Parallelizing LU Factorization

Scott Ricketts

December 3, 2006

Abstract

Systems of linear equations can be represented by matrix equations of the form $A\vec{x} = \vec{b}$. LU Factorization is a method for solving systems in this form by transforming the matrix A into a form that makes backwards and forward susbstitution feasible. A common algorithm for LU factorization is Gaussian elimination, which I used for my serial and parallel implementations. I investigated using asynchronous communication to overlap communication and computation for a pipelining effect. I found that this did not provide an improvement in performance. I also compared several 1-dimensional partitioning techniques and found the best performance from the row cyclic layout. I believe that to complete this investigation, it will be necessary to complete a 2-dimensional blocked cyclic layout and to incorporate the BLAS libraries for fast matrix-matrix multiply.

1 Background

For most of my background research I used [3] and [2]. See these references for further detail. Below I will give a high level introduction to LU Factorization.

LU factorization is a method used for solving systems of linear equations. Consider a system with n equations and n unknowns. Each equation will have n coefficients $a_{i0}, a_{i1}, ..., a_{i,n-1}$ and solution b_i:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + \ldots + a_{0,n-1}x_{n-1} &= b_0 \\ \vdots &\quad \vdots \\ a_{i0}x_0 + a_{i1}x_1 + \ldots + a_{i,n-1}x_{n-1} &= b_i \\ \vdots &\quad \vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned}$$

We can then represent this system as matrix-vector multiplication, as in

$$A\vec{x} = \vec{b} \tag{1}$$

where A is the $n \times n$ coefficient matrix ($A(i,j) = a_{ij}$), $\vec{x} = (x_0, x_1, ..., x_{n-1})$ is the variable vector, and $\vec{b} = (b_0, b_1, ..., b_{n-1})$ is the solution vector. Now to solve the system, we need to

find $\vec{x}$. If we had a lower triangular matrix L and an upper triangular matrix U such that $A = LU$, then we would have

$$A\vec{x} = L(U\vec{x}) = L\vec{y} = \vec{b} \tag{2}$$

In this way we could use forward substitution to solve the system $L\vec{y} = \vec{b}$ for $\vec{y}$, and then backward substitution to solve the system $U\vec{x} = \vec{y}$ for x. Forward and backward substitution will have workloads like $\frac{1}{2}n^2$ – 1 operation for the first unknown, 2 for the second, and so on. The process of finding L and U is called LU Factorization.

Gaussian elimination is a common LU algorithm. It uses elementary row operations to eliminate (turn to 0) entries below the main diagonal of A. This leaves an upper triangular matrix. But if instead of putting zeros below the main diagonal we put the multipliers used in the row operations, we get L below the main diagonal and U above it. This is the psuedo code:

for i from 0 to $n-1$ **do**
  for j from $i+1$ to $n-1$ **do**
    $A(j,i) \leftarrow A(j,i)/A(i,i)$
    for k from $i+1$ to $n-1$ **do**
      $A(j,k) = A(j,k) - A(j,i) * A(i,k)$
    end for
  end for
end for

At each step i, the entry $A(i,i)$ is called the pivot, row i the pivot row, and column i the pivot column. The algorithm subtracts a scaled version of the pivot row from each row below it, and then stores all of the scalars in the pivot column. The algorithm is shown graphically in Figure 8. If A_f is the matrix after this algorithm terminates, on successes we will have $A_f = L + U$, where L is below the main diagonal and U is above and includes the main diagonal. This technique has a workload of about $\frac{2}{3}n^3$ flops. We can then pass on L and U quit easily to forward and backward susbtition routines to solve the system in 1.

There will be problems with this algorithm however, if at any step the pivot is zero or very small. If it is zero, then we will have a divide by zero error. On the other hand, if it is close to zero, we may get floating point rounding errors. To evade this problem, we can employ a method called partial pivoting. We will essentially check at every step for the best pivot row – the pivot row with the maximal pivot. The best pivot row is then swapped with the current pivot and the algorithm proceeds.

There is a certain flavor of matrices, however, where pivoting is never necessary: symmetric positive-definite matrices. To be in this class, an $n \times n$ matrix M must satisfy $x^{\mathrm{T}}Mx > 0$, $\forall x \in \mathbb{R}^n$. The following condition is also necessary and sufficient, and is easier to check computationally. The ith principal minor of M is the submatrix consisting of the first i rows and columns of M. If the determinants of all of the principle minors of M are strictly positive, then M is symmetric positive-definite. Cholesky factorization takes advantage of some of the properties of these matrices, especially their symmetry, in order to speed up LU factorization. Its workload is about $\frac{n}{3}$ flops – half that of Gaussian elimination.

2 My Implementations

2.1 Initial Goals of the Project

My proposal was to implement one and two dimensional pipelined parallel version of Gaussian elimination using asynchronous communication. I planned to restrict my input matrices to those that would not require pivoting. This simplifying assumption was motivated by the existence of such matrices in practice. For example, in portfolio optimization, it is common to need to solve a large system of linear equations with coefficients derived from positive asset risks [1]. The system turns out to yield a symmetric positive-definite coefficient matrix.

2.2 Input Matrices

Input matrices we generated using a random number generator. Entries were drawn from a uniform distribution between -10 and 10 and stored as doubles. This scheme produced dense matrices that rarely required pivoting for divide by zero errors. If there were rounding errors, they were not detected by my verification tests using the serial algorithm (discussed below). However, these tests were run on smaller problem sizes than the performance runs.

2.3 Data Layout

There are three classes of 1-dimensional layouts to consider. They can each be oriented by row or column. Since Gaussian elimination is not symmetric with respect to rows and columns, there is not a sense that a row-oriented layout should be equivalent to a column-oriented layout. However, the algorithm, in terms of wits workload, is almost symmetric, and so we would expect the row- and column-oriented layouts to show similar performances. But of course the implemenation details will differ. In the row-oriented 1-dimensional layouts, the algorithms follow this:

for i from 0 to $n-1$ **do**
  if I own any rows below or including i **then**
    if I do not own row i **then**
      Wait for my upward neighbor to send me pivot row i
    end if
    Asynchronously send pivot i to my downward neighbor
    Use the pivot row to eliminate within my block
  end if
end for

In the column-oriented layouts, the scheme is

for i from 0 to $n-1$ **do**
  if I own any columns below or including i **then**
    if I do not own column i **then**
      Wait for my leftward neighbor to send me pivot column i
    end if

Asynchronously send pivot column i to my rightward neighbor
Use the pivot column and my part of the pivot row to eliminate within my block
end if
end for

The first layout is the basic blocked layout – each process gets a block of size $\frac{n}{p}$ as shown in Figure 3.

It is clear that there will be a load balancing problem in both the row and column blocked layouts. If the block size is b, we know that after iteration $b-1$ the first process will be finished, after iteration $2b-1$ the second process will be finished, and so on, giving a load picture like in Figure 7.

A posible solution to the load balancing problem is to assign block sizes smaller than $\frac{n}{p}$ and lay them out cyclicly, as in Figure 4. As we shrink the block size, we would expect the load imbalance problem to disappear.

When the block size is 1, we call the layout row/column cyclic. It would be reasonable to assume that this approach would be the best 1D layout for load balancing.

I ran my performance experiment on Data Star with $n = 3200$. Each run was performed twice, with the minimum run time between the two being reported. I ran each layout on 1, 2, 3, 4, 5, and 6 nodes with 8 tasks per node. The row blocked cyclic and column blocked cyclic layouts were run with block sizes of 40. The plot of this data is shown in Figure 2.

I have mentioned that Guass elimination is not symmetric with respect to row/column orientation, but we would still expect the row and column versions of each of the three layouts to track each other closely. This should give 3 pairs of lines in the plot – one pair for each layout. We almost see that, expect that the column cyclic drops down in performance to track the row/column blocked cyclic pair. This may be attributed to cache effects from the non-optimal layout of data for the column operations. However this effect should appear for the other column-oriented layouts; we do not see this in the data. Additionally, the row blocked and column blocked layouts outperform the row blocked cyclic and column blocked cyclic layouts, even though from the load balancing problem discussed above we would expect the opposite.

From the same experiment I have plotted the efficiency measurements in Figure 5. We see a slight decrease in efficiency with number of processes as we expect with most parallel applications. This behavior is especially evident in the row/column cyclic and row/column blocked cyclic layouts, but not as much in the row/column blocked layouts. The best efficiency is with the row cyclic layout running on 2 nodes (16 processes).

2.4 Communication

Another design decision implied above is the use of asynchronous sends. There is no need to wait for other processes to be ready before starting computation. So at each step, the algorithms are implicitly following

1: If I have any data that another process needs, asynchronously send it.
2: If I need data from another process, receive it.
3: Perform any computation that I can

An interesting question then becomes, how much does performance improve by using this communication protocol as opposed to a synchronous one, where a process must wait at step 1 for the destination process to receive the data. So I created new functions for each of the layouts that used synchronous sends in step 1 above. I compared the performance of the two versions of each parallelization on Valkyrie with 4 processes and $n = 3200$. The results are shown in Figure 1. There is no clear difference.

So then I took a moment to consider the conditions that would be necessary to give an advantage to implementations with asynchronous communication. At step 1 above, if the process uses a synchronous send, it will have to wait only if the destination process is not ready for the data. Of course since the processes are working on blcoks of equal size using the same computation algorithm, there will be litle variation in the workload between each pivot. So in order for a process to gain performance from an asynchronous send, it must have performed some pieve of previous computation faster than the destination process. This would most likely not hapeen in the dense matrix case that I explored, because every entry must be processed in every case. So perhaps if some structure is exploited in a sparse matrix, we may have cases where certain processes speed ahead at certain points in the computation, thereby allowing for performance improvements from asynchronous communication. This leaves an interesting question for futher investigation.

2.5 Correctness Verification

To test the correctness of my code I first verified my serial code by hand on small systems ($n = 3$, $n = 4$). To test it on larger systems I used my handy TI-83 calculator – not the most efficient method, but it seemed convenient at the time. Convinced of the correctness of my serial code, I used it (with partial pivoting) to test the correctness of all parallel implementations thereafter. I wrote a script that compared output of the serial code and parallel code on 1, 2, 3, 4, and 5 processes running on Valkyrie. I varied the problem size and block size parameters accordingly.

3 Improvements

In the results discussed, we see three major problems. The first is the load balancing issue that seems to be solved by the row cyclic layout. The second is that no advantage seems to be gained from locality by increasing block size in the block cyclic layouts. It seems like we should be able to gain something by giving a processor entries that are close together. But the data shows otherwise. I should take a moment to expand upon this issue.

We can make the observation that taken in blocks of size b, the elimination step can be transformed into a matrix-matrix multiply and then a matrix-matrix subtraction. To explain this, let us revisit Figure 8. At step i, we can first perform Gaussian elimination within the block $B = A(i, i)$ to $A(i + b, i + b)$. Now let $T = A(i, i)$ to $A(i + b, n - 1)$ and $L = A(i, i)$ to $A(n - 1, i + b)$. We can then send B rightward and downward so that we can eliminate within T and L. Now what is leftover is the submatrix $M = A(i + b, i + b)$

to $A(n-1, n-1)$. The key idea is that the processed version of M after step $i+b$ should be $M - LT$. Thus we can effectively save computation from step i to step $i+b$, and then perform a block of computation using a call to a fast matrix multiply routine like in BLAS. I did not use this method in my implementations. Leaving this out was damaging to my analysis, because it removed a strong advantage of using a blocked approach as opposed to a row or column cyclic approach.

The third problem is that we do not seem to be gaining any improvement from using asynchronous communication. I mentioned already that investigating sparse matrices might change the analysis here. But perhaps with the BLAS routines there will be an asymmetry between the processes that could help gain performance from asynchronous communication.

Finally, I should mention one more layout that I attempted unsuccessfully to implement. Shown in Figure 9, 2D blocked cyclic seeks to combine the advantages of a blocked approach with the good load balancing of a finely grained cyclic approach. My implementation of this layout provided correct output under only simple parameters and even then ran too slowly to be considered correct. I think that I need to rethink the architecture of my code to help remove the bugs.

References

[1] Thomas Coleman. Speedy portfolio computations by using structure. online article.

[2] Jim Demmel. Cs267: Notes for lecture 14. online lecture notes.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Pearson Education Limited, 2 edition, 2003.