Final Project: Sparse Linear Equation Solvers Using OpenMp or MPI
Team ONE

Daniel Putt (ID:1654588)   Vignesh Jeganathan (ID:1552354)

December 8, 2018

Problem Description:

Problem:

- Solve $Ax = F$ for x, where we know A, and one or more vectors B.
- We are targeting sparse, square input matrices. Specifically, the spike algorithm we have written solves tridiagonal matrices.
- $A = \begin{pmatrix} a_1 & b_1 & . & . & . & . & 0 \\ c_2 & a_2 & b_2 & . & . & . & . \\ . & c_3 & a_3 & b_3 & . & . & . \\ . & . & \ddots & \ddots & \ddots & . & . \\ . & . & . & b_i & a_i & a_i & . \\ . & . & . & . & \ddots & \ddots & \ddots \\ 0 & . & . & . & . & c_n & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_{n-1} \\ f_n \end{pmatrix}$
- Most algorithms for solving linear equations aren't very easily parallelized. The goal of the spike algorithm is to transform the matrix into a problem that can be solved by domain decomposition.
- One of the fastest algorithms for solving a square tridiagonal matrix in serial is the Thomas algorithm.
  - This algorithm 'walks' down the tridiagonal matrix, turning the lower diagonal into zeroes. The other elements in each row, including its respective element of B, are changed accordingly.

- At the bottom row, the row is solved. The algorithm then 'walks' back up the rows, turning the upper diagonal into zeroes.
- The system is solved.

- This algorithm is of order O(N), but it cannot be solved in parallel. Solving each row depends on the row immediately above it, so the domain cannot be decomposed.
- The spike algorithm is of order O(Nlog(N)), but the domain can be decomposed, allowing for the system to be easily solved in parallel.
- The Spike algorithm is described below.

Motivation:

- Banded matrices are prominent in finite difference schemes. Specifically, they are found in materials science codes and fluid dynamics codes.
- A good example of this is related to our second project. The system was approximated by only looking at each particle's closest neighbors.
- Systems we can solve with finite difference schemes are limited by how efficient our algorithms are. Developing new, improved methods of solving tridiagonal systems in parallel may make solving more complex, fine-grained, or larger systems feasible.

SPIKE ALGORITHM

Overview:

Differing from the various LU factorizations, the spike algorithm works to reduce the input matrix before solving. The algorithm only works for banded matrices. The way that this is implemented makes a parallel implementation relatively easy to achieve. Some various versions of the SPIKE algorithm have been created, but the basics are each the same. We explain here, from scratch, the process. We will hit on some limitations on the way.

Definitions:

- Banded Matrix - the main diagonal, and adjacent diagonals of a matrix are filled, while the remainder of the matrix is filled with zeroes.

We will start by describing a single stage of the recursive spike algorithm. We want to take a tridiagonal matrix system Ax = B, and transform it into a new system: Sx = G.

Start:

- Start with (an example) matrix A:
- $\begin{pmatrix} 8 & 8 & 0 & 0 \\ 4 & 2 & 4 & 0 \\ 0 & 1 & 7 & 3 \\ 0 & 0 & 6 & 2 \end{pmatrix} \Rightarrow \left(\begin{array}{cc} \boxed{\begin{matrix} 8 & 8 \\ 4 & 2 \end{matrix}} & \boxed{\begin{matrix} 0 & 0 \\ 4 & 0 \end{matrix}} \\ \boxed{\begin{matrix} 0 & 1 \\ 0 & 0 \end{matrix}} & \boxed{\begin{matrix} 7 & 3 \\ 6 & 2 \end{matrix}} \end{array} \right)$
- We want the 'block diagonal' matrix. This is a matrix composed of the blocks on the main diagonal above. If your system is larger, there will simply be more blocks.
- Explicitly, $D = \begin{pmatrix} 8 & 8 & 0 & 0 \\ 4 & 2 & 0 & 0 \\ 0 & 0 & 7 & 3 \\ 0 & 0 & 6 & 2 \end{pmatrix}$

We find the inverse of D:

It turns out that this is parallelizable. Each square submatrix in D can be inverted without communication with the rest of the matrix values. There is a limition. If a submatrix is singular, there is no inverse to the submatrix. This does not mean that the whole system doesn't have a unique solution. We solve this problem with 'diagonal boosting', which involves adding a small value precision zero to the first element. We will come back to that later.

- $D^{-1} = \begin{pmatrix} -0.125 & 0.5 & 0 & 0 \\ 0.25 & -0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0.75 \\ 0 & 0 & 1.5 & -1.75 \end{pmatrix}$

Getting S matrix:

Left Multiplying both sides of Ax = B by D^{-1} gets us the spike matrix S, and the new system: $Sx = G$

- $\begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & -0.5 & 1 & 0 \\ 0 & 1.5 & 0 & 1 \end{pmatrix} \Rightarrow \left(\begin{array}{cc} \mathrm{I} & \boxed{\begin{matrix} 2 & 0 \\ -2 & 0 \end{matrix}} \\ \boxed{\begin{matrix} 0 & -0.5 \\ 0 & 1.5 \end{matrix}} & \mathrm{I} \end{array} \right)$

We are left with a new, reduced problem to solve

- Our problem is now $Sx = D^{-1}B$
- In this example, if we find the inverse of this new matrix S, and multiply it by our new G, we find our answer vector, x
- $x = S^{-1}D^{-1}B$

Recursive Spike:

If we look at a larger tridiagonal system, we can see how the spike algorithm can be applied recursively.

An example system:

$$A = \begin{pmatrix} a & b & 0 & 0 & 0 & 0 & 0 & 0 \\ c & d & v & 0 & 0 & 0 & 0 & 0 \\ 0 & w & a & b & 0 & 0 & 0 & 0 \\ 0 & 0 & c & d & v & 0 & 0 & 0 \\ 0 & 0 & 0 & w & a & b & 0 & 0 \\ 0 & 0 & 0 & 0 & c & d & v & 0 \\ 0 & 0 & 0 & 0 & 0 & w & a & b \\ 0 & 0 & 0 & 0 & 0 & 0 & c & d \end{pmatrix}$$

$$D = \begin{pmatrix} a & b & 0 & 0 & 0 & 0 & 0 & 0 \\ c & d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a & b & 0 & 0 & 0 & 0 \\ 0 & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & 0 \\ 0 & 0 & 0 & 0 & c & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & b \\ 0 & 0 & 0 & 0 & 0 & 0 & c & d \end{pmatrix}$$

$$D^{-1} = \begin{pmatrix} d/z & -b/z & 0 & 0 & 0 & 0 & 0 & 0 \\ -c/z & a/z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d/z & -b/z & 0 & 0 & 0 & 0 \\ 0 & 0 & -c/z & a/z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d/z & -b/z & 0 & 0 \\ 0 & 0 & 0 & 0 & c/z & a/z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d/z & -b/z \\ 0 & 0 & 0 & 0 & 0 & 0 & -c/z & a/z \end{pmatrix}$$

$$D^{-1}A = S = \begin{pmatrix}
1 & 0 & -bv/z & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & av/z & 0 & 0 & 0 & 0 & 0 \\
0 & dw/z & 1 & 0 & -bv/z & 0 & 0 & 0 \\
0 & -cw/z & 0 & 1 & av/z & 0 & 0 & 0 \\
0 & 0 & 0 & dw/z & 1 & 0 & -bv/z & 0 \\
0 & 0 & 0 & -cw/z & 0 & 1 & av/z & 0 \\
0 & 0 & 0 & 0 & 0 & dw/z & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & -cw/z & 0 & 1
\end{pmatrix}$$

Another Banded Matrix:

You can see here that this resultant S matrix is also a banded matrix. This means that it can also be solved with the spike algorithm.

$$S = \left(\begin{array}{cc}
\boxed{\begin{matrix} 1 & 0 & \text{-bv/z} & 0 \\ 0 & 1 & \text{av/z} & 0 \\ 0 & \text{dw/z} & 1 & 0 \\ 0 & \text{-cw/z} & 0 & 1 \end{matrix}} & \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \text{-bv/z} & 0 & 0 & 0 \\ \text{av/z} & 0 & 0 & 0 \end{matrix} \\
\begin{matrix} 0 & 0 & 0 & \text{dw/z} \\ 0 & 0 & 0 & \text{-cw/z} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} & \boxed{\begin{matrix} 1 & 0 & \text{-bv/z} & 0 \\ 0 & 1 & \text{av/z} & 0 \\ 0 & \text{dw/z} & 1 & 0 \\ 0 & \text{-cw/z} & 0 & 1 \end{matrix}}
\end{array} \right)$$

We can split this domain into two partitions, each of which can be solved in parallel.
Our problem is then something like $S_1 = D_1^{-1} D_0^{-1} B$
We can perform the spike algorithm repeatedly until we get two single spikes:

$$S_1 = \begin{pmatrix}
1 & 0 & 0 & 0 & a & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & b & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & c & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & d & 0 & 0 & 0 \\
0 & 0 & 0 & e & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & f & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & g & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & h & 0 & 0 & 0 & 1
\end{pmatrix}$$

Taking the inverse of this matrix, S^{-1}, and multiplying by our newest $D_i^{-1} D_{i-1}^{-1} ... D_0^{-1} B$. This gives us our answer vector, x.

LU Decomposition:

The spike blocks, since they have a consistent structure, can be solved in a more efficient way. We don't have to store the inverse matrices either, since we only want the inverse to create the new spikes and update our B/G vector. I used LU decomposition at first, but quickly noticed the inverses have similar formats, regardless of size:

$$\begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & c & 1 & 0 \\ 0 & d & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & \frac{ac}{1-bc} & \frac{-a}{1-bc} & 0 \\ 0 & \frac{1}{1-bc} & \frac{-b}{1-bc} & 0 \\ 0 & \frac{-c}{1-bc} & \frac{1}{1-bc} & 0 \\ 0 & \frac{-d}{1-bc} & \frac{bd}{1-bc} & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & b & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & c & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & d & 0 & 0 & 0 \\ 0 & 0 & 0 & e & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & f & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & g & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & h & 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & ae/(1-de) & -a/(1-de) & 0 & 0 & 0 \\ 0 & 1 & 0 & be/(1-de) & -b/(1-de) & 0 & 0 & 0 \\ 0 & 0 & 1 & ce/(1-de) & -c/(1-de) & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/(1-de) & -d/(1-de) & 0 & 0 & 0 \\ 0 & 0 & 0 & -e/(1-de) & 1/(1-de) & 0 & 0 & 0 \\ 0 & 0 & 0 & -f/(1-de) & fd/(1-de) & 1 & 0 & 0 \\ 0 & 0 & 0 & -g/(1-de) & gd/(1-de) & 0 & 1 & 0 \\ 0 & 0 & 0 & -h/(1-de) & hd/(1-de) & 0 & 0 & 1 \end{pmatrix}$$

My implementation uses this method to calculate the new elements of the new spikes, G vector, so that the inverse matrix doesn't have to be stored at each level.

Openmp Parallelization and Results Discussion

For Openmp, we utilize shared memory, and create teams of two threads as we repeatedly partition the system. When the recursion hits the 'bottom' (2x2 matrix size), the 2x2 is solved. The program then starts working back up, performing the calculations on the spikes and B/G vector as described above.

- We wrote two versions of the code for Openmp.
- First:
  - One creates all of the threads at the beginning.
  - This version uses "tasks" pragmas, with "taskwait"s present, to make sure each spike is solved before the threads move back up to the previous level. Tasks can be written recursively. The taskwait pragmas only apply to the siblings within the parent task.
- Second:
  - Creates teams of threads as the program 'partitions' our input matrix.
  - This is the Openmp we learned this semester. We start with no threads created. We use parallel regions, with the sections construct at each level of recursion, and for each block. This can also result in difficulties. The same synchronization problems exist.
- Both versions have merits, and their own difficulties.
- Since each thread/processor is reading from the same shared memory, there are a lot of memory read/write conflicts to deal with. We were not able to fully solve these issues, as this was a deceptively complicated project for us. We know where some of our issues lie, and where to proceed from here to improve on our algorithm.
- We can stop creating/ using new threads at any depth we want. There comes a point where the work performed by the thread takes less time than the overhead involved in preparing the thread and data. Further exploration here is needed.
- The partitions only read from A at the bottom level of recursion. The partitions also both read and write to and from the shared W and V spike vectors, almost all at the same time. It would be worth looking into adapting the program to have each local partition build its local spikes, in its own personal array, and then have the functions piece together the results when the subblocks are solved.

- The actual solving of the inverse matrix at each level could still be optimized more. I create local arrays to hold a lot of information, so I can update V,W, and G. This means a lot of copy operations are performed.

- The calculations of the new spikes and the G vector can also be performed in parallel. This is also not fully implemented, and bears further exploration and implementation.

Unfortunately, our OpenMp algorithm wasn't able to reach the efficiency of the serial Thomas algorithm we wrote. With the above problems worked through, and tuning to the specific system, we are confident that the code will surpass the serial version in efficiency, particularly with larger systems.

Since we did not achieve the same efficiency as the serial Thomas algorithm, we did not attempt to compare our results with other available sparse matrix solvers. It would be interesting to compare with those solvers if the changes required with the code are implemented.

Mpi and Openmp?

Mpi isn't efficient beyond a certain level of recursion. The passing of information between nodes is relatively slow. When the partition can be calculated faster than the data is communicated between the nodes, passing the information to a new processor isn't worth it. With OpenMp, there is a limitation of memory, as we only use a single node. Assuming the OpenMp algorithm is tuned to be efficient, this means that we could use a combination MPI/OpenMp program that could be faster than either method, with parallelization passing off from MPI to OpenMp at a predetermined level of recursion.