Alexander Hayman

6.338 Final Project
Parallelized KLU Factorization

Introduction:

Electrical engineers use circuit simulation software extensively to help with design verification. My thesis research involves the development of a high efficiency solar inverter. The design of this particular circuit needs to be verified at several dozen operating points, and each simulation takes about ten minutes. The software being used is called LTspice IV, which is freeware provided by Linear Technologies. The new version was released in January, and boasts that it "features multi-threaded solvers designed to better utilize current multicore processors." Cadence, a company that sells circuit design and simulation software, also recently unveiled a new parallelized simulator.

The aim of this project was to learn more about how the software works, and how it can be parallelized. The circuits that are being simulated involve components that store energy and nonlinear switches, which means a transient analysis is necessary. A transient analysis is performed by using numeric integration methods (trapezoidal rule) to keep track of the state of various components as a function of time. This requires that a nonlinear system of equations be solved at every time step. The duration between time steps will vary to boost simulation speed and accuracy.

The Newton method is used to solve the nonlinear system at each time step. The Newton method involves making a guess at the operating point, linearizing the system around that point, solving the system, and then readjusting the operating point, relinearizing, and resolving until reasonable convergence occurs. This means that for a given time step, multiple linear systems of equations must be solved.

LU Factorization:

At each time step, an LU factorization of the circuit description matrix is performed, since Newton's method involves solving $Ax=b$ for varying b until convergence. The LU factorization turns out to be the most time consuming part of a circuit simulation, and significant research has been dedicated towards optimizing the speed of LU factorization algorithms. Timothy Davis at the University of Florida developed the KLU algorithm, an LU factorization algorithm specifically optimized for circuit description matrices, which are typically highly sparse, and have a mostly zero-free diagonal.

The goal of this project was to study KLU, implement it serially, and explore ways to parallelize aspects of the algorithm. KLU is published in a package called SuiteSparse, which I installed

successfully, and used as a baseline for comparison with parallel improvements that I hoped to make.

The KLU algorithm can be described as a series of preprocessing steps, followed by the actual factorization step. The performance is evaluated as the sum of the time it takes to perform the preprocessing, factorizing, and solving of the sparse linear system.

The preprocessing steps involve permuting the matrix to block upper triangular form (BTF) and using an Approximate Minimum Degree (AMD) algorithm to reduce the order of each block. The preprocessed matrix is then LU factorized using the Gilbert/Peierls' left looking algorithm with partial pivoting, and the matrix is solved

The first step in the KLU algorithm is to permute the matrix to block upper triangular form (BTF). This is followed by using the Approximate Minimum Degree (AMD) algorithm to reduce the fill in each block. After these preprocessing steps, the factorization is performed using the Gilbert/Peierls' left looking algorithm with partial pivoting.

BTF:

My stated goal was to parallelize KLU, an algorithm that is made up of several discrete steps. Because the steps are sequential, I decided to focus on parallelizing each step, starting with BTF. BTF itself is an algorithm that is broken down into two discrete steps. The first step is to compute a column permutation to minimize the number of zeros on the diagonal. This is referred to as a maximum transversal. The second step computes a permutation of the matrix to upper block triangular form. The algorithms for performing these two steps are derived from graph theory.

Maximum Transversal:

The maximum transversal problem can be expressed as trying to find a maximum match of a undirected bipartite graph. In other words, we are looking for a mapping from an old location to a new location for each column. Physically, the mapping is a vector of column indices.

The maximum transversal algorithm adopted by Timothy Davis is based on ACM Algorithm 575. This algorithm is designed to find a maximum match as quickly as possible. The search starts off by permuting the first column (*0*) to a new index (*i*) based on the location of the first nonzero row (*i*). The same is done for a second column, ensuring that each column is assigned to one (and only one) index. The algorithm is more reflective of a depth first search through a directed graph where a column (*i*) is a node, and a directed edge is drawn from every nonzero row (*j*) in the column (*i*) to every new column (*j*).

Strong Components:

After finding the maximum transversal, the matrix must be permuted to upper block triangular form. This form is described as a matrix with blocks along the diagonal, and all entries below the diagonal being zero. Finding the upper block triangular permutation boils down to another graph problem that is quickly and efficiently solved with a depth first search. A graph can be drawn where each node (i) represents a diagonal at (i,i). Edges are drawn from node i to node j, given that the element at (i,j) is nonzero. The necessary permutation is discovered by finding the strong components of the graph.

A subgraph is strongly connected if a path can be found from any node to any other node. A strong component of a directed graph is a strongly connected subgraph that cannot be expanded. Each strong component of a graph is a block of the upper block triangular form.

The algorithm used by Timothy Davis is based on ACM Algorithm 529. Generally it involves a depth first search to find a cycle, which is by definition strongly connected. All nodes in the cycle are then represented as a single node, and the depth first search continues to try to find a bigger cycle.

AMD and Gilbert/Peierls:

Because of time constraints, the algorithms for finding an approximate minimum degree for each block and for performing the LU factorization were not fully explored. It was clear from documentation and preliminary studies of the algorithms that depth first search is used over and over again, both in AMD as a postprocessing step, and during the LU factorization step. The algorithms are based on finding a match or a path in a graph description of a sparse matrix, and since we are optimizing for speed to a solution and efficiency, depth first search is used.

Parallelization:

It is theoretically possible to speed up KLU by taking advantage of parallel processors. We could describe KLU more simply as a sequence of depth first searches. The speed of a depth first search can be improved by using multiple processors to search a tree at the same time, as long as we can easily ensure that the different processors search different spaces of the search tree at any given time. Parallel DFS was explored by Jon Freeman at the University of Pennsylvania.

With the parallel programming tools (Star-P) available to us, ensuring that each processor is searching a different space, or setting different search rules is not possible (i.e. “always pick the first branch” or “always pick the last branch”). It is possible to achieve an approximation of this by randomly selecting which branch to iterate through or randomly selecting a uniform branching rule.

In the algorithm, we sacrifice some time to generate random permutations of which branch to

iterate through, in the hope that if 8 or 16 processors are doing the same thing, a solution will be found more quickly on average than a single processor using a single branching rule. We can only expect a win with a randomized algorithm if we know that the depth first search involves a fair amount of backtracking. Otherwise, the extra memory and extra assignments are not worth the hassle.

The speed of a depth first search can be improved by using multiple processors and ensuring that the processors are searching different spaces of the tree. When a solution is found, the solution should be broadcast to all the processors and they should move on to the next step (in this case, finding the block upper triangular permutation).

This approach to depth first search is similar to a Las Vegas algorithm, where a solution to a problem is guaranteed, but the time to reach the solution takes a varying amount of time (whose distribution is known). This is as opposed to a Monte Carlo algorithm, where the time is known, but the function will give a right or wrong answer with a known probability.

Implementation and Analysis:

I implemented a randomized version of the maximum transversal algorithm by making modifications to the Matlab/C code published by Timothy Davis. I tested the speed of the original and randomized maximum transversal algorithm on circuit simulation matrices available from Davis' website. The randomized algorithm was sometimes faster for a small percentage of the matrices. In general, the randomized algorithm was anywhere from 2 to 10 times slower. This indicates that the maximum transversal search does not involve significant backtracking, and as a result, is probably not the best function to try to parallelize. It is possible that one or several of the depth first search speeds for BTF, AMD post-processing, and the LU factorization could be improved by using a parallel Las Vegas approach.

Conclusions:

My lack of experience with graph theory, linear algebra, and algorithms slowed my progress on this project. I learned a lot about how circuit simulators work, how sparse matrices are analyzed and manipulated so that they can be solved quickly, and how the associated algorithms take advantage of graph theory. Unfortunately, in this case, parallelization of the maximum transversal did not present an improvement in performance. Obviously, further research can be done to explore whether parallelization of the other depth first searches built into the KLU algorithm will help.

Bibliography:

Davis, Timothy A and Ekanathan Palamadai Natarajan. "Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems." ACM Transactions on Mathematical Software. 2007.

Duff, I. S. "On Algorithms for Obtaining a Maximum Transversal." ACM Transactions on Mathematical Software, Vol 7, No 3, September 1981.

Duff, I. S. and J. K. Reid. "An Implementation of Tarjan's Algorithm for the Block Triangularization of a Matrix." ACM Transactions on Mathematical Software, Vol 4, No. 2, June 1978.

Freeman, Jon. "Parallel Algorithms for Depth-First Search." University of Pennsylvania, 1991.

Tarjan, Robert. "Depth-first Search and Linear Graph Algorithms." SIAM J. Comput. Vol. 1,, No. 2, June 1972.

Volpi, Leonardo. "Note of Numeric Calculus." May 14, 2004.