Solving Systems of Equations with Multi-core Computers

Tyler Helmus*, Caleb Reese*, and Dr. Matthew Heun

Engineering Department, Calvin College, Grand Rapids, Michigan, USA

Phone: (616) 526-6663

Fax: (616) 526-6501

Email: mkh2@calvin.edu

Abstract

As simulations of physical systems become increasingly detailed, they require solving increasingly large systems of equations. The goal of this research was to create a fast numerical equation solver for large systems of equations by utilizing efficient decomposition of the system of equations and parallel processing. Our numerical equation solver utilizes an occurrence matrix to decompose equation sets into blocks of equations and layers of blocks amenable to parallel processing. The performance of our algorithms depends on many factors, including the structure of the equation set, the number of available processors, and the overhead of thread management. Experimental results were obtained using Java on several different computers and operating systems, including a 16-core Linux machine. With 2000 equations in 500 blocks with 16 processors, we show that the multi-core approach decreases the solve time by a factor of 11.2, or 70% of the theoretical speed increase, compared to single-core techniques.

Keywords: Numerical methods, parallel processing, equation solving, algorithms

*These authors contributed equally to this work and are listed alphabetically.

1. Introduction

Over the past few decades, researchers have looked for ways to further increase the speed of solving large systems of equations with numerical algorithms such as the Newton-Raphson method. In the literature, greater computational speed has been achieved in two ways: by decomposing large systems of equations into smaller solvable subsets or by using more powerful computational methods to solve the subsets. Recent advances, namely low-cost, multi-core processors, provide new opportunities for greater computational speed when solving large-scale systems of equations. We obtain another increase in computational speed by utilizing parallel processing technology on multi-core computers. When decomposing a large system of equations into several smaller subsets, independent subsets can potentially be solved simultaneously on parallel processors available in most computing devices today. This approach greatly increases the speed at which these systems of equations are solved and, given that most personal computers today are equipped with at least two processors, it is a speed advantage that is accessible to almost all users. We developed two algorithms: a linear algorithm for decomposition of a large-scale system of equations and determination of an efficient linear solution path through the system of equations, and a layering algorithm that prepares the system of equations for parallel processing. This paper focuses on the design and implementation of the algorithms and the use of parallel processing on large-scale systems of equations. We present results demonstrating that each additional processor offers a speed advantage.

2. Literature Review

Himmelblau is the forerunner of the topics addressed by this paper. His work used the idea of known and unknown variables to reduce computational time. He assigned unknown variables to specific equations based on solution dependencies to reduce complexity in calculation, which is an important part of the solution process in our methods. Himmelblau first uses Boolean matrices to determine the dependencies based on graph vertices and edges [1, 2]. Although our methods also use Boolean occurrence matrices, we use a different algorithm to directly decompose a system into smaller sub-sets of equations.

Building off the work done by Himmelblau, Ramirez and Vestal developed algorithms that decompose systems of equations by determining an optimal set of design variables [3]. They use a pair of algorithms to determine an optimal solution path for a large-scale system of equations with more unknown variables than mathematical relations among the variables. Given such a system of equations, the algorithms determine sets of specified design variables and unknown variables which will minimize the amount of simultaneous calculations needed, the idea being that selection of these "free" variables amounts to "designing" the system. For the purposes of our research, we are more interested in finding the most efficient solution path given a specified set of known and unknown variables, not determining the optimal combination of variables that would constitute a "design" problem.

Duff and Reid published research on Tarjan's algorithm and the algorithm of Sargent and Westerberg, which uses graph theory to look through vertices in a directed graph and search for edges between strongly connected vertices [4, 5, 6]. The algorithm developed below uses directed acyclic graphs to place blocks to be solved in layers, but Duff and Reid's algorithm did not.

Sridhar et al used occurrence matrices and an algorithm similar to ours to decompose large systems of equations [7]. The Sridhar algorithm optimizes an under-constrained system for a user-specified set of design variables. Then the algorithm checks for redundant constraints and determines an efficient solution path through the equations. Sridhar's algorithm also performs decomposition after each design variable is added to the occurrence matrix. Although Sridhar's approach to finding an efficient solution path is very similar to ours, our blocking algorithm does not focus on under-constrained systems or the specification of design variables.

Alexander and Blackketter further expands upon the work done by Himmelblau and Ramirez and Vestel, by using an occurrence matrix and a technique called Logical Equation Set Decomposition (LESD) to decompose larger sets of equations to smaller subsets [8]. The LESD algorithm that Alexander and Blackketter developed is similar to Ramirez and Vestel's algorithm in that it determines an optimal solution path. However, Alexander and Blackketter's algorithm takes a set of known and unknown variables and produces the optimal solution path for these variables, whereas Ramirez and Vestel's algorithm takes a system of equations and finds both an

optimal solution path for the set and the corresponding optimal combination of known and unknown variables for the path. The LESD algorithm operates in much the same way that our blocking algorithm does. However, Alexander and Blackketter do not parallelize the resulting solution path.

Our linear algorithm is similar to Blackketter LESD algorithm and is used to find the optimal solution path given a large-scale system of equations with known and unknown variables. Our layering algorithm extends the linear algorithm to create a solution path amenable to modern multiprocessor CPUs.

There has also been recent work on automatic parallelization of pointer-based dynamic data structures, specifically in Java, by Chan and Abdelrahman [9]. The work exploits parallelism by creating new asynchronous threads of execution for method invocations in a Java program.

3. Terms

This section defines the terminology used throughout this paper.

System of equations

A *system of equations* is the original group of equations that describes a problem to be solved. The system of equations typically "models" a real-life system, such as a chemical process.

Equation block

An *equation block* is a group of equations, extracted from the larger system of equations, which can be solved independently from other equation blocks. Equation blocks contain no other solvable sub-blocks. The order in which equation blocks must be solved is determined by the solution path. The number of equation blocks can never be greater than the number of equations.

Occurrence matrix

An *occurrence matrix* is an $m \times n$ boolean matrix of 1s and 0s used to represent relationships among variables and equations within an equation set: it shows which variables appear in each equation. The rows of the occurrence matrix represent equations and the columns represent variables. A "1" in the mth row and the nth column of the occurrence matrix indicates that equation m contains variable n. If equation m contains variable n once or multiple times, the mn

location in the occurrence matrix is set to 1, otherwise the *mn* location is set to “0.” See Figure 1 for a sample occurrence matrix.

E0: $a + b = 7$

E1: $b - a = 3$

E2: $c = a + d$

E3: $d = bc$

E4: $f = a/b$

E5: $g^2 = b^2 - a$

E6: $ea = f^b$

e\v	a	b	c	d	e	f	g	ω(e)
E0	1	1	0	0	0	0	0	2
E1	1	1	0	0	0	0	0	2
E2	1	0	1	1	0	0	0	3
E3	0	1	1	1	0	0	0	3
E4	1	1	0	0	0	1	0	3
E5	1	1	0	0	0	0	1	3
E6	1	1	0	0	1	1	0	4
ω(v)	6	6	2	2	1	2	1	

Figure 1. Example system of equations and corresponding occurrence matrix.

Variable frequency, ω(v)

The *variable frequency*, ω(v), represents the number of independent equations that contain a given variable. ω(v) is found by summing down a column of an occurrence matrix. See Figure 1.

Equation frequency, ω(e)

The *equation frequency,* ω(e), represents the number of variables that appear in each equation. ω(e) is found by summing across a row of the occurrence matrix. See Figure 1.

Solvable

A group of equations (an equation set, an equation subset, or an equation block) is considered to be *solvable* if both (a) the group of equations contains the same number of equations as variables, i.e., if the number of equations for which $\omega(e) > 0$ is equal to the number of variables for which $\omega(v) > 0$ and (b) the equations are algebraically independent.

Subsets

A *subset* is a group of equations that are related by common unknown variables. Subsets may contain smaller equation blocks. A subset may or may not be itself solvable, and further

decomposition may be required. See Figure 2 for an example of subsets containing smaller equation blocks.

	a	b	c	d	e	f	g	h	i	j	k	l	ω(e)
0	1	1	1	0	0	0	0	0	0	0	0	0	3
1	1	1	1	0	0	0	0	0	0	0	0	0	3
2	1	1	1	0	0	0	0	0	0	0	0	0	3
3	0	0	1	1	1	0	0	0	0	0	0	0	3
4	0	0	0	1	1	1	0	0	0	0	0	0	3
5	0	0	0	1	1	1	0	0	0	0	0	0	3
6	0	0	0	0	0	0	1	1	1	0	0	0	3
7	0	0	0	0	0	0	1	1	1	0	0	0	3
8	0	0	0	0	0	0	1	1	1	0	0	0	3
9	0	0	0	0	0	0	0	0	1	1	0	1	3
10	0	0	0	0	0	0	0	0	0	1	1	1	3
11	0	0	0	0	0	0	0	0	0	1	1	1	3
ω(v)	3	3	4	3	3	2	3	3	4	3	2	3	

Figure 2. Sample occurrence matrix with two subsets (dashed lines), each containing four 3x3 equation blocks.

Solution path

A *solution path* is the order in which equation blocks from a larger system of equations are solved. The solution path identifies which variables are solved by which block. The solution path is created before any arithmetic computations are executed. The solution path is determined independently of the arithmetic operations used within the equations of the system. The solution path is independent of the type of equations (linear or non-linear). The solution path is independent of how many times a variable appears within an individual equation.

Layers

A *layer* consists of a number of equation blocks that can be solved independently of each other, and simultaneously, at a given step along the solution path. Being independent of each other, the equation blocks in a layer are amenable to parallel processing, as each of the blocks in a layer can be solved simultaneously on a separate processor. The number of layers can never be greater than the number of blocks. See Figure 3 for an illustration of layering.

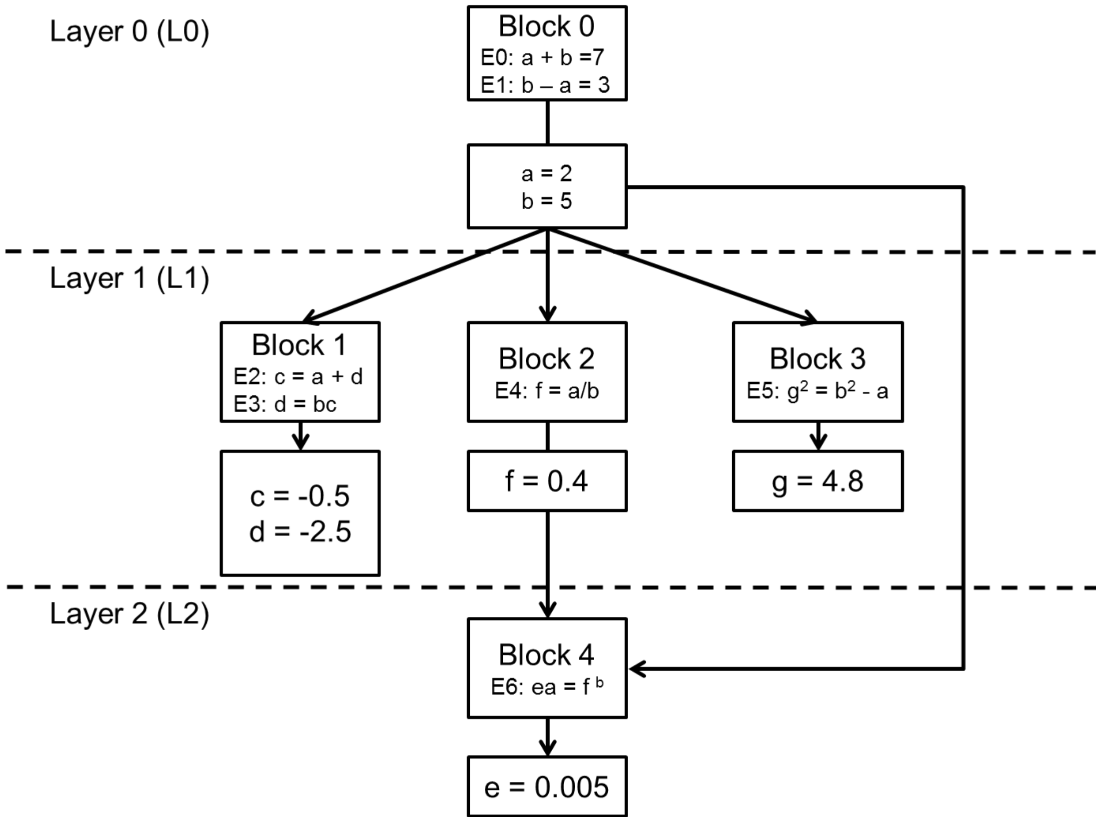

Figure 3. Layers of equation blocks from Figure 1.

Blockiness

The *blockiness* (b) of a system of equations is a normalized parameter that describes the relationship between the number of blocks (B) and the number of equations (E) within a system of equations. The blockiness of a system of equations is given by equation 1. The value of b ranges from 0.0 to 1.0 and is undefined when $E = 1$.

$$b = \frac{B-1}{E-1} \tag{1}$$

High values of blockiness correspond to higher numbers of equation blocks and smaller block sizes. Small values of blockiness correspond to lower numbers of equation blocks and larger block sizes. Systems of equations with high values of blockiness are more likely to achieve speed increases when a numerical solver takes advantage of the block structure of an equation set.

Layeredness

The *layeredness* (l) of a system of equations is another normalized parameter that describes the relationship between the number of layers (L) and the number of blocks (B) within a system of equations. The layeredness of a system of equations is given by equation 2.

$$l = 1 - \frac{L-1}{B-1} \tag{2}$$

High values of layeredness correspond to more blocks per layer. Low values of layeredness correspond to fewer blocks per layer. High values of layeredness are more likely to achieve speed increases due to parallel processing. The value of l ranges from 0.0 to 1.0 and is undefined when $B = 1$.

4. Methods

4.1 Introduction to Algorithms

We use two methods to decompose equation sets into an efficient solution path: The Linear algorithm and the Layering algorithm. The Linear algorithm is composed of two sub-algorithms called the Blocking algorithm and the Subsetting algorithm.

4.2 Linear Algorithm (Blocking)

The Blocking algorithm assesses equation dependencies and manipulates the occurrence matrix to create an ordered list of blocks that can be solved independently, in sequence. The Linear Algorithm is a connected component algorithm that takes advantage of properties of occurrence matrices to determine the smallest connected components possible. Each block has fewer equations than the equation set. The Blocking algorithm is illustrated in Figure 3.

The Blocking algorithm begins by assessing whether the equation set has the same number of variables as unknowns. If the number of variables for which $\omega(v) > 0$ equals the number of equations for which $\omega(e) > 0$, the equation set meets a necessary condition for solvability, and the algorithm proceeds.

Next, the Blocking algorithm looks for two special cases: equations for which $\omega(e) = 1$ and variables for which $\omega(v) = 1$. These cases can be thought of as a special kind of min-cut algorithm which only cut single connections to equations or variables.

Equations for which $\omega(e) = 1$ are assignment statements of the form a = 2. The Blocking algorithm identifies these equations and removes both the equation and its variable from the occurrence matrix by setting the equation's row and the corresponding variable's column in the occurrence matrix to zeroes. Removing a variable from the occurrence matrix affects all equations that contain that variable, therefore, the occurrence matrix is rechecked for any additional $\omega(e) = 1$ equations that may have been created. This process is repeated until no $\omega(e) = 1$ equations remain in the occurrence matrix (see Figure 4-1.) Each assignment statement is contained in its own block, and the blocks are pushed onto a First In First Out (FIFO) queue.

Any equation that contains a variable for which $\omega(v) = 1$ can also be removed from the occurrence matrix, because such equations *must* be used to solve for the $\omega(v) = 1$ variable. (see Figure 4-2.) Equations that contain a $\omega(v) = 1$ variable are put in their own block, and the blocks are pushed onto a Last In First Out (LIFO) stack. Because removal of $\omega(v) = 1$ equations may have created additional $\omega(v) = 1$ equations within the occurrence matrix, the occurrence matrix is rechecked for $\omega(v) = 1$ variables. The process is repeated until no $\omega(v) = 1$ variables remain in the occurrence matrix.

Next, the occurrence matrix is checked to see if any equations remain (i.e., if there are any rows in the occurrence matrix for which $\omega(e) > 0$). If some equations remain, a conditioning stage is entered in Figure 4-3 to temporarily remove equations that need not be considered at the moment, because they contain more variables than a block size for which we will soon search. The desired block size is based on the variable *iCalc*, which is set to the minimum $\omega(e)$ value in the occurrence matrix. The algorithm temporarily removes remaining equations with $\omega(e)$ greater than *iCalc*, thereby allowing the algorithm to check smaller, easier-to-solve blocks first and larger, harder-to-solve blocks later (Figure 4-3). The Blocking algorithm also temporarily removes additional equations if the temporary removal of equations with $\omega(e) > iCalc$ causes any remaining equations to contain a variable with $\omega(v) = 1$ (Figure 4-3). If the above removals result

in no remaining equations, all temporarily removed equations are restored, *iCalc* is incremented by 1, and the process of Figure 4-3 is repeated. The process of Figure 4-4 checks if the process in Figure 4-3 created an independent equation block (number of remaining equations $<$ *iCalc* +2 and the remaining equations are solvable, i.e. have the same number of equations and variables). If so, the remaining equations in the occurrence matrix are permanently removed as a block to the FIFO stack. Thereafter, all temporarily removed equations are restored and loop is restarted at the top. If the process in Figure 4-3 does not create an independent equation block, the Subsetting algorithm (described below) identifies subsets in the remaining equations.

Figure 4-4 describes a step to check whether the remaining equations can be further decomposed to smaller blocks. If the remaining equations are solvable and the number of equations in the occurrence matrix is less than *iCalc* + 2, the equations can be removed to the FIFO queue as a block. The number *iCalc* + 2 is the cutoff point because the smallest additional block that could be found within a the occurrence matrix after the ω(e) and ω(v) removals must have at least 2 equations. Also, the *iCalc*-sized block will be found in that subset if it is solvable, therefore the sum of these two block sizes is the cutoff point.

Figure 4-5 is the final stage in creating an independent equation block. Each subset found by the Subsetting algorithm is evaluated for solvability. If the subset is solvable and the number of equations in the subset is less than *iCalc* +2, the subset can be removed to the FIFO queue as a block, otherwise the algorithm searches all possible combinations of equations to find smaller solvable equation blocks. If any solvable blocks are found, they are removed to the FIFO queue. If no smaller equation blocks are found, the entire subset is permanently removed as a block to the FIFO queue.

After all subsets have been evaluated by section Figure 4-5, all temporarily removed equations are restored, and we return to the top of the loop. When all equations have been removed, the while loop exits. A linear solution path through the system of equations is constructed by popping blocks first from the FIFO queue and last from the LIFO stack.

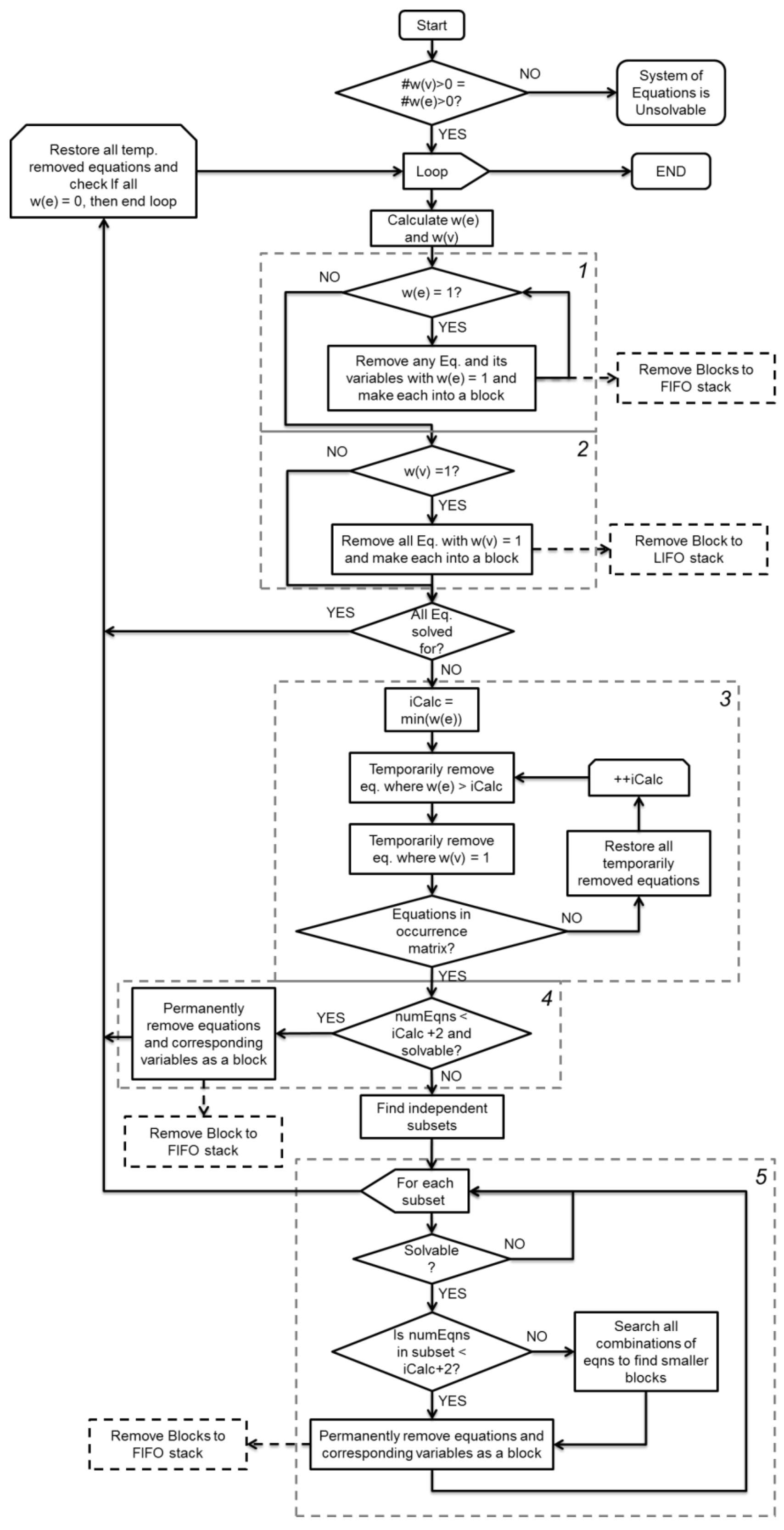

Figure 4. Blocking algorithm flow diagram.

4.3 Linear Algorithm (Subsetting)

If the process in Figure 4-4 cannot find an independent equation block, the occurrence matrix is sent to the Subsetting algorithm, which finds equation subsets for the current *iCalc* value (Figure 5). To find independent subsets, the Subsetting algorithm looks at each equation and checks whether it is contained in the subset already. (At the beginning of this loop, no equations are in the subset.) If the equation is not already in the subset, it is added to the subset, its variables are added to the subset's list of variables, and all equations with variables in common with the subset are added to the subset. Each time an equation is added to the subset, the process begins again until no additional equations are added to the subset. When all related equations are accounted for in the subset, the subset is then added to a list of subsets. Some systems of equations contain only one subset. The Subsetting algorithm continues until every equation has been added to a subset. The Subsetting algorithm is a great example of a straight forward connected component algorithm.

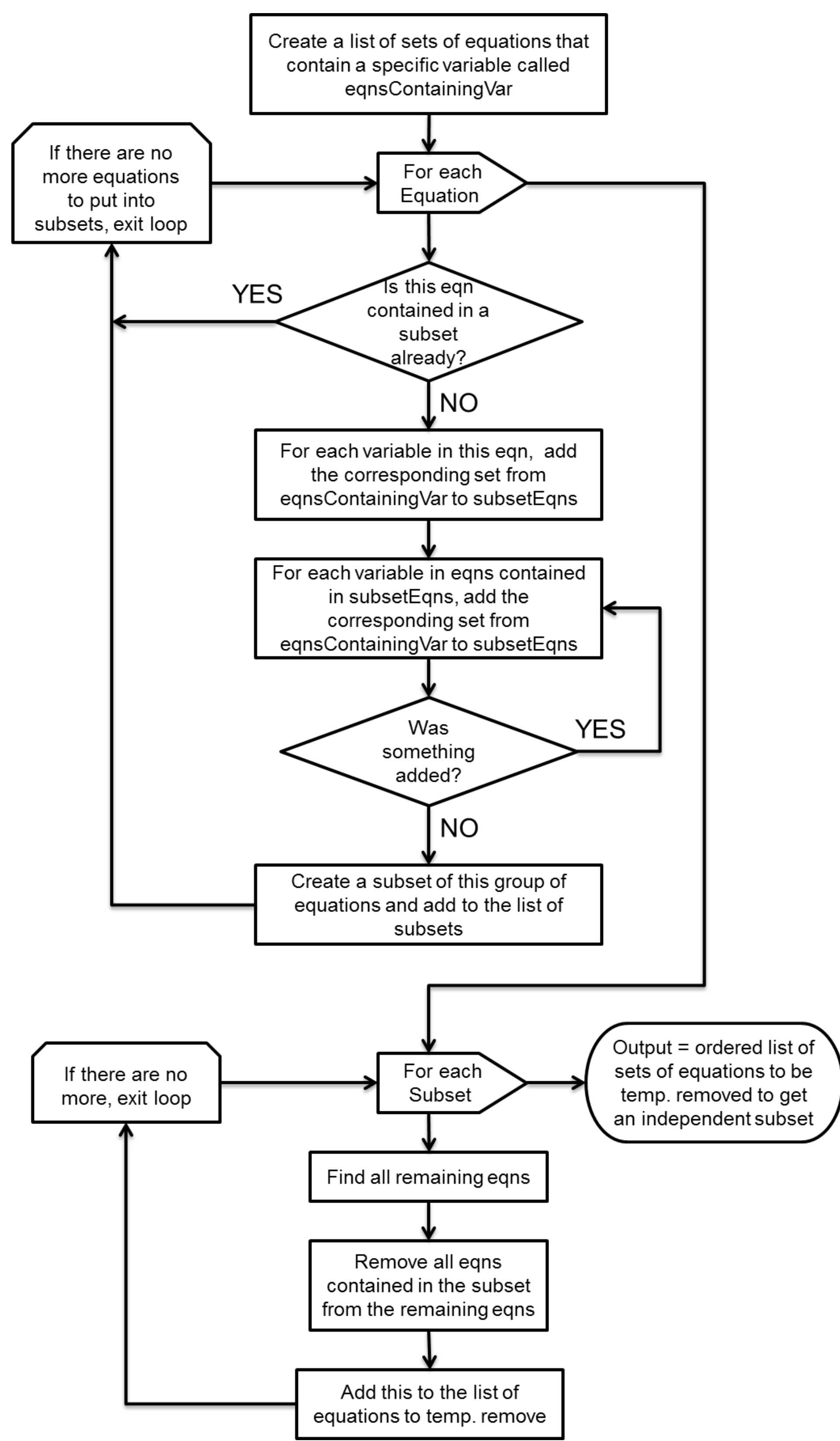

Figure 5. Subsetting algorithm flow diagram.

4.4 Layering Algorithm

After the Linear algorithm constructs a linear solution path, the Layering algorithm creates a layered solution path that is amenable to parallel processing. The Layering algorithm uses graph theory, and our implementation utilizes the Java graph libraries JGraph and JGraphT [10, 11].

The Layering algorithm determines which independent equation blocks can be solved simultaneously on parallel processors. The solution path determined by the Linear algorithm is used to create a corresponding directed acyclic graph (DAG) in which equation blocks are represented by vertices and variable dependencies are represented by edges. To determine which blocks can be placed in a layer, the layering algorithm begins at the top of the graph and searches for edges between vertices. After a layer has been filled with all possible blocks, the algorithm collapses the edges between the vertices in the current layer and the vertices in the next layer. By doing so, the vertices in the next layer then become the top of the graph.

In Figure 6-1 the Layering algorithm adds each block to the DAG as a vertex. Then, each block is evaluated for the variables the block solves for and the variables the block depends upon. Based on the variable dependencies, the algorithm connects directed edges between blocks in the DAG. The direction for the edges is *from* the block that solves for a variable *to* any blocks that depend upon the variable. The algorithm continues to add blocks until all the blocks from the linear solution path have been added to the DAG.

In Figure 6-2, the Layering algorithm searches for blocks that are completely independent of any other blocks. The algorithm first checks whether a block does not depend upon any variables (has no incoming edges). Thereafter it checks whether there are any blocks that depend on it (has any outgoing edges). This process continues until all the blocks in the Linear solution path have been evaluated for incoming and outgoing edges. Independent blocks have both no incoming edges and no outgoing edges. If an independent block is found during this search, it is set aside as a block that can be solved at any time and in parallel with any other block.

At this point a solution path graph has been created, and the relationships among vertices are defined in the graph. The next step consists of ordering blocks in the graph based on these relationships, thereby creating a solution path tree. The process in Figure 6-3 takes an equation

block from the graph and evaluates the variables and blocks upon which it depends. The algorithm then updates the DAG with the block as a "child" of the blocks upon which it depends. This step creates a hierarchical tree of blocks based on block dependencies.

In Figure 6-4 the Layering algorithm creates the Layered Solution Path by traversing the DAG to find independent blocks that can be placed in the same layer. The algorithm takes a block from the graph and checks for any incoming edges. When a block with no incoming edges is found, it is added to the layer that is being assembled. This process continues until the algorithm finds all the blocks that have no incoming edges, which is a signal to the algorithm that a new layer needs to be created. Before the Layering algorithm creates a new layer, it checks to see if there are any unused processors available (i.e. if the number of blocks in the layer is less than the number of processors). The algorithm fills the unused processors with any available independent blocks identified in Figure 6-2. If there are both available processors and independent blocks, then these blocks are added to the current layer to be solved by the unused processor. This process continues until there are no more available processors or there are no independent blocks available. Next, the algorithm removes both the blocks on the current layer their outgoing edges from the DAG. A new layer is created and algorithm searches again for blocks that have no incoming edges. The Layering algorithm repeats these steps in Figure 6-4 until all the blocks in the solution path graph have been added to the Layered Solution Path.

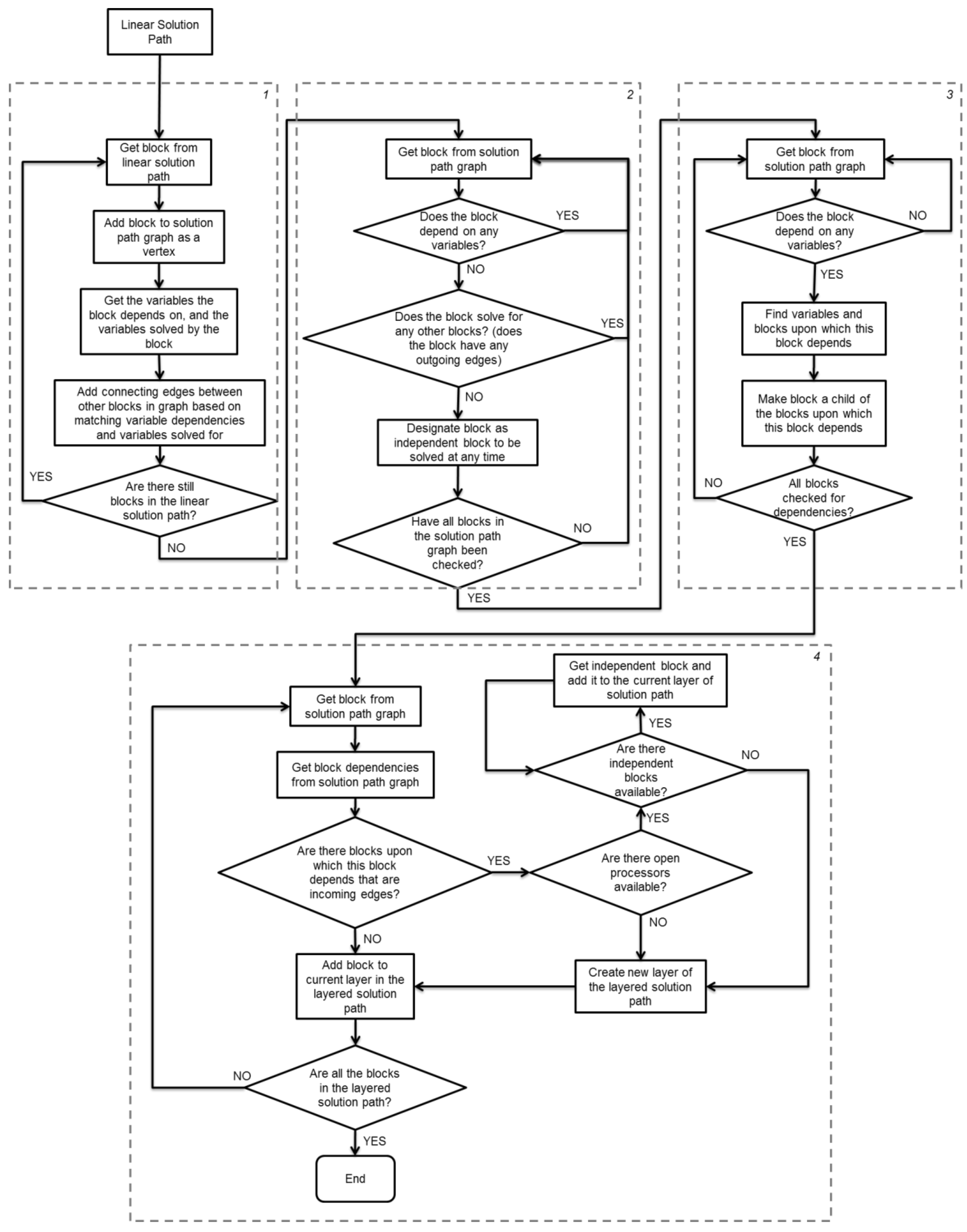

Figure 6. Layering algorithm flow diagram.

4.5 Solution Path Comparisons

A main concept in speed enhancement for the Linear and Layering algorithms is reducing the number of equations to be solved at any one time by the solver, because numerical solution of systems of equations goes as $O(N^2)$ where N is the number of equations. A Layered solution path is faster than a Linear solution path, which, in turn, is faster than a Single Step solution path in which all equations in a system are solved simultaneously.

Although there is an increase in speed when comparing the Linear path to the Single step path, and the Layered path to the Linear path, there are costs associated with set-up time for Linear and Layered paths. In the case of the Linear Solution Path, additional set up time is required to run the Linear algorithm (both Blocking and Subsetting). The Layered Solution Path requires running the Linear algorithm *and* the Layering algorithm, further adding to the set up time. A table comparing three example solution paths can be seen below in Table 1.

Table 1. Solution path comparison.

	Single Step	Linear	Layered
	N = 100 B = 1 b = undefined L = 1 l = undefined	N = 100 B = 5 b = .04 L = B l = 0	N = 100 B = 5 b = .04 L = 2 l = .75
Setup Considerations	No Setup Time	Linear Algorithm Required	Linear and Layering Algorithms Required
Solve Considerations	$t_{Solve}: O(N^2)$	$t_{Solve}: O\left(B\left(\frac{N}{B}\right)^2\right)$ All blocks solved on same thread	$t_{Solve}: O\left(L\left(\frac{B/L}{N_{processors}}\right)\left(\frac{N}{B}\right)^2\right)$ Multiple threads will decrease t_{Solve}, overhead of thread management will increase t_{Solve}

5. Testing Methods

To assess the benefits of the linear and layering algorithms and the advantages of parallel processing, several methods for generating systems of equations were developed. These systems of equations were then used for testing the computational speed of various solution paths.

5.1 User Created Systems of Equations

Ten systems of equations were created for the purpose of validating the algorithms and solver (See Figure 7). These 10 systems of equations have a variety of complexities and sizes to allow thorough testing of the linear and layered algorithms. These systems of equations were solved by three methods:

- By inspection and hand calculations.
- The Engineering Equation Solver (EES) software package [12].
- The X-Solve software package which uses Blackketter's logical equation set decomposition (LESD) algorithm [13].

The two software packages (EES and X-Solve) confirmed that the systems of equations could be solved using numerical methods. The solutions from the three methods were compared to confirm that they agreed. We tested the systems of equations with our Newton-Raphson solver (based on Java Matrix Package (JAMA)) and solution paths to verify the accuracy of our code [14].

Equation Set 1

E0: $a = 1$
E1: $b = 2$
E2: $c = 3$
E3: $d = 4$

Equation Set 2

E0: $a + b = 5$
E1: $b + c = 7$
E2: $a – c = b + 4$

Equation Set 3

E0: $ab = 10$
E1: $bc + d = 15$
E2: $c/d = 2$
E3: $a – d = 2$

Equation Set 4

E0: $a = 4$
E1: $c = 6$
E2: $a + b = c$
E3: $b + c = d$
E4: $c + d = e$

Equation Set 5

E0: $ab = 0$
E1: $b\cos(a) + c\sin(d) = 12$
E2: $c + b = 18$
E3: $d + a = 30$

Equation Set 6

E0: $a = 16$
E1: $b = a^{0.5} + c - d$
E2: $c^2 + b = 2d$
E3: $d^2 + 2d + 5 = 10c + b$

Equation Set 7

E0: $a + b = 20$
E1: $c = 2a$
E2: $c – b = 2 + a$

Equation Set 8

E0: $a + b = 7$
E1: $b – a = 3$
E2: $c = a + d$
E3: $d = bc$
E4: $f = a/b$
E5: $g^2 = b^2 – a$
E6: $ea = f^b$

Equation Set 9

E0: $d = 5$
E1: $a + b = 90$
E2: $b/a = 2$
E3: $c = d\sin(a – b)$
E4: $e = (f + 10)\cos^2(a) + \cos^2(b)$
E5: $f = 4(e – 1) + 1$
E6: $g = c + d + e - f$

Equation Set 10

E0: $a^2 – 2ab + b^2 = 1$
E1: $a^2 + b = 7$
E2: $c^2 + bc + a = d + 1$
E3: $d + 2c = 10$
E4: $g = fa$
E5: $f = g - b$
E6: $e = d^c$
E7: $h = gef$

Figure 7. Systems of equations used for testing.

5.2 Creation of Large Systems of Equations for Testing

Many different types of systems of equations exist with varying sizes, blocking, and layering. To test our algorithms with systems of equations with a variety of blockiness (*b*) and layeredness (*l*), we created systems of equations by first creating the desired occurrence matrix. This occurrence matrix can be used to ensure the equation set has 3 of the 4 defined characteristics the user desires: the number of equations, the number of blocks, and the number of layers. The number of equations is set by the size of the occurrence matrix. To achieve the desired number of blocks an equal number of equations is distributed to each block to the degree possible. If the number of equations can't be distributed evenly, the remaining equations are added to the last blocks. For example, if 8 equations in 5 blocks is desired, 8 / 5 = 1 with a remainder of 3 therefore the resulting block sizes would be [(1) (1) (2) (2) (2)]. A similar method is used to set the number of blocks on each layer.

After the blocks are created, they are added to the occurrence matrix. For 8 equations in 5 blocks and 1 layer, the occurrence matrix shown in Figure 8 is created.

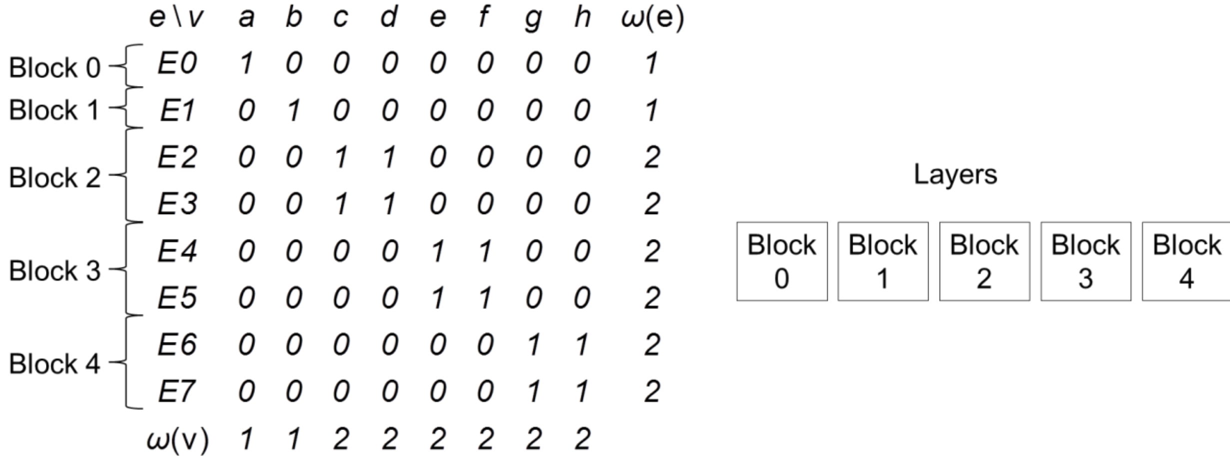

Figure 8. Blocking for N = 8, B = 5.

For a different layering structure, these blocks need to be judiciously linked. The blocks are distributed as evenly as possible to the layers. This method of layering can be seen in Figure 9 below. By adding variables that are solved in one block to equations in another block, blocks become dependent upon another, resulting in more than 1 layer. Continuing with the prior example, 4 layers can be constructed as shown by the following occurrence matrix. There are many different approaches to achieve the correct blocking and layering for given values of the parameters *b* and *l*. Our method provides consistency so that solution times are repeatable.

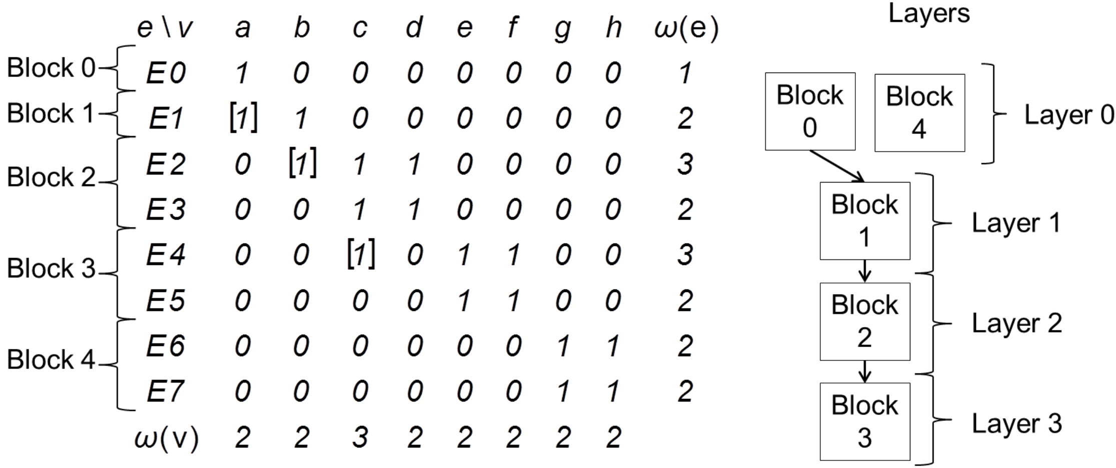

Figure 9. Layers added to blocking for L = 4. (Bracked values create layer dependencies.)

After the occurrence matrix is constructed, equations are generated with variables as dictated by the occurrence matrix. The equations were created as polynomials of arbitrary odd order. Coefficients (*c*) for the variables and a constant (*a*) were also added to each equation using a random number generator that produces values between -0.5 and 0.5. The value of the constant on the right hand side of the equation (*a*) was determined by dividing the number of terms in a created polynomial by 2. The form of each equation is as follows (See Equation 3).

$$\sum_{1=0}^{N-1}\sum_{j=1}^{o} c_{i,j-1} x_i^{2j-1} = a \qquad (3)$$

The variables *N* and *o* represent the number of variables in the system of equations and the polynomial order, respectively. Equation 4 is an example equation generated by Equation 3, with $o = 5$, $N = 2$, and *c* and *a* representing the randomized matrix of coefficients. Systems of these polynomial equations were used for testing our Linear and Layering algorithms.

$$c_{00} x_0^{\,1} + c_{01} x_0^{\,3} + c_{02} x_0^{\,5} + c_{10} x_1^{\,1} + c_{11} x_1^{\,3} + c_{12} x_1^{\,5} = a \qquad (4)$$

5.3 Blackketter Equation Block

To determine the advantages of layering and parallel processing and to compare computational times in a manner similar to Alexander and Blackketter, we implemented a system of equations

similar to the one used by Blackketter [8]. The Blackketter equation block consists of 4 equations. To create a larger system of equations, we replicate the Blackketter equations several times depending on the desired size of the overall system of equations. This method creates a system of equations with $l = 1$. See Figure 10 for a sample occurrence matrix with Blackketter equation blocks for $N = 8$.

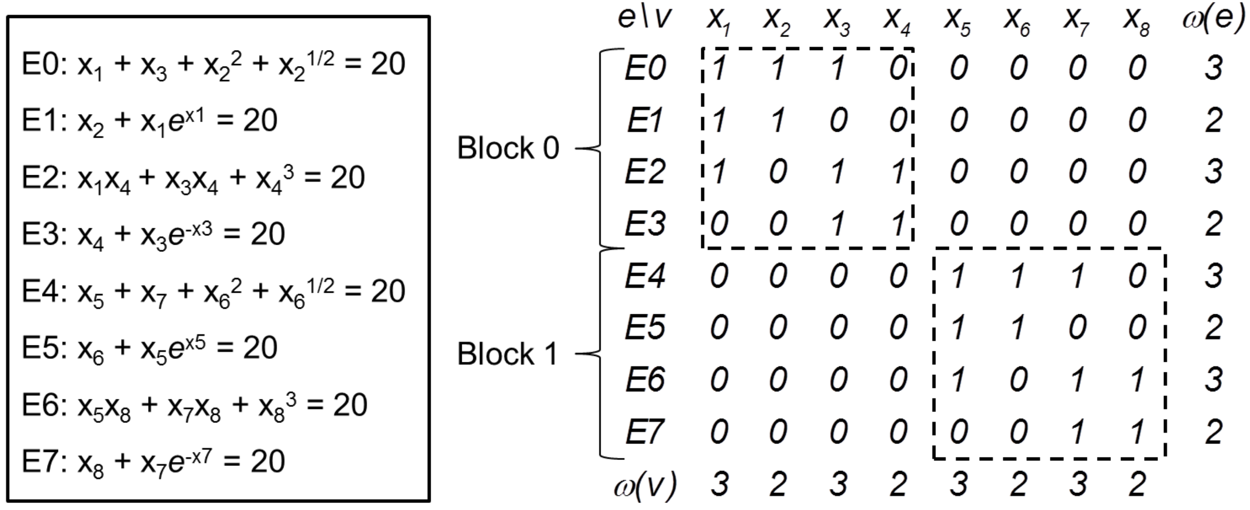

Figure 10. Sample Blackketter system of equations with two equation blocks.

6. Results

6.1 Computers Used For Testing

There were four different computers used for conducting tests, including the Calvin College 16 core supercomputer Dahl [15]. These computers are summarized in Table 2 below.

Table 2. Characteristics of computers used for testing.

Computer	Processor Type	Number of Cores	Speed	Operating System	Java Version
Apple MacBook Pro	Intel core i7	4	2 GHz	Mac OSX 10.7.3	Java 1.6.0_3
HP Pavilion dv6000	Intel Centrino	2	2 GHz	Windows 7	Java 1.6.0_26
Dell Optiplex 780	Intel Core 2 duo	2	2.93 GHz	Windows 7	Java 1.6.0_29
Dahl Super Computer	4 quad-core Intel Xeon 7300	16	2.4 GHz	Ubuntu 10.04 LTS	Java 1.6.0_31

6.2 Testing Methods

Tests were developed in Java that would run the algorithms with a given system of equations. Millisecond timing was used to measure the time that the algorithms spent decomposing the equations and the time that was spent numerically solving the equations. To account for problems that may arise with garbage collection in Java, 11 runs were executed each time (where a run consists of the system of equations being decomposed and solved). The first run was thrown out each time, to account for any set up anomalies associated with initializing the code, and the set up and solve times of the subsequent 10 runs were averaged.

6.3 Advantages of Subsetting Algorithm Over a Combination Generator

One of the greatest advantages of being able to create large systems of polynomial equations is that all types of equation systems can be tested for setup time considerations. We observed that use of the combination generator only (Figure 4-5) without subsetting (Figure 5) caused setup time to be unacceptably large for systems of equations with low values of blockiness (b), due to the number of combinations being very large. For systems of equations with high blockiness, the combination search is computationally *inexpensive*, and preferred to the subsetting algorithm (Figure 5).

Equation 5 shows the number of equation combinations where N is the number of remaining equations, and *iCalc* is the size of the block being searched for. The time to find a block is proportional to the number of combinations.

$$N_{Combinations} = \frac{N!}{iCalc!(N - iCalc)!} \tag{5}$$

Set-up times are shown in Figure 11. We observe a steep increase in setup time as the number of blocks (B) decreases when using the combination generator alone. This increase happens because the peak number of combinations for a specific system of equations occurs when $N = 2(iCalc)$ which is what the system of equations approaches as blockiness decreases.

The subsetting algorithm drastically reduces the number of equations being sent to the combination generator. This resulted in much more reasonable setup times for the linear solution path; in Figure 11, these times are always less than 70 ms, even with few blocks.

The subsetting algorithm increases the computational burden of the linear algorithm resulting in increased setup times for systems of equations with high blockiness. (See the inset of Figure 11 above 80 blocks.) Of course, one does not know, *a-priori*, the blockiness characteristic of a system of equations. But, it appears to be an acceptable tradeoff to use subsetting for every situation, as the time penalty in situations with high blockiness is very small.

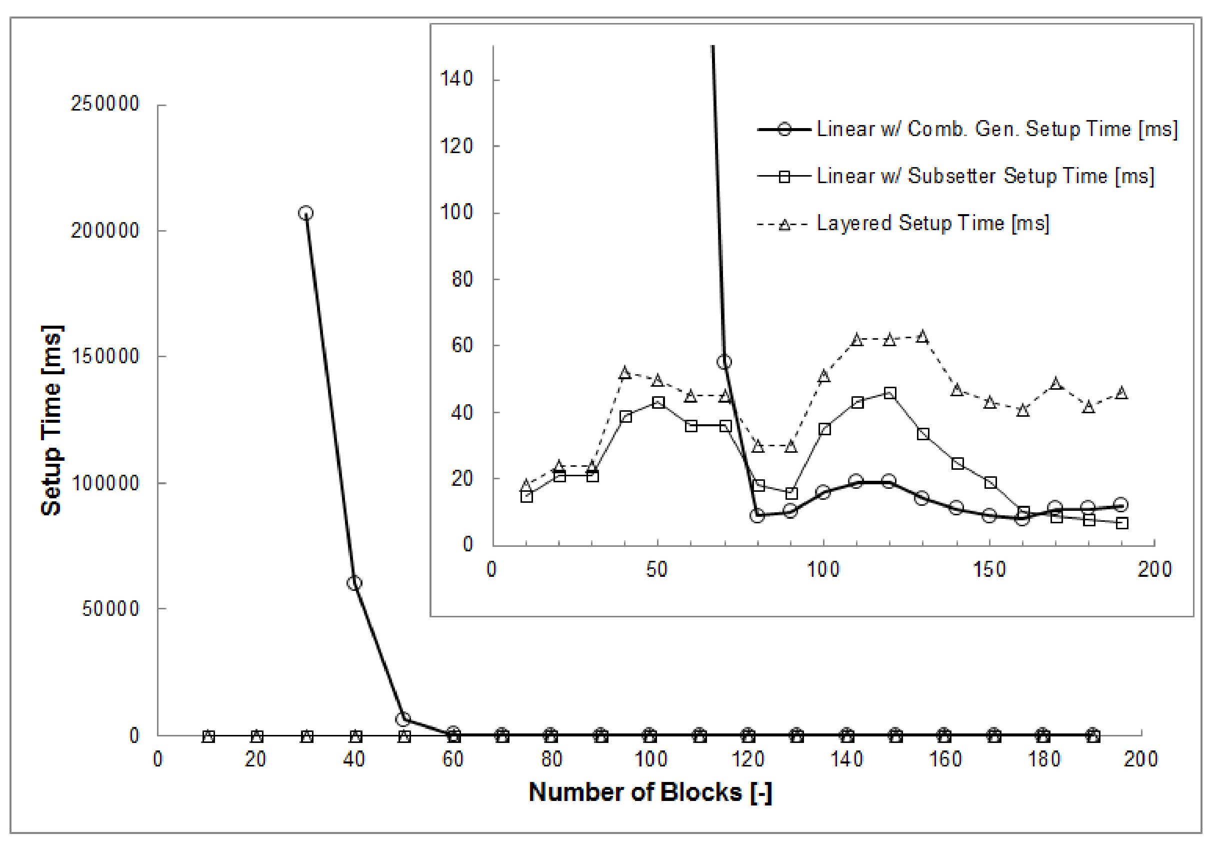

Figure 11. Setup time as a function of number of blocks. N = 200, l = 0.7 (HP Pavilion dv6000).

6.4 Speed Differences Among the Three Solution Path Types

The benefit offered by the Linear and Layering algorithms is a decrease in numerical solution time for a system of equations. Both solution paths offer a large increase in computational speed compared to the Single Step solution path. Figure 12 shows a comparison between the solve

times (not including setup time) for the three solution paths. A Blackketter system of equations with up to 1000 blocks (4 equations per block) was used to test the three solution paths.

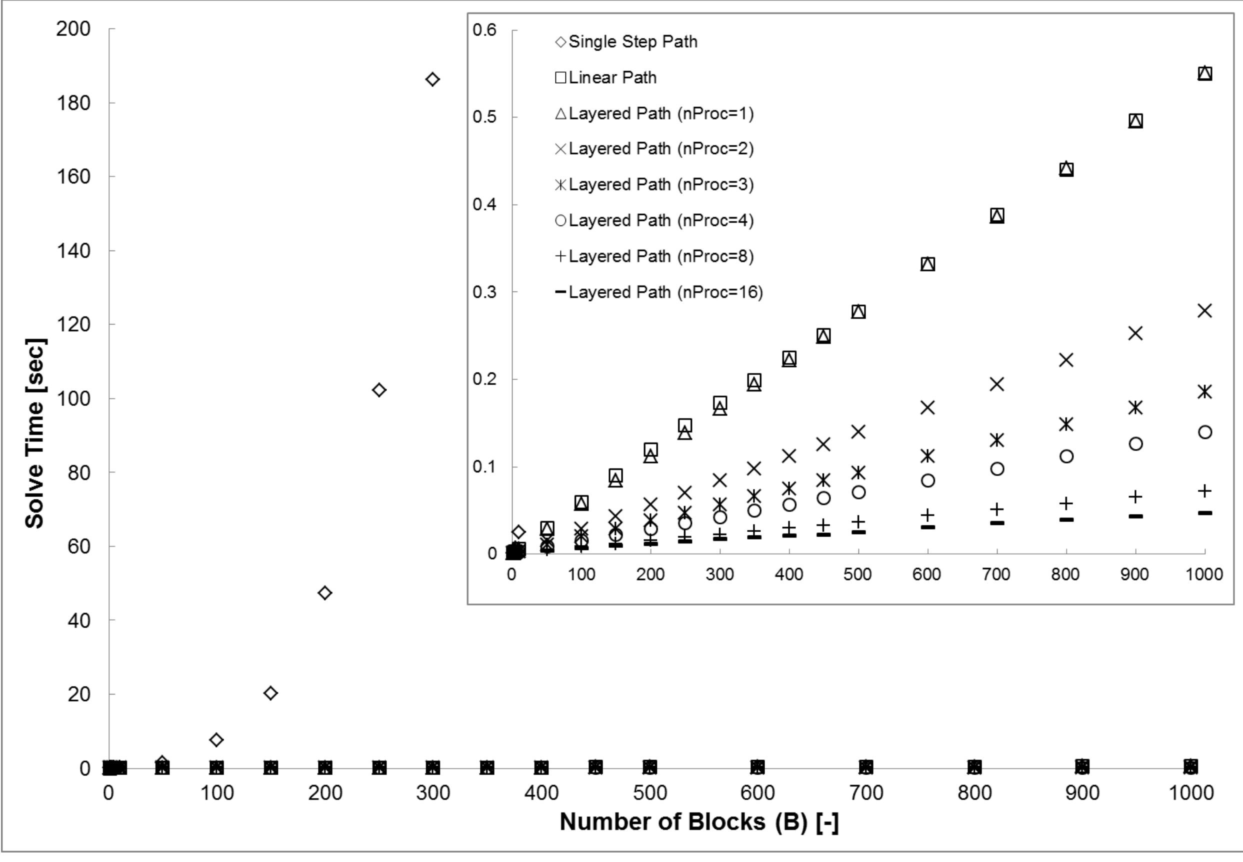

Figure 12. Solve time as a function of number of blocks (Dahl Supercomputer).

The solve time for the Blackketter equations using the Single Step solution path quadratically increases, consistent with $O(N^2)$ execution time for the standard Newton-Raphson technique. In contrast, the time taken to solve the system of equations with both the Linear and Layered solution paths increases linearly with the number of blocks, consistent with the results from Blackketter (2003). The Layered solution path on 1 processor has nearly the same performance as the Linear solution path, as expected. The benefits of multiple processors are obvious for the Layered solution path.

6.5 The advantages of parallel processing

As expected, utilizing parallel processing for solving independently solvable equation blocks simultaneously offers another increase in computational speed. As shown in Figure 13, as the

number of parallel processors increases for a system of equations with high layeredness (l = 1.0 in this case), the solve time decreases. This effect is shown in Figure 13 as a decreasing ratio of Layered solve time to Linear solve time as the number of processors increases.

Each additional processor reduces the required solve time significantly in the case of high layeredness, which takes full advantage of parallel processing. For lower values of layeredness, the speed advantage is not as pronounced. Figure 14 further shows the advantages of parallel processing, as the relative solve time ratio decreases as the number of processors increases.

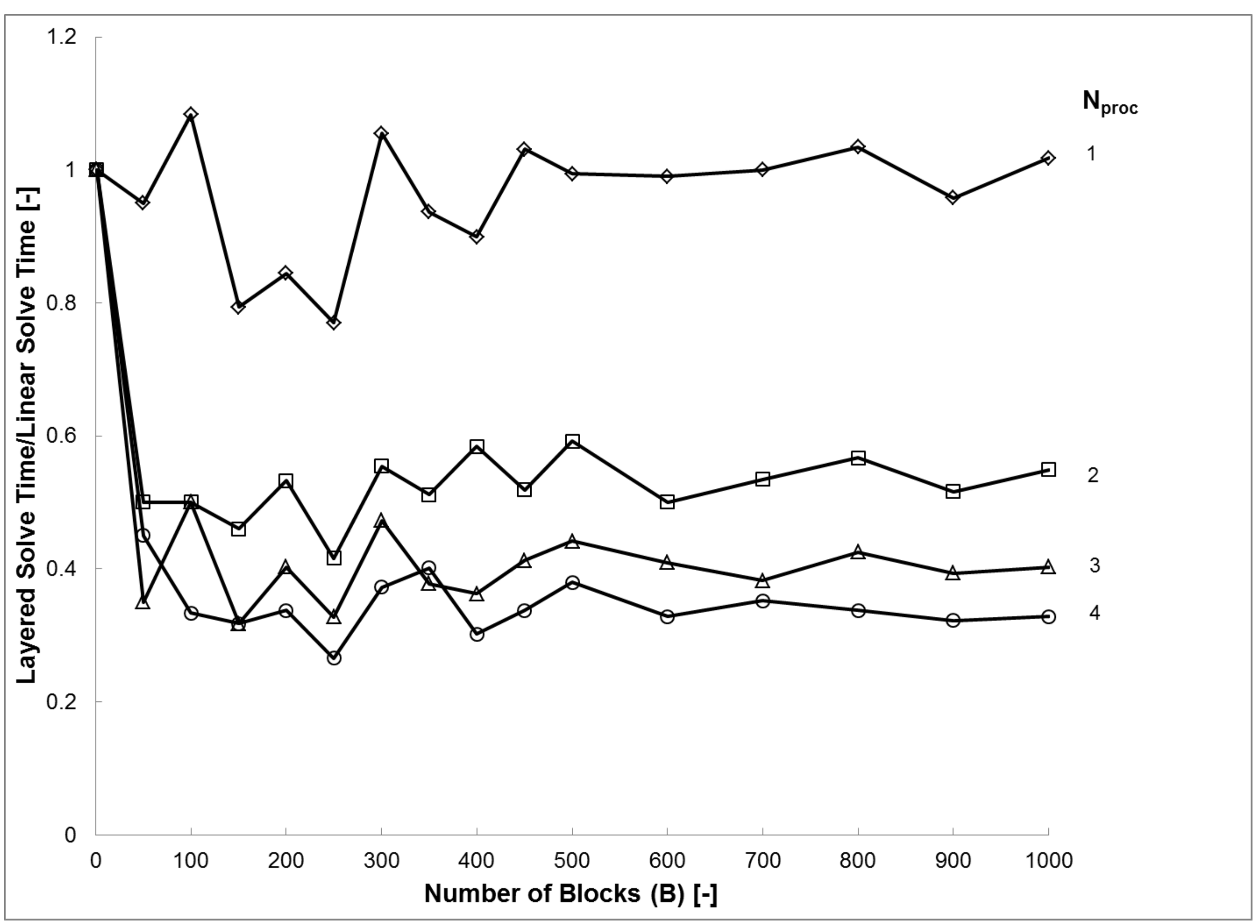

Figure 13. Relative solve time (Layered time/Linear time) as a function of number of blocks for multiple processors (Apple MacBook Pro).

However, there are some costs associated with parallel processing. These costs are seen in the additional time that needs to be dedicated to thread management when multiple processors are added. It is expected that additional processors reduce the solve time by $1/N_{proc}$. Figure 14 shows

a comparison between this theoretical time ratio ($1/N_{proc}$) and the actual time ratio for varying numbers of processors on different computers. All computers show time ratios above the theoretical line, as expected, due to the overhead of managing threads. With 2000 equations in 500 blocks with 16 processors, we show that the multi-core approach decreases the solve time by a factor of 11.2, or 70% of the theoretical speed increase, compared to single-core techniques.

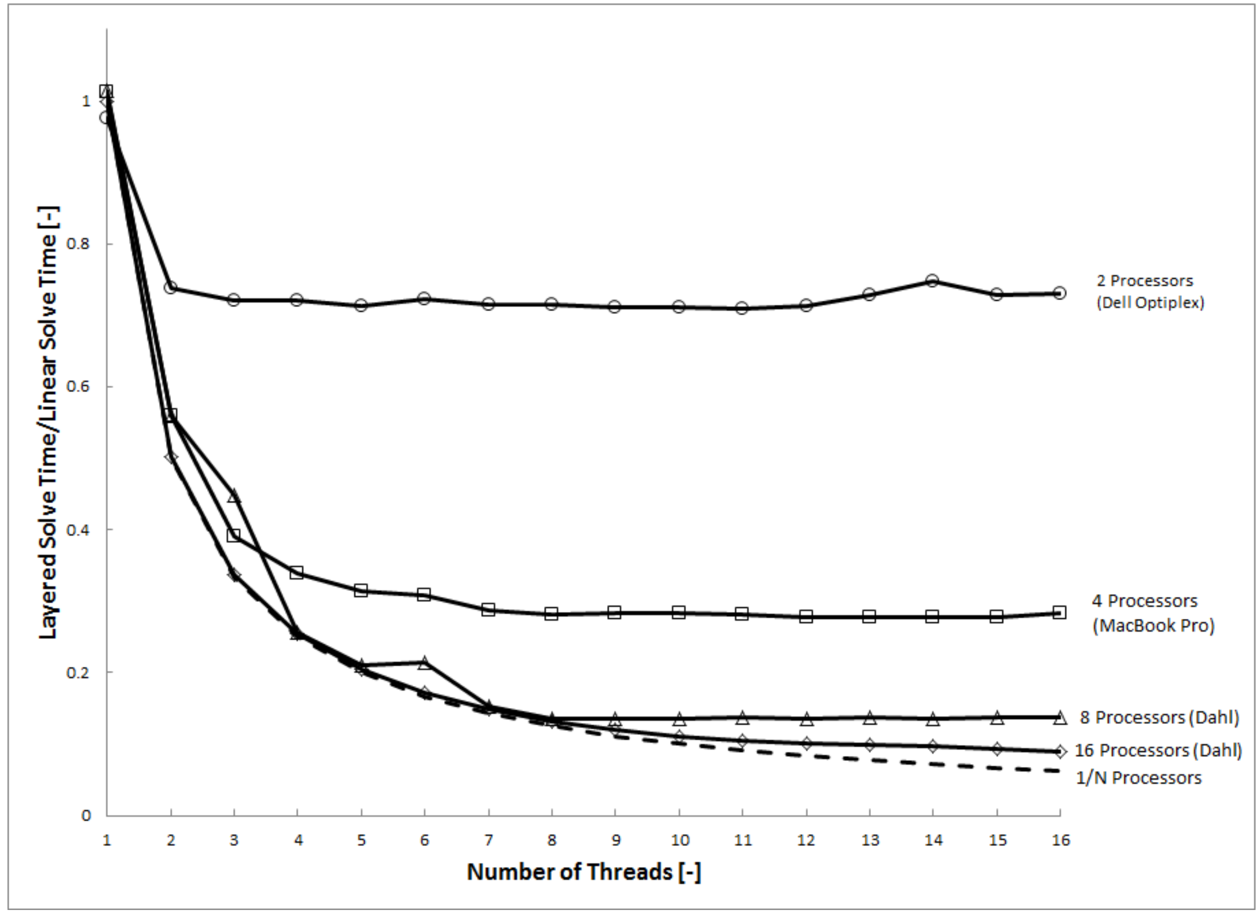

Figure 14. Relative solve times (Layered time/Linear time) as a function of number of threads being used (Blackketter equation set, N = 2000).

7. Conclusion

The advantages of parallel processing have been shown by developing two algorithms, the Linear Algorithm and Layering Algorithm and by using them for decomposition of large systems of equations. Previous studies were confirmed that showed advantages to using a linear solution path, and we showed that layering can improve solve times further. The speedup advantages

were demonstrated on multi-processor machines, and we confirmed that parallel processing offers advantages in the solve time of large systems of equations.

8. Acknowledgements

We are grateful to the Jack and Lois Kuipers Applied Mathematics Endowment for providing summer research fellowships for Mr. Helmus and Mr. Reese.

Chris Rozema developed the Newton-Raphson solver used for this paper.

Global Aerospace Corporation provided use of their code libraries and advice.

Calvin College Computer Science & Information Systems Department provided access to the Dahl supercomputer.

Calvin College professor J. Adams provided advice and support for using Dahl throughout this project.

9. References

[1] Himmelblau DM (1966) Decomposition of large scale systems – I. Systems composed of lumped parameter elements. Chem Eng Sci. 21: 425-438. doi: http://dx.doi.org/10.1016/0009-2509(66)85054-6

[2] Himmelblau DM (1967) Decomposition of large scale systems – II. Systems containing nonlinear elements. Chem Eng Sci. 22: 883-895. doi: http://dx.doi.org/10.1016/0009-2509(67)80152-0

[3] Ramirez FW, Vestal CR (1972) Algorithms for structuring design calculations. Chem Eng Sci. 27: 2243-2254. doi: http://dx.doi.org/10.1016/0009-2509(72)85102-9

[4] Duff IS, Reid JK (1978) An implementation of Tarjan's algorithm for the block triangularization of a matrix. ACM Trans on Math Softw. 4 (2): 137-147. doi: http://dx.doi.org/10.1145/355780.355785

[5] Tarjan, RE (1972) Depth first search and linear graph solutions. SIAM J Comput. 1: 215-225. doi: http://dx.doi.org/10.1137/0201010

[6] Sargent RWH, Westerberg AW (1964) "SPEED-UP" in chemical engineering design. Trans Inst Chem Engr. 42: 190-197.

[7] Sridhar N, Agrawal R, Kinzel, GL (1993) Active occurrence-matrix-based approach to design decomposition. Comput-Aided Des. 25 (8): 500-512. doi: http://dx.doi.org/10.1016/0010-4485(93)90081-X

[8] Alexander DG, Blackketter DM (2003) Optimized solution strategy for solving systems of equations. Eng Comput. 20 (2): 178-191. doi: http://dx.doi.org/10.1108/02644400310465308

[9] Chan B, Abdelrahman TS (2004) Run-Time Support for the Automatic Parallelization of Java Programs. J Supercomput. 28 (1): 91-117. doi: 10.1023/B:SUPE.0000014804.20789.21
[10] Gaudenz A (2000) JGraph. JGraph. http://www.jgraph.com/. Accessed 21 May 2012.
[11] Naveh B (2003) JGraphT. JGraphT. http://jgrapht.org/. Accessed 21 May 2012.
[12] Beckman WA, Klein SA (1992) Engineering Equation Solver. F-Chart Software. http://www.mhhe.com/engcs/mech/ees/. Accessed 21 May 2012.
[13] Alexander DG, Blackketter DM, Cegnar E (2004) X-Solve. Caelex, Inc. http://www.caelex.com/x_solveinfo.htm. Accessed 10 June 2011.
[14] Hicklin J, Moler C, Webb P, Boisvert RF, Miller B, Pozo R, Remington K (2005) JAMA. National Institute of Standards and Technology. http://math.nist.gov/javanumerics/jama/. Accessed 21 May 2012.
[15] Adams J (2008) Dahl.calvin.edu – HPC at Calvin. Calvin College. http://dahl.calvin.edu/history/. Accessed 21 May 2012.
[16] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". *Communications of the ACM* **16** (6): 372–378