15)
\end{aligned}
$$

where (1) equals the time required for the Cholesky factorization of each diagonal block; (2)equals the time required for broadcast of the lower triangular part which results from the Cholesky factorization of each diagonal block;(3)equals the time required for DTRSM operation; (4) equals the time required for broadcast of the ith row panel (5) equals the time required for DGEMM operation; (6) equals the time required for reduction to one; (7) equals the time required for DAXPY operation.

Thus, the total cost $T(n, rc)$ of our algorithm is given by

$$T(n, rc) = \sum_{i=1}^{\frac{n}{k}} T(n, rc, i)$$

This amounts to approximately

$$
\begin{aligned}
T(n, rc) \quad &= \quad \alpha[(2\lceil logr \rceil + \lceil logc \rceil)\frac{n}{k}] && (16) \\
&+ \quad \beta[\lceil logr \rceil (\frac{kn}{2} + \frac{n^2}{2c}) + \lceil logc \rceil \frac{n^2}{2r}] && (17) \\
&+ \quad \gamma[\frac{k^2 n}{3} + (\frac{1}{2r} + \frac{2}{c})kn^2 + (1 + \lceil logc \rceil)\frac{n^2}{2r}] && (18)
\end{aligned}
$$

In order to establish the scalability of our algorithm, we will analyse the case where $c = r = \sqrt{p}$. Given the complexity in (8),(9),and(10), the estimated speedup is

$$
\begin{aligned}
S(n, p) \quad &= \quad \frac{n^3 \gamma}{3T(n, rc)} && (19) \\
&\simeq \quad \frac{n^3 \gamma}{3[\beta(\lceil log\sqrt{p} \rceil \frac{n^2}{2\sqrt{p}} + \lceil log\sqrt{p} \rceil \frac{n^2}{2\sqrt{p}}) + \gamma(\frac{1}{2\sqrt{p}} + \frac{2}{\sqrt{p}})kn^2 + (1 + \lceil log\sqrt{p} \rceil)\frac{n^2}{2\sqrt{p}}]} && (20) \\
&= \quad \frac{p}{3[\beta\lceil log\sqrt{p} \rceil \frac{\sqrt{p}}{n} + \gamma(5k + 1 + \lceil log\sqrt{p} \rceil)\frac{\sqrt{p}}{2n}]} && (21)
\end{aligned}
$$

The corresponding parallel efficiency is

$$E(n, p) \quad = \quad \frac{S(n, p)}{p} \qquad (22)$$

$$= \frac{1}{3[\beta\lceil log\sqrt{p}\rceil\frac{\sqrt{p}}{n} + \gamma(5k+1+\lceil log\sqrt{p}\rceil)\frac{\sqrt{p}}{2n}]} \quad (23)$$

$$= \frac{1}{\beta\Theta(\frac{\sqrt{p}log\sqrt{p}}{n}) + \gamma\Theta(\frac{\sqrt{p}log\sqrt{p}}{n})} \quad (24)$$

Once p gets to be reasonable large, we can treat $log\sqrt{p}$ as constant for practical purpose.So, If we matain memory use per node ,this algorithm will maintain efficiency.

6 Experimental Results

The 4 node Intel Paragon which we have used has a 16Kbyte instruction cache and a 16Kbyte data cache. The size of core memory is 32Mbyte but 12Mbyte is used by operating system. The value of α and β are xx microseconds and xx microseconds respectively for data size less than xx. For data size larger than xx, the value of α and β are xx microseconds and xx microseconds respectively.The reason for this difference in latency is that messages of xx bytes or less travel using route-acquisition protocol, whereas larger messages need a kind of hand-shaking communication before the messages can be sent.

Smaller block size produces better load balancing on the nodes and thus decrease the amount of idle time. However, this decrease of idle time is at the expense of an increase in the communication overhead. Larger blocksize, on the other hand,decrease the communication overhead and makes it possible to take advantage of the hierarchical memory stucture which increase the performance of the float point operations on each node. But they may result in load imbalancing between nodes and thus incur idle waits in the execution. Hence,the optimal blocksize occurs at the point where the tradeoff between idle waits and communication overhead is overweighed by the float point performance of the individual nodes. Since DGEMM and DTRSM are the dominant operation in the parallel portion of the algorithm, we tried to find the optimal block size for these operations. As you can see in Figure xx and Figure xx,the performance of DGEMM and DTRSM gets better as we increase the block size. When we did the the same test on the parallel algorithm of the Cholesky factorization, we found a little different result(see figure xx). The flat irregular fluctuation of the performance shows that parallel Cholesky factorization is insensitive to the block size .

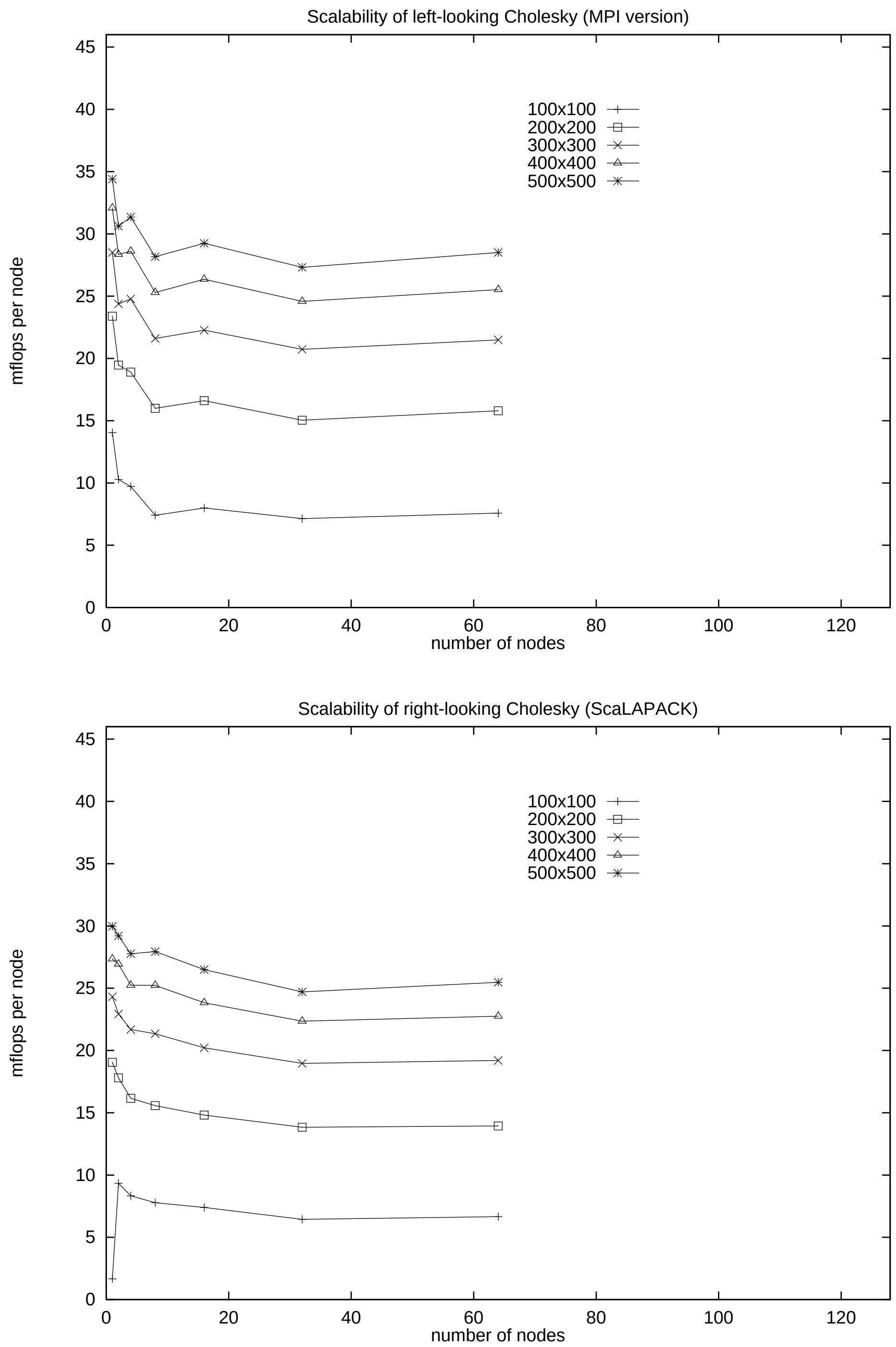

Figure 1: Scalability of our left-looking (*top*) and the ScaLAPACK right-looking (*bottom*) Parallel Cholesky factorization.

7 Conclusion

References

[1] R. van de Geijn and Jerrell Watts. SUMMA:Scalable Universal Matrix Multiplication Algorithm

[2] Jaeyoung Choi, Jack J.Dongarra, L.Susan Ostrouchov, Antoine P. Petitet, David W. Walker and R. Clint Whaley. The Design and Implementation of the Scalapack LU,QR,and Cholesky Factorization Routines. Technical Report LAPACK working note 80, Computer Science Department, University of Tennessee, Knoxville, TN.Sep. 1994.