8-2011

A Low Communication Condensation-based Linear System Solver Utilizing Cramer's Rule

Kenneth C Habgood
khabgood@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Kenneth C Habgood entitled "A Low Communication Condensation-based Linear System Solver Utilizing Cramer's Rule." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Itamar Arel, Major Professor

We have read this dissertation and recommend its acceptance:

Fangxing Li, Gregory Peterson, Xiaobing H. Feng

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Kenneth C. Habgood entitled "A Low Communication Condensation-based Linear System Solver Utilizing Cramer's Rule." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Dr. Itamar Arel, Major Professor

We have read this dissertation
and recommend its acceptance:

Dr. Fangxing Li

Dr. Gregory Peterson

Dr. Xiaobing H. Feng

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

A Low Communication Condensation-based Linear System Solver Utilizing Cramer's Rule

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Kenneth C. Habgood

August 2011

Acknowledgements

I would like to thank Dr. Arel for his patience and eternal optimism, as well as the Innovative Computing Laboratory (ICL) for their insightful suggestions and feedback.

Abstract

Systems of linear equations are central to many science and engineering application domains. Given the abundance of low-cost parallel processing fabrics, the study of fast and accurate parallel algorithms for solving such systems is receiving attention. Fast linear solvers generally use a form of LU decomposition. These methods face challenges with workload distribution and communication overhead that hinder their application in a true broadcast communication environment.

Presented is an efficient framework for solving large-scale linear systems by means of a novel utilization of Cramer's rule. While the latter is often perceived to be impractical when considered for large systems, it is shown that the algorithm proposed has an order Nˆ3 complexity with pragmatic forward and backward stability. To the best of our knowledge, this is the first time that Cramer's rule has been demonstrated to be an order Nˆ3 process. Empirical results are provided to substantiate the stated accuracy and computational complexity, clearly demonstrating the efficacy of the approach taken.

The unique utilization of Cramer's rule and matrix condensation techniques yield an elegant process that can be applied to parallel computing architectures that support a broadcast communication infrastructure. The regularity of the communication patterns, and send-ahead ability, yields a viable framework for solving linear equations using conventional computing platforms. In addition, this dissertation demonstrates the algorithm's potential for solving large-scale sparse linear systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

At the heart of many scientific computations is finding the solution to a system of linear equations. Simulation and optimization of power systems, for example, repeatedly solves systems of linear equations. In order to efficiently solve these systems, reliable tools and methods are required. A number of different approaches exist, and most depend on an algorithm to quickly and accurately find the solution.

This dissertation focuses on one particular algorithm that shows great promise. The approach is based on two established mathematical methods: Cramer's rule and Chio's matrix condensation. When combined, these methods can solve for a set of variables in a highly distributed fashion. At a first glance, a simple combination of these methods involves higher computational complexity than existing techniques. However, the addition of a clever data structuring scheme allows this algorithm to efficiently solve a linear system while still retaining its distributed nature.

Like all numerical methods, it must produce solutions that are accurate and timely. For these reasons, the algorithm requires careful study of numerical stability, accuracy and computational complexity. Once a level of confidence is established in these traits, the parallelization potential can be explored.

1.1 Numerical Solutions to Systems of Linear Equations

Historically, the algorithm that required the least amount of computation to solve a system worked best. Not only could it complete its work quicker than more computationally complex algorithms but fewer calculations generally meant fewer rounding and truncation problems. As hardware speed improved, the time to complete floating point calculations decreased and the bottle neck shifted to memory bandwidth. The time required to solve a problem was dominated by how often an algorithm moved data in and out of the processor. For this reason, a great deal of research focused on algorithms that efficiently moved data through a machine's memory hierarchy.

As networks and multicore computers became cost-effective, new platforms arrived that could distribute work over more hardware. Calculations could be done in parallel to reduce processing time. Unfortunately, the nature of linear systems makes it difficult to solve pieces independently. This leads to communication between processing units to share data. The run time of distributed solvers depends not only on the number of calculations but also on the amount and speed of communication.

In addition, the problem of workload for each unit has become a consideration for distributed solvers. An algorithm that can distribute calculations uniformly across the different processors will complete quicker, since the run time of an algorithm depends on the last processor to complete its work. An unbalanced work load results in some processors lagging behind while others wait. This leads to allocating work in non-intuitive ways so that resources do not remain idle.

For these reasons, efficient linear solvers have continually been of interest in scientific computing. Not only must an algorithm display accuracy and stability, but it must now consider effectively balancing workloads, communication overheads and memory optimization. The only constraint that has lessened has been the amount of computational workload due to the improved speed of hardware.

1.2 Algorithm Requirements

Research into an algorithm needs criteria to evaluate its effectiveness. A successful process in one situation may not be optimal for another. An algorithm that runs very quickly with less stability may be beneficial in real-time systems, whereas an extremely accurate and stable method that requires extended run times is less useful. In most cases, the user must identify a balance between these requirements. The following items are the attributes considered important in this thesis for evaluating a parallel linear system solver.

1.2.1 Accuracy and Stability

Paramount to any numerical method is the ability to provide an adequately accurate answer. The acceptable level of accuracy depends on the application but in all situations a basic level of accuracy is desirable. There can be a balance between high levels of accuracy and inefficient calculations. Pivoting in LU factorization provides a good example. Complete pivoting produces a more stable algorithm whereas partial pivoting a potentially less stable algorithm, however most actual implementations use partial pivoting. The reason is partial pivoting provides a more efficient algorithm with little loss in accuracy. This, again, emphasizes the balance between different aspects of an algorithm.

Errors that reduce accuracy and ultimately cause instability are generally divided into two categories: round-off error and truncation error [20]. Round-off error can be combated by using higher precision. In computer hardware terms, this means more bits to store data. Floating point values generally stored as singles can be expanded to doubles where arithmetic calculations will involve more digits and thus more accurate answers. This comes at the cost of larger memory requirements and additional hardware.

Truncations errors arise from the inability of a computer or process to exactly represent the solution to an individual calculation. This type of problem is generally associated with operations that involve a finite number of steps [39]. For example, if a process provides a second order polynomial as a solution but the exact answer is actually a third order polynomial, error is introduced.

If these inaccuracies, generally referred to as computational errors, can be made small and remain small even as the problem size grows, an algorithm is considered stable. The challenge becomes presenting an argument which proves that an algorithm is stable. The two most common strategies are forward analysis and backward analysis [26]. Forward error is a relatively intuitive measure. It's a bound on the difference between the algorithm's solution and the actual correct solution. The challenge here is in providing a known correct solution for comparison. The other strategy, backward analysis, estimates the potential perturbations that the algorithm could 'impress' upon the original data [39]. These strategies can also take the label of a priori error analysis.

1.2.2 Performance Attributes

There are a number of standard measures for the performance of an algorithm, including floating-point operations per second, wall time, and processor cycles. In simplest terms, an algorithm that returns an answer in less time than another will be considered faster. This speed is dependent on a number of factors and an algorithm that completes quickly on one hardware platform may perform poorly on another. Below are factors that affect how quickly a algorithm can deliver a solution.

Computation Complexity

The amount of computation, or number of arithmetic operations, is an indicator on how long an algorithm will take to return a solution. A measurement of this values is typically given in big $\mathcal{O}$ notation and ties to the size of the problem. Since hardware platforms differ in how many operations they can do per time period this notation provides a means to compare the algorithms as opposed to the hardware their implement on. An algorithm with a lower computational complexity should consistently compute quicker than a higher complexity algorithm, independent of the computer used.

Memory Optimization

One of the other key influences on a program's efficiency is how much time the processor sits idle waiting on data from memory. In order to minimize a CPU's idle time an in depth understanding of the memory hierarchy and hardware platform are required.

Systems typically have a memory hierarchy with large amounts of slow memory at the bottom and very small amounts of fast memory at the top. Moving the data between these different levels requires time. The optimization of memory tries to minimize data movement between these layers. If that's not possible the software can try to pre-schedule a miss so that the processor has data before it needs it. This essentially hides the calls to memory by having it happen while the CPU is working with other data. The term for this concept is prefetch [42].

Communication Complexity

The nature of linear systems requires communication between parallel processes once the data has been distributed. Inter-process communication generally takes the form of one-to-one messages or broadcast messages. If all the processes need a particular piece of data then a message can be broadcast. Depending on the underlying network system in use, this can be extremely efficient. For example, an Ethernet network readily supports broadcasts and can reduce the amount of traffic handled by the network. A process can send one broadcast message that is replicated by the network and distributed to all other process, rendering the sending process available to work on other tasks.

Communication complexity also encompasses the overhead associated with tracking which processes require what pieces data [29]. The book-keeping associated with properly sending and receiving data can contribute not only to complexity of the algorithm but to its run time. In addition, the timeliness of the communication can affect run times. If multiple process must wait for data before doing any work, this contributes to processor idle time. If data can be sent early then other processors can begin as soon as they are ready.

Workload Distribution

When multiple processors contribute to a solution it's best if each complete their work at roughly the same time. Since an algorithm doesn't generally return the solution until the last process completes, the run time depends largely on this process. If all processors are equal, then the last process to finish is generally the one with the largest workload. If other processors that were idle could share the workload, then run time can be improved.

1.3 Motivation

The primary motivation for this research was to study an algorithm that can solve large scale linear systems efficiently on enterprise-grade hardware. This conceived scenario involves a small number of parallel nodes with an Ethernet network infrastructure, instead of super-computing grids or large computing centers. This dissertation, and the proposed algorithm, target small sized data centers that have a limited number of servers which can operate in parallel to solve problems for real-time operations. For example, a small utility that needs to continually solve power flow optimization problems to keep it's electric system efficient in day-to-day operations. With this environment in mind, this dissertation focuses on the design of the algorithm and evaluation of its effectiveness.

1.4 Key Contributions

The following are the most important contributions to the body of scientific knowledge from this research:

- The refinement of the mirroring scheme to delay the substitution of the solution vector (b column) till the end of the algorithm. This reduces the bookkeeping and streamlines the algorithm.
- Implementation of the MxM condensation to help reduce the number of memory accesses.
- A detailed error analysis of the algorithm.
- The development of a parallel scheme for the algorithm and evaluation of the communication complexity.
- Exploration of the algorithm for suitability with sparse problems and implementation improvements for dealing with sparse matrices.

1.5 Dissertation Outline

The following chapters provide background and detail on the proposed algorithm and various outcomes of the research. Chapter 2 provides general information pertaining to this area of study, including introduction to the mathematical pieces of the algorithm. Chapter 3 introduces the proposed algorithm in detail. Chapter 4 focuses on the accuracy and stability attributes of the algorithm. Chapter 5 provides results from a serial implementation along with associated optimization details. Chapter 6 discusses the parallel design of the algorithm and the results both in terms of communication complexity and run times. Chapter 7 explores the potential of the algorithm for sparse matrices, in the context of both serial and parallel implementations. The final chapter provides a summarizing discussion and draws key conclusions on the research outcomes.

Chapter 2

Background on Linear Systems Solvers

2.1 Common Direct Solvers

Fast linear solvers generally use a form of Gaussian elimination [22], the most common of which is LU-factorization. This process involves a computation complexity of $W_{LU} \approx \frac{2}{3}N^3$ [28], where N denotes the number of linearly independent columns in a matrix. The factor 2 accounts for one addition and one multiplication. If only multiplications are considered, then $W_{LU} \approx \frac{N^3}{3}$, which is the operation count often quoted in the literature.

The advantage of LU-factorization is trivial. By breaking up the solution into two matrices, the user can solve multiple right hand sides with only minimal effort [38]. This feature combined with the low computational complexity and partial pivoting techniques makes LU-factorization extremely efficient.

A number of other methods could also be used to solve certain linear systems. Professional computing packages commonly include factorization routines such as QR and Cholesky, in addition to LU factorization [12]. There are also common numerical methods such as Gaussian Elimination or Gauss Jordan [20]. However, the dominant approach for general systems, which has been extensively studied in the literature, remains LU-factorization.

2.2 Iterative Solvers

Many sparse matrices use an iterative solver to find the variables. The iterative solvers are well suited to sparse matrices because they typically don't need to factorize the matrix like direct solvers. Instead, iterative solvers provide an initial guess and then refine that guess to a specified tolerance. This provides two key advantages. First, the refinement of the of the guess is much less intensive than factorization of the matrix. Second, the specified tolerance can reflect the nature of the problem being solved. If a high level of accuracy is needed, the tolerance will be small. If the accuracy required is less, then the amount of refinement can be reduced. This will provide a less computationally demanding solution.

The draw-back for iterative solvers is the amount of refinement required. While direct solvers will return an answer if the presented problem is not singular, an iterative solver may never reach the specified tolerance. The iterative solver will continue to run until it diverges or meets some other stopping criteria. Also, for a problem that demands a small tolerance the computational workload required for the iterative refinement may surpass the computation needed for a direct solver. Thus eleminating it's computational workload advantage.

2.3 Cramer's Rule

The proposed algorithm centers on the mathematically elegant Cramer's rule, which states that the components of the solution to a linear system in the form $Ax = b$ (where A is invertible) are given by

$$x_i = det(A_i(b))/det(A), \tag{2.1}$$

where $A_i(b)$ denotes the matrix A with its i^{th} column replaced by b [26]. As mentioned above, the matrix must of course be non-singular, otherwise a unique solution is not available.

Cramer's rule is cleverly based on the adjoint of a matrix A, $adj(A)$ [14]. if the inverse of a matrix can be computed by the equation

$$A^{-1} = \frac{adj(A)}{det(A)}$$

The adjoint of the matrix is simply the transpose of cofactors for each position in the matrix, with a cofactor given by

$$C_{i,j} = (-1)^{i+j} det(M_{i,j})$$

and $M_{i,j}$ being the minor corresponding to matrix entry $a_{i,j}$ where the i^{th} row and j^{th} column of A are eliminated. This gives the adjoint as

$$adj(A) = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{bmatrix}^T$$

It can then be easily seen why Cramer's rule provides a solution for a given unknown.

$$x = A^{-1}b = \frac{adj(A)}{det(A)}b = \frac{1}{det(A)} \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{bmatrix}^T \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Therefore:

$$x_i = \frac{C_{1i}b_1 + C_{2i}b_2 + \cdots + C_{ni}b_n}{det(A)}$$

Unfortunately, when computing large matrices, Cramer's rule is generally considered impractical. This stems from the fact that the determinant and cofactor values are calculated via minors. As the number of variables increases, the determinant computation becomes unwieldy [7]. The time complexity is widely quoted as $\mathcal{O}(N!)$, which would make it useless for any practical application when compared to a method like LU-factorization at $\mathcal{O}(N^3)$.

2.3.1 Accuracy Concerns

The other concern with Cramer's rule pertains to the numerical instability, which has received far less attention by the research community [26]. A simple example put forward in [32] suggests that Cramer's rule is unsatisfactory even for 2-by-2 systems, mainly because of round error difficulties. However, that argument heavily depends on the method for obtaining the determinants. If an accurate method for evaluating determinants is used then Cramer's rule can, in fact, be numerically stable. If greater precision is utilized only for the determinant calculations, Cramer's rule offers accuracy comparable to that of LU-factorization. In fact, a later paper [16] revisited the cited example and provided an example where Cramer's rule yielded a highly accurate answer while Gaussian elimination with pivoting a poor one.

2.3.2 Implementations in Literature

As stated earlier Cramer's rule is rarely deployed in actual computations, however there are a few proposed algorithms that utilize it. The main interest being in parallel implementations. One example is a proposed algorithm that creates a tree like structure and reduces the matrices in the tree with an elimination method akin to Gaussian Elimination [43]. The algorithm was named Parallel Cramer's Rule (PCR) and suggested that with $2n^2$ processors it could solve a linear system in n steps. The distinctive tree structure is shown in Figure 2.1.

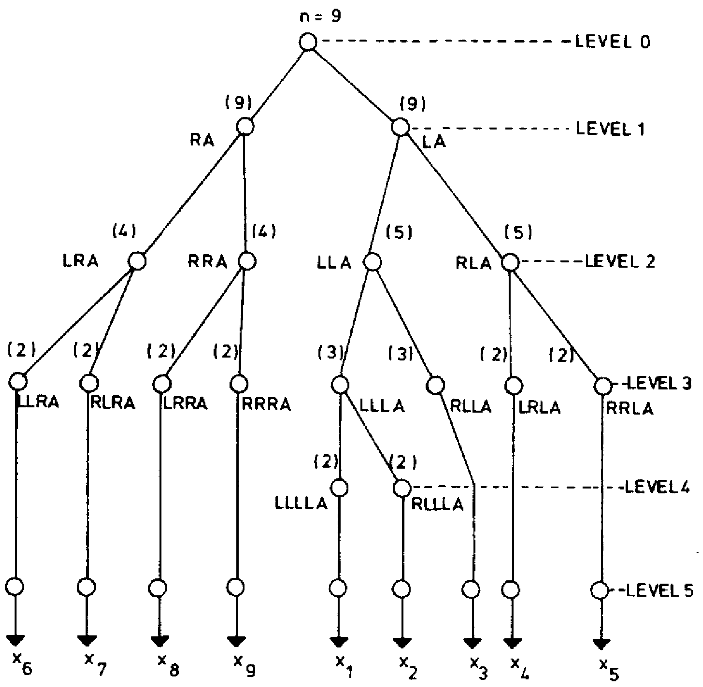


Figure 2.1: Parallel solution using Cramer's Rule. Source: M.K. SRIDHAR *A New Algorithm for Parallel Solution of Linear Equations, 1987*

The most intuitive approach to Cramer's rule is in combination with a condensation technique to solve the determinants. This provides a strait forward parallel implementation that can be distributed among numerous processors with little or no communication between them. A good example of this type of parallel solution can be found in a master's thesis at UT [5]. The problem with this implementation is that it has a computation complexity of $\mathcal{O}(N^4)$. Assuming N processors, each performs what a serial LU-factorization process could perform by itself as an $\mathcal{O}(N^3)$ algorithm.

The layout that has the most promise is a combination of a condensation technique and mirroring scheme. The research presented here is based largely on a paradigm first introduced by Arun Nagari, Itamar Arel and Ben Thompson [33, 34]. Although this algorithm differs in it's handling of the solution vector and the condensation scheme, it serves as the foundation for this dissertation.

2.4 Chio's Matrix Condensation

Chio's condensation [17] method reduces a matrix of order N to order $N-1$ when evaluating its determinant. As will be shown, repeating the procedure numerous times can reduce a large matrix to a size convenient for the application of Cramer's rule. Chio's pivotal condensation theorem is described as follows. Let $A = [a_{ij}]$ be an $N \times N$ matrix for which $a_{11} \neq 0$. Let D denote the matrix obtained by replacing each element a_{ij} by $\begin{vmatrix} a_{11} & a_{1j} \\ a_{i1} & a_{ij} \end{vmatrix}$, then it can be shown that $|A| = \frac{|D|}{a_{11}^{n-2}}$ [17].

Note that this process replaces each element in the original matrix with a 2×2 determinant consisting of the a_{11} element, the top value in the element's column, the first value in the element's row and the element being replaced. The calculated value of this 2×2 determinant replaces the initial $a_{i,j}$ with $a'_{i,j}$. The first column and first row are discarded, thereby reducing the original $N \times N$ matrix to a $(N-1) \times (N-1)$ matrix with an equivalent determinant. As an example, we consider the following 3×3 matrix:

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \text{ and its condensed form:}$$

$$\begin{vmatrix} 0 & 0 & 0 \\ 0 & (a_{11}a_{22} - a_{21}a_{12}) & (a_{11}a_{23} - a_{21}a_{13}) \\ 0 & (a_{11}a_{32} - a_{31}a_{12}) & (a_{11}a_{33} - a_{31}a_{13}) \end{vmatrix} = \begin{vmatrix} \times & \times & \times \\ \times & a'_{22} & a'_{23} \\ \times & a'_{32} & a'_{33} \end{vmatrix}$$

Obtaining each 2×2 determinant requires two multiplications and one subtraction. However, if the value of $a_{1,1}$ is one, then only a single multiplication is required. In the example above we note that $a_{1,1}$ is used in each element as a multiplier to the matrix element, for example, the equation for the matrix element in position $(2,2)$ is $a_{11}a_{22} - a_{21}a_{12}$. If in this situation $a_{11} = 1$, then the equation changes to $a_{22} - a_{21}a_{12}$. This holds true for every element in the matrix. Therefore for each condensation step k, if $a_{kk} = 1$ then $(N-k)^2$ multiplications are removed.

In order to guarantee $a_{kk} = 1$, an entire row or column must be divided by a_{kk}. This value would need to be stored because the determinant value calculated by Chio's condensation would be reduced by this factor. To find the true value at the end of the condensation, the calculated answer would need to be multiplied by each a_{kk} that was factored out. Multiplying all of these values over numerous condensation steps would result in an extremely large number that would exceed the floating point range of most computers. This is where the elegance of Cramer's rule is exploited. Cramer's rule determines each variable by a ratio of determinants, $x_i = det(A_i(b))/det(A)$. Given that both determinants are from the same condensation line, they both are reduced by the same a_{kk} values. The a_{kk} values factored out during Chio's condensation cancel during the application of Cramer's rule. This allows the algorithm to simply discard the a_{kk} values in the final computations. The actual determinant values are not correct, however the ratio evaluated at the core of Cramer's rule remains correct.

The cost of using Chio's condensation is equivalent to computing $(N-k)^2$ 2×2 determinants and $(N-k)$ divisions to create $a_{kk} = 1$. Hence, the computational effort required to reduce an $N \times N$ matrix to a 1×1 matrix is $\mathcal{O}\left(N^3\right)$, since

$$\sum_{k=1}^{N-1} 2(N-k)^2 + (N-k) = \frac{2N^3}{3} - \frac{N^2}{2} - \frac{N}{6} \sim \mathcal{O}(N^3). \qquad (2.2)$$

Combined with Cramer's rule, this process can yield the determinant to find a single variable. In order to find all N variables, this would need to be repeated N times, suggesting that the resulting work would amount to $\mathcal{O}(N^4)$.

2.5 Parallel Solutions

Highly efficient parallel algorithms balance computational complexity, memory constraints and communication overhead. An optimized algorithm for a single processor may not produce the best parallel implementation. The ability for a program to utilize multiple processors depends on numerous factors. Two of interest in this research are the ability to balance workload across resources and the communication requirements. An extremely fast algorithm on a single core may have difficulty evenly distributing workload across multiple processors. The algorithm might also require constant communication, causing a node to stall while waiting on data or instructions from other processors [37].

A software package for solving a system of linear equations, whether implemented on serial or parallel platforms, is based on an established mathematical framework. As in the serial implementations, standardized software packages typically use a form of LU-factorization. It is well studied and accepted to be stable and accurate within reasonable bounds. Moreover, it lends itself well to scalable parallel implementations [13]. Although there are other linear systems solvers available, with vast implementations, this research focuses on parallel LU-factorization as a baseline for comparison.

A parallel implementation of decomposing a matrix A into its LU factorization must address two main issues [28]:

1. Partitioning of the matrix A among the available processors
2. Organizing the code to efficiently compute the factorization at each node.

One of the main challenges with parallel factorization is dividing up the workload among the processors. Figure 2.2 shows the communication/computation pattern of a parallel LU-solver if data is distributed as rows. In the first stages of the algorithm nearly all processors have work. Processor P_1 broadcasts some shared data and all the other processors proceed with their local computations. Assuming a single row to each processor, the elimination of the first row in LU-factorization leaves the processor associated with that row idle as the algorithm continues. The processor, in this case P_1, sits idle because it does not have any work allocated to it. At the completion of step #2 another processor becomes idle and so on. Just before the algorithm completes, this scenario would have P_{n-1} processors sitting idle while a single processor finishes the computations. This format is inefficient [28].

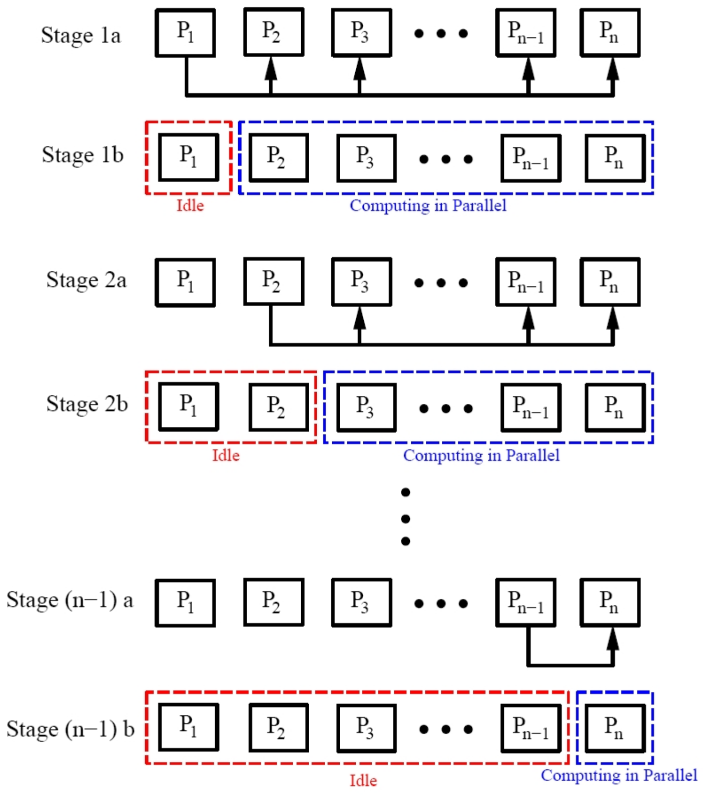


Figure 2.2: Schematic of communication (a) and computation (b) pattern when a matrix is partitioned such that each processor contains one row. source: Parallel Scientific Computing in C++ and MPI (Figure 9.2) [28]

The common solution to this problem is dividing up the matrix data in a non-intuitive manner. Instead of distributing groups of rows or columns to processors, the calling program assigns rows or columns of data to the processors in a cyclical manner. Consider an 8×8 matrix for allocation to four processors. Processor P_0 receives the first two columns, P_1 the next two, P_2 the next two, and P_3 the final two columns. After the first two condensation steps P_0 would sit idle.

In a cyclic distribution the columns would be assigned in a round-robin. The first column would be assigned to P_0, the next column P_1 and so on. Once all the processors have received their first column the assignment starts over. Processor P_0 would receive a second column and so forth. In this way each processor has data spread out across the entire matrix. As the factorization proceeds and rows and columns are eliminated, each processors still has portions of the matrix that will need computation. The left portion of Figure 2.3 shows a column cyclic distribution or a 1-D cyclic distribution.

2.5.1 2-D Cyclic Block Distribution

While the 1-D cyclic distribution provides good load balancing it has two draw backs. The first is that it limits the number of processors for a particular problem. For example, using nine processors for an 8×8 matrix leaves one processor without a column to factor. In typical parallel clusters this is rarely a concern since the problem sizes are much larger than the number of available nodes. Problems of small size are typically better solved with serial implementations.

The second problem cited in the literature is the theoretical communication minimum [25], [18], [9]. The communication required for a 1-D distribution to proceed with factorization is larger than the minimum lower bound on communication for factorization of a matrix. However, this assumes that true broadcast messages are not available in the parallel platform and will be further discussed in the following section.

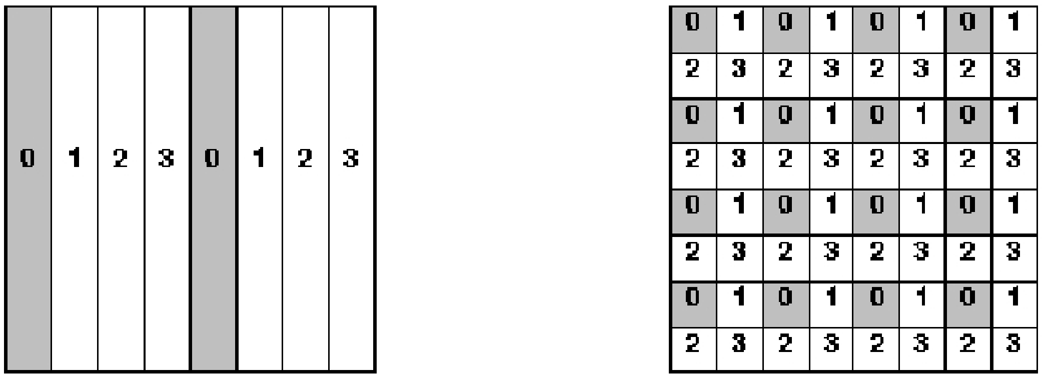


Figure 2.3: ScaLAPACK block Cyclic distribution example, Source: ScaLAPACK user's guide (Figure 4.4) [6]

In order to overcome these limitations parallel algorithms will distribute the matrix as small blocks of data. These blocks are distributed in a cyclic manner but will contain only a portion of rows and columns. A 2-D block-cyclic distribution of the matrix is demonstrated in the right portion of figure 2.3. In this case processor 0 is responsible for a number of small blocks over the entire matrix. Even as columns and rows are eliminated there are still portions of the matrix process 0 can work on.

The 2-D block-cyclic data distribution allows for good workload balancing but presents additional communication when pivoting occurs. Arrays of data must be passed between processors in order to physically pivot a row from one position to another. This generally occurs before further factorization can progress, which in most cases causes a natural delay for processors not involved in the pivoting.

The other concern is the conceptual complexity of distributing the data in this manner. Assigning portions of the matrix to a processor is largely left to the calling program and must match what the factorization routine expects. For casual users this can be cumbersome and confusing.

Once the data is distributed to the processors the actual computations involved can reuse the BLAS operations developed for serial versions. Since these are designed for blocks of data the 2-D distribution fits well with these highly optimized subroutines. This allows a parallel implementation to leverage the memory optimizations

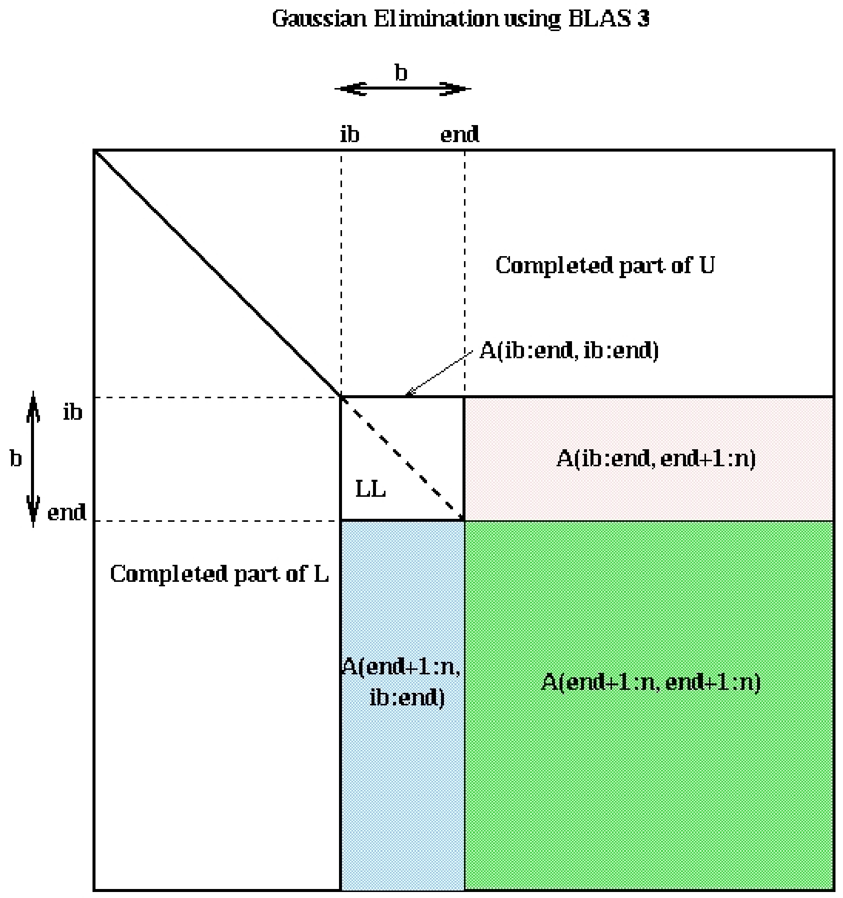


Figure 2.4: LU-decomposition block factorization

image source: James Demmel lecture notes (http://www.cs.berkeley.edu/˜demmel/cs267/lecture13X/lecture13X.html) and high-speed instructions from earlier work. In fact, the ScaLAPACK code calls the serial version of LU-factorization to perform factorization on each processor. Figure 2.4 depicts the overall factorization of a matrix where one processor would hold the $b \times b$ block of the matrix and can apply the serialized factorization to that particular piece.

2.5.2 Parallel Communication

As mentioned earlier, the literature considers 2-D block distribution to have less communication than 1-D distributions. This is an important consideration for this research. In order to complete a Gauss transformation on a square $n \times n$ matrix the communication is at least $2n(\sqrt{\alpha p} - 1)$ [25]. Assuming a specific problem size, this simplifies to some factor times $\sqrt{p}$. The calculated communication for a 1-D distribution is $(2/\gamma)n(p-1)$, where γ is some constant that satisfies $2n \leq \gamma n$. Clearly this equation holds a factor of p while the cited minimum states a factor of $\sqrt{p}$. However, this depends on the assumption that all communication is processor to processor and not broadcasted from one processor to many. If a highly reliable broadcast medium is available the overall communication requirement is drastically reduced [41].

The concept of broadcast communication is not clear in many implementations. For example, in the most common parallel message passing interface, the broadcast command is not a true broadcast. In the MPICH2 implementation of MPI, which was used for this research, the broadcast call is actually a combination of two communication algorithms [44]. For smaller broadcasts, less than 1,500 doubles, a binomial tree algorithm is used. This means that the first processor, P_0, sends to one other processor in step one. After that completes, both P_0 and P_1 send to another processor. At the completion of that step four processors have the data. All those processors then send their data to another processor. Each time the number of sending

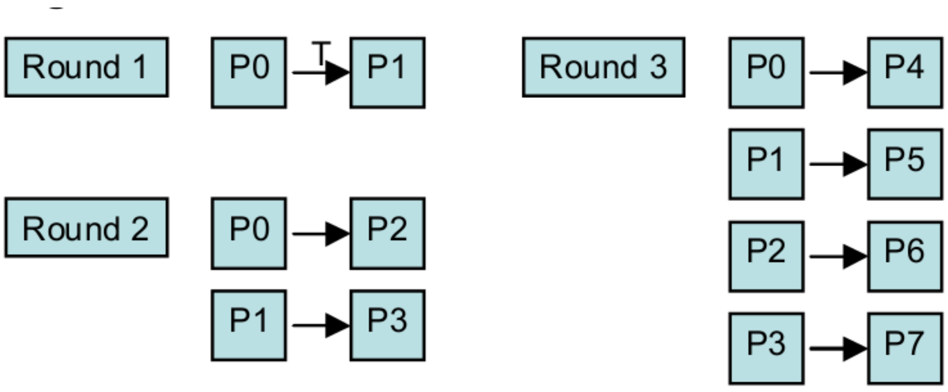


Figure 2.5: Representation of a binomial tree for broadcast communication [44]

processors doubles until all processors have the data. This is represented in Figure 2.5.

There are two obvious problems with this method. First it will take much longer than a single broadcast of the data because it must step through several rounds. Second, it will cause congestion if a bus type communication infrastructure is used. Near the end of the process, half the processors will attempt to send at roughly the same time. This could result in collisions and delays. The reason a simple broadcast is not done is fault tolerance. This method ensures that all processors receive the message.

When a very large message, greater than 1,500 doubles, is sent the MPICH2 code scatters pieces of the message to the processors and then calls an all-gather. All the different processors exchange data with each other so that at the end of the routine all the processors have the whole message. This suffers the similar challenges as the binomial tree.

An Ethernet network infrastructure supports broadcast and to realize reduced communication this functionality could be used. A true broadcast would reduce the communication time to something on the order of the message size. There would be no dependence on the number of processors. It would also eliminate congestion on the network, since only one processor is sending. The challenge would be a reliable broadcast medium and a method to identify when messages were lost or corrupted and recover or rebroadcast in a timely manner.

Assuming a true broadcast infrastructure the 1-D distribution requires less communication than the 2-D block distribution. If a matrix is divided into columns then a simple broadcast from one processor to all others would provide the data needed to factor the matrix. Whereas, the distribution of matrix blocks to various processors requires numerous one-to-one or one-to-subset communication patterns.

2.5.3 Parallel LU-factorization

Figure 2.6 shows a simplified communication pattern for LU-factorization using 2-D block distribution. The numbers in each block represent the processor for that portion of the matrix. Processor 00 must factorize its portion of the matrix and then send information to U01 and U02 as well as L10 and L20. Those nodes must then do a matrix-matrix operation and communicate those results to the nodes directly beneath or to their right. Not only does this require one-to-one communication but in some cases requires pauses while the A11 - A22 processors wait on row/column leads to complete their calculations [4].

As mentioned earlier, handling the pivot row also generates excess communication. If the matrix was distributed as rows or columns then the pivot could simply be communicated as a particular location and all nodes could adjust as required. Instead the actual data contained in the lead row/column must be communicated to the processor with the pivot. In some cases this may be the same node but in many case it will not. The pivoting information must be transferred for the entire length, as well. This means even the previously factorized portion of the matrix must be transferred.

Finally, LU-factorization requires forward and backward substitution to arrive at the solution. Since portions of the decomposed matrix are distributed across multiple processors, these substitutions require additional communication. For example, At

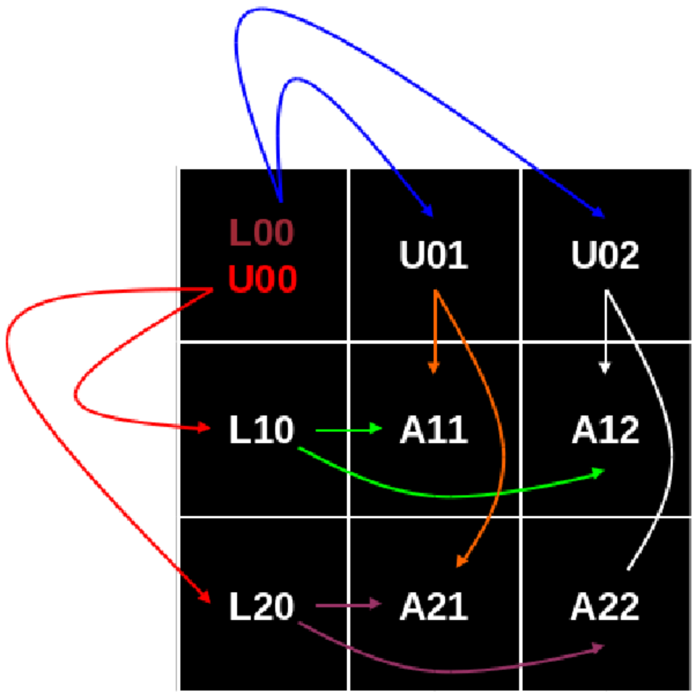


Figure 2.6: Simplified communication pattern of parallel LU-factorization

the outset of backward substitution every node in the domain sits idle waiting for one node to give the first few solution elements.

2.5.4 Pipelined Communication

Commonly referred to as send-ahead, pipelined communication is the concept of sending and receiving data while computations are being done. The goal of this technique is to hide the time required for communication by passing messages while calculations are proceeding. This requires two elements. First the computer's ability to handle communication independently of processing data, and second an algorithm that provides the necessary data in advance of the calculations.

This practice allows the processors to work asynchronously, where no process waits for the others to finish an iteration before beginning the following iteration [23]. Pipelined communication can be implemented in Figure 2.6 by having required data pass from node to node instead of originating from it's source for each message. For

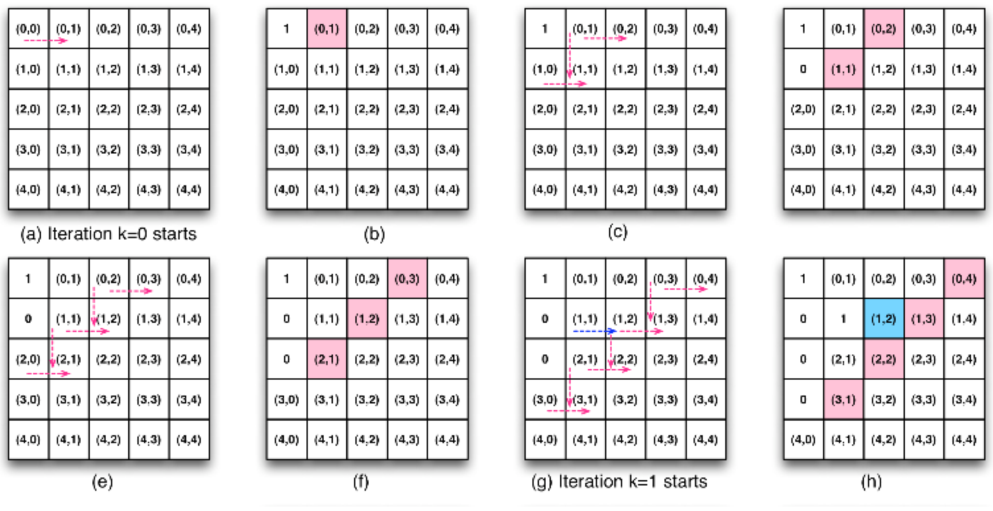


Figure 2.7: Pipelined Gaussian elimination [23]

example, processor 00 would send data to processor U01 but instead of sending to node U02 also it would continue with other work. processor U01 would then send to processor U02 while it also does it's computation workload. Once processor U01 finishes it's computation workload it will pass the data to processor A11, and then A11 would pass along the information to processor A21. While this shows no improvement for the initial condensation step it allows A11 to finish it's computations, begin it's factorization and then pass along the required data to A12 and A21 while they're still working on the previous condensation. On the second condensation step A12 and A21 already have the required data available and can proceed asynchronously. The condensation flows as a 'front' as seen by the progression of the pink squares in Figure 2.7 and overlaps communication with computation to hide the time required for communication.

Pivoting to improve accuracy and stability obviously hinders the ability for the processors to proceed asynchronously since all processors must find the pivot and communicate that correctly before continuing each condensation iteration.

2.6 Parallel Solvers

There are numerous parallel solvers available in the public domain; PETSc (Portable, Extensible Toolkit for Scientific Computation) from Argonne National Laboratory, PINEAPL (Parallel Industrial NumErical Applications and Portable Libraries) from the Numerical Algorithms Group Ltd (NAG), and the ParaSol project (CLRC Rutherford-Appleton Laboratory and collaborators) [3] are a few examples. The most prevalent, however, is ScaLAPACK (Scalable LAPACK) from the Innovative Computing Laboratory (ICL) at the University of Tennessee. This package provides optimized routines for solving linear systems and will thus can serve as a measuring stick for this research.

2.6.1 ScaLAPACK

ScaLAPACK [6] is based on small optimized subroutines called PBLAS (Parallel Basic Linear Algebra Subroutine). These subroutines provide methods with highly localized memory access to complete basic linear algebra operations. The ScaLAPACK package then calls these PBLAS routines to compute more complex methods such as LU-factorization.

As with most parallel packages, ScaLAPACK provides inter-process communication via MPI (Message Passing interface), however another software layer is placed between ScaLAPACK and MPI to provide something more linear algebra friendly. The BLACS library (Basic Linear Algebra Communication Subprograms) provides a communication interface that supports linear algebra type messages as well as a platform independent connection.

ScaLAPACK grew from the LAPACK project and in development of the software an effort was made to keep the interfaces as similar as possible. This leads to parallel code that looks almost exactly like it's serial version [8] and makes conversion for LAPACK users to ScaLAPACK straightforward. Unfortunately, it also carries the somewhat cryptic naming conventions and structure that came out of the original FORTRAN coding.

Chapter 3

Matrix Condensation-based Realization of Cramer's Rule

Fortunately, Cramer's rule can be realized in far lower complexity than the typically quoted $\mathcal{O}(N!)$. The complexity of Cramer's rule depends predominantly on the determinant calculations. If the determinants are calculated via minors the factorial complexity holds. In an effort to overcome this limitation, a matrix condensation technique and clever mirroring of the matrix can reduce the size of the original matrix to one that may be solved efficiently and quickly. As a result, Cramer's Rule becomes an $\mathcal{O}(N^3)$ process, which is similar to LU-factorization.

3.1 Matrix Mirroring

The overarching goal of the proposed approach is to obtain an algorithm with $\mathcal{O}\left(N^3\right)$ complexity and low storage requirement overhead. As discussed earlier, Chio's condensation and Cramer's rule provide an elegant solution with $\mathcal{O}\left(N^4\right)$ complexity. In order to retain $\mathcal{O}\left(N^3\right)$ computational complexity, it is necessary to reuse some of the intermediate calculations performed by prior condensation steps. This is achieved

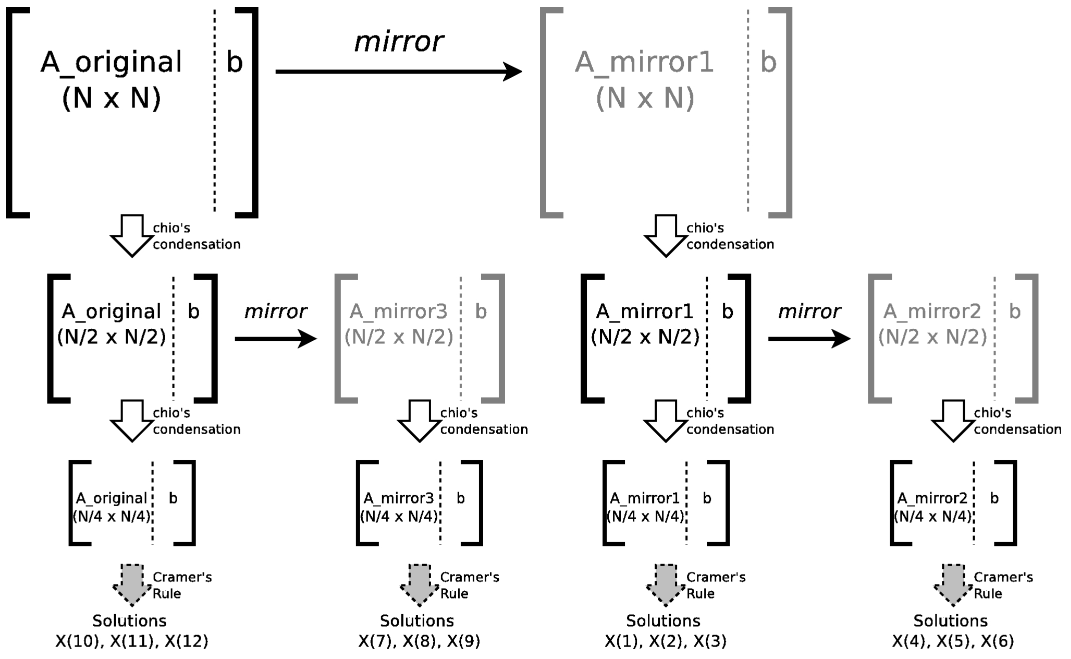


Figure 3.1: Tree architecture applied to solving a linear system using the proposed algorithm

by constructing a binary, tree-based data flow in which the algorithm mirrors the matrix at critical points during the condensation process, as detailed next.

A matrix A and a vector of constants b are passed as arguments to the algorithm. The latter begins by appending b to A creating an augmented matrix. All calculations performed on this matrix are also performed on b. Normal utilization of Cramer's rule would involve substitution of the column corresponding to a variable with the vector b, however the proposed algorithm introduces a delay in such substitution such that multiple variables can be solved utilizing one line of Chio's condensation. In order to delay the column replacement, b must be subject to the same condensation manipulations that would occur had it already been in place. This serves as the motivation for appending b to the matrix during condensation.

The condensation method removes information associated with discarded columns, which suggests that the variables associated with those columns cannot be computed once condensed. For this reason, a mirror of the matrix is created each time the matrix size is halved. The mirrored matrix is identical to the original except the order of its columns is reversed. For example, the first and last column are swapped, the second and second to last column are swapped, and so on. A simple 3x3 matrix mirroring operation would be:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \rightarrow mirrored \rightarrow \begin{vmatrix} a_{13} & a_{12} & -a_{11} \\ a_{23} & a_{22} & -a_{21} \\ a_{33} & a_{32} & -a_{31} \end{vmatrix}$$

In the example above, the third column of the mirror is negated, which is performed in order to retain the correct value of the determinant. Any exchange of columns or rows requires the negation of one to preserve the correct determinant value. As discussed in more detail below, this negation is not necessary to arrive at the correct answer, but is applied for consistency.

Following the mirroring, each matrix is assigned half of the variables. The original matrix can solve for the latter half while the mirrored matrix solves for the first half of the variables. In the example above, there are three variables: x_1, x_2, x_3. The original matrix could solve for x_2, x_3, and the mirrored matrix would provide x_1. Each matrix uses condensation to yield a reduced matrix with size at least equal to the number of variables it's responsible for. For the case of a 3×3 matrix, we have the pair of matrices:

$$\underset{original : x_2, x_3}{\begin{vmatrix} \times & \times & \times \\ \times & a'_{22} & a'_{23} \\ \times & a'_{32} & a'_{33} \end{vmatrix}} \quad \underset{mirrored : x_1}{\begin{vmatrix} \times & \times & \times \\ \times & a'_{22} & a'_{21} \\ \times & a'_{32} & a'_{31} \end{vmatrix}}$$

Once this stage is reduced, the algorithm either solves for the variables using Cramer's rule or mirrors the matrix and continues with further condensation. A process flow depicting the proposed framework is illustrated in Figure 3.1.

3.2 Extended Condensation

Chio's condensation reduces the matrix by one order per repetition. Such an operation is referred to here as a condensation step of size one. It's possible to reduce the matrix by more than one order during each step. Carrying out the condensation one stage further, with leading pivot $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$ (assumed to be non-zero), we have [2]

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}^{-1} \times \begin{vmatrix} \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \end{vmatrix} \\ \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{41} & a_{42} & a_{43} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{24} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} \end{vmatrix} \tag{3.1}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}^{-1} \times \begin{vmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & a'_{33} & a'_{34} \\ \times & \times & a'_{43} & a'_{44} \end{vmatrix} . \tag{3.2}$$

In this case, each of the matrix elements $\{a_{33}, a_{34}, a_{43}, a_{44}\}$ are replaced by a 3×3 determinant instead of a 2×2 determinant. This delivers a drastic reduction in the number of repetitions needed to condense a matrix. Moreover, a portion of each minor is repeated, namely the $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$ component. Such calculation can be performed once and then reused multiple times during the condensation process. Letting M denote the size of the condensation, and using the example above, $M = 2$ while for the basic Chio's condensation technique $M = 1$. As an example, a 6×6 matrix could be reduced to a 2×2 matrix in only two condensation steps, whereas it would take four traversals of the matrix to arrive at a 2×2 matrix if $M = 1$. The trade-off is a larger determinant to calculate for each condensed matrix element. Instead of having 2×2 determinants to calculate, 3×3 determinants are needed, suggesting a net gain of zero.

However, the advantage of this formulation is that larger determinants in the same row or column share a larger number of the minors. The determinant minors can be calculated at the beginning of each row and then reused for every element in that row. This reduces the number of operations required for each element with a small penalty at the outset of each row. In a practical computer implementation, this also involves fewer memory access operations, thus resulting in higher overall execution speed.

Pivoting in the case of $M > 1$ requires identifying a lead determinant that is not small. As with pivoting for LU-factorization, ideally the largest possible lead determinant would be moved into the top left portion of the matrix. Unfortunately, this severely compromises the computational complexity, since an exhaustive search for the largest lead determinant is impractical. Instead, a heuristic method should be employed to select a relatively large lead determinant when compared to the alternatives.

The pseudocode in algorithm 1 details a basic implementation of the proposed algorithm using extended condensation. The number of rows and columns condensed during each pass of the algorithm is represented by the variable M, referred to earlier as the condensation step size. The original matrix is passed to the algorithm along with the right-hand side vector in A, which is an $N \times (N+1)$ matrix. The $mirrorsize$ variable represents the number of variables a particular matrix solves for. In other words, it reflects the smallest size that a matrix can be condensed to before it must be mirrored. If original matrix, A, has a $mirrorsize = N$ it solves for all N variables. It should be noted that this will result in bypassing the while loop completely at the initial call of the algorithm since mirroring must occur before any condensation is done. The first mirror will have $mirrorsize = \frac{N}{2}$, since it only has to solve for

Algorithm 1 Extended Condensation

```
{A[][] = current matrix, N = size of current matrix}
{mirrorsize = variables for this matrix to solve for}
while (N-M) > mirrorsize do
  lead_determinant = CalculateMinor(A[][], M+1, M+1)
  if lead_determinant = 0 then then
    return(error)
  end if
  A[1:N][1] = A[1:N][1] / lead_determinant;
  {calculate the minors that are common}
  for i = 1 to M step by 1 do
    for j = 1 to M step by 1 do
      {each minor will exclude one row & col}
      reusableminor[i][j] = CalculateMinor(A[][], i, j);
    end for
  end for
  for row = (M+1) to (N+1) step by 1 do
    {find the lead minors for this row}
    for i = 1 to M step by 1 do
      Set leadminor[i] = 0;
      for j = 1 to M step by 1 do
        leadminor[i] =leadminor[i] + $(-1)^{j-1}$A[row][j] $\times$ reusableminor[i][j]
      end for
    end for
    {Core Loop; find the MxM determinant for each A[][] item}
    for col = (M+1) to (N+1) step by 1 do
      for i = 1 to M step by 1 do
        {calculate MxM determinant}
        A[row][col] =A[row][col]+ $(-1)^{j}$leadminor[i]$\times$A[i][col]
      end for
    end for
  end for
  {Reduce matrix size by condensation step size}
  N = N - M;
end while
if N has reached Cramer's rule size (typically 4) then
  {solve for the subset of variables assigned}
  x[] = CramersRule(A[][]);
else {recursive call to continue condensation}
  A_mirror[][] = Mirror(A[][])
  Recurisvely call Algorithm (A_mirror[][], N, mirrorsize/2)
  Recursively call Algorithm (A[][], N, mirrorsize/2)
end if
```

half of the variables. After this mirror has been created, the algorithm will begin the condensation identified within the while loop.

Three external functions assist the psuedocode: $CalculateMinor$, $CramersRule$ and $Mirror$. $CalculateMinor$ finds an $M \times M$ determinant from the matrix passed as an argument. The two additional arguments passed identify the row and column that should be excluded from the determinant calculation. For example, $CalculateMinor(A[][], 2, 3)$ would find the $M \times M$ determinant from the top left portion of matrix A[][], whereby row 2 and column 3 are excluded. The method would return the top left $M \times M$ determinant of A[][] without excluding any rows or columns by calling $CalculateMinor(A[][], M+1, M+1)$, since an $M+1$ value excludes a column beyond the M rows and M columns used to calculate the determinant. This is used in the algorithm to find the lead determinant at each condensation step. When $M = 1$, the method simply returns the value at $A[1][1]$.

The $CramersRule$ method solves for the variables associated with that particular matrix using Cramer's rule. The method replaces a particular column with the condensed solution vector, b, finds the determinant, and divides it by the determinant of the condensed matrix to find each variable. The method then replaces the other columns until all variables for that particular leaf are calculated. The $Mirror$ function creates a mirror of the given matrix as described in section 3.1.

The arrays labeled *reusable_minor* and *lead_minor* maintain the pre-calculated values discussed earlier. The array *reusable_minor* is populated once per condensation step and holds M^2 minors that will be used at the outset of each row. The latter will then populate the *lead_minor* array from those reusable minors. The *lead_minor* array holds the M minors needed for each matrix element in a row. Hence, the *lead_minor* array will be repopulated $N - M$ times per condensation step.

The algorithm divides the entire first column by the top left $M \times M$ determinant value. This is performed for the same reason that the first column of the matrix was divided by the a_{11} element in the original Chio's condensation formulation. Dividing the first row by that determinant value causes the lead determinant calculation to retain a value of one during the condensation. This results in a '1' being multiplied by the $a_{i,j}$ element at the beginning of every calculation, thus saving one multiplication operation per each element.

3.3 Computation Complexity

As illustrated in the pseudocode, the core loop of the algorithm involves the calculation of the $M \times M$ determinants for each element of the matrix during condensation. Within the algorithm, each $M \times M$ determinant requires M multiplications and M additions/subtractions. Normally, this would necessitate the standard computational workload to calculate a determinant, i.e. $\frac{2}{3}M^3$, using a method such as Gaussian elimination. However, the reuse of the determinant minors described earlier reduces the effort to $2M$ operations within the core loop.

An additional workload outside the core loop is required since M^2 minors must be pre-calculated before Chio's condensation commences. Assuming the same $\frac{2}{3}M^3$ workload using Gaussian elimination to find a determinant and repetition of this at each of the $\frac{N}{M}$ condensation steps, yields an overhead of

$$\frac{N}{M} \times M^2 \times \frac{2}{3}M^3 = \frac{2M^4N}{3}. \tag{3.3}$$

In situations where $M \ll N$, this effort is insignificant, although with larger values of M relative to N, it becomes non-negligible.

The optimal size of M occurs when there's a balance between pre-calculation done outside the core loop and the savings during each iteration. In order to reuse intermediate calculation results, a number of determinant minors must be evaluated in advance of each condensation step. These reusable minors save time during the core loops of the algorithm, but do not utilize the most efficient method. If a large number of minors are required prior to each condensation, their additional computation annuls the savings obtained within the core iterations.

The optimal size of M is thus calculated by setting the derivative of the full complexity formula, $\frac{2}{3}N^3 - MN^2 + \frac{N^2}{M} + \frac{2}{3}M^4N + M^2N$, with respect to M, to zero. This reduces to $\frac{8}{3}M^5 + 2M^3 = NM^2 + N$, suggesting an optimal value of $M \sqrt[3]{\frac{3}{8}N}$. As an example, the optimal point for a 1000×1000 matrix is ≈ 7.22. Empirical results indicate that the shortest execution time for a 1000×1000 matrix was achieved when M=8, supporting the theoretical result.

In order to condense a matrix from $N \times N$ to $(N - M) \times (N - M)$, the core calculation is repeated $(N - M)^2$ times. The algorithm requires N/M condensation steps to reduce the matrix completely and solve using Cramer's rule. In terms of operations, this equates to

$$\begin{aligned}\gamma =&\sum_{k=1}^{N/M} 2M(N - kM)^2 \\ =&2M\sum_{k=1}^{N/M}(N^2 - 2NMk + M^2k^2) \\ =&2M\left(\frac{N}{M}N^2 - 2NM\left(\frac{\frac{N}{M}\left(\frac{N}{M}+1\right)}{2}\right) + M^2\left(\frac{\frac{N}{M}\left(\frac{N}{M}+1\right)\left(\frac{2N}{M}+1\right)}{6}\right)\right) \\ =&\frac{2}{3}N^3 - MN^2 + \frac{M^2N}{3}. \end{aligned} \tag{3.4}$$

resulting in a computational complexity, γ, of $\frac{2}{3}N^3$ to obtain a single variable solution.

Mirroring occurs with the initial matrix and then each time a matrix is reduced in half. An $N \times N$ matrix is mirrored when it reaches the size of $\frac{N}{2} \times \frac{N}{2}$. Once the matrix is mirrored, there is double the work. In other words, two $\frac{N}{2} \times \frac{N}{2}$ matrices each require a condensation, where previously there was only one. However, the amount of work for two matrices of half the size is much lower than that of one $N \times N$ matrix, which avoids the $O(N^4)$ growth pattern in computations. This is due to the $O(N^3)$ nature of the condensation process.

Since mirroring occurs each time the matrix is reduced to half, $log_2 N$ matrices remain when the algorithm concludes. The work associated with each of these mirrored matrices needs to be included in the overall computation load estimate. The addition of the mirrors follows a geometric series resulting in roughly 2.5 times the original workload, which leads to a computational complexity of $\frac{5}{3}N^3$ when ignoring the lower order terms.

The full computational complexity is the combination of the work involved in reducing the original matrix, γ, and that of reducing the mirrors generated by the algorithm. Hence, the total complexity can be expressed as follows:

$$\gamma + \sum_{k=0}^{log_2 N} 2^k \left(\frac{2}{3} \left(\frac{N}{2^k} \right)^3 - M \left(\frac{N}{2^k} \right)^2 + \frac{M^2}{3} \left(\frac{N}{2^k} \right) \right). \tag{3.5}$$

The latter summation can be simplified using the geometric series equivalence $\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$ [1], which when ignoring the lower order terms reduces to:

$$\gamma + \frac{8}{9}N^3 = \frac{14N^3}{9} \approx \frac{5}{3}N^3 \tag{3.6}$$

3.4 Mirroring Considerations and Related Memory Requirements

Equation (3.6) expresses the computational complexity assuming a split into two matrices each time the algorithm performs mirroring. One may consider a scenario in which the algorithm creates more than two matrices during each mirroring step. In the general case, the computational complexity is given by

$$\gamma + \sum_{k=0}^{log_S N} (S-1)S^k \left(\frac{2}{3} \left(\frac{N}{S^k} \right)^3 - M \left(\frac{N}{S^k} \right)^2 + \frac{M^2}{3} \left(\frac{N}{S^k} \right) \right) \tag{3.7}$$

where S denotes the number of splits. When lower order terms are ignored, this yields

$$\gamma + \frac{2S^2}{3(S+1)}N^3. \tag{3.8}$$

As S increases the complexity clearly grows. The optimal number of splits is thus two, since that represents the smallest value of S that can still solve for all variables. Additional splits could facilitate more work in parallel, however they would generate significantly greater overall workload.

The memory requirement of the algorithm is $2 \times (N+1)^2$, reflecting the need for sufficient space for the original matrix and the first mirror. The rest of the algorithm can reuse that memory space. Since the memory requirement is double the amount required by typical LU-factorization implementations and similar to LU-factorization, the original matrix is overwritten during calculations.

3.5 Solution Subsets

One benefit of Cramer's rule is the ability to target specific variables. If there's a large linear system but only a subset of the variables need to be solved for, the algorithm can easily focus on those and thus vastly reduce the computational workload. For example, if only one unknown is needed then the computational workload reduces from $\frac{5}{3}N^3$ to $\frac{2}{3}N^3$, which is the same workload that of LU-factorization.

As explained in section 3.1, Cramer's rule algorithm mirrors initially so that columns are retained for all the variables even after the application of Chio's condensation. The mirrored matrix solves for the first half of the variables and the original matrix solves for the last half of variables. This creates two initial matrices for condensation. If there are variables that the algorithm does not need to solve for, the algorithm can delay the initial mirroring thus reducing the workload. The real benefit being a reduction at the outset of the condensation when the matrix is it's largest and requires the most computation.

Chapter 4

Algorithm Accuracy and Stability

The stability properties of the proposed algorithm are very similar to those of Gaussian Elimination techniques. Both schemes are mathematically accurate yet subject to truncation and rounding errors. As with LU-factorization, if these errors are not accounted for, the algorithm returns poor accuracy. LU-factorization utilizes partial or complete pivoting to minimize truncation errors. As will be shown, the proposed algorithm employs a similar technique.

Each element during a condensation is affected by the lead determinant and the 'lead minors' discussed earlier. In order to avoid truncation errors, these values should go from largest on the first condensation to smallest on the last condensation. This avoids a situation where matrix values are drastically reduced, causing truncation, and then significantly enlarged later, magnifying the truncation error. The easiest method to avoid this problem is by moving the rows that would generate the largest determinant value to the lead rows before each condensation. This ensures the largest determinant values available are used in each step.

Once the matrix is rearranged with the largest determinant in the lead, normalization occurs. Each lead value is divided by the determinant value, resulting in the lead determinant equaling unity. This not only reduces the number of floating point calculations but serves to normalize the matrix.

4.1 Backward Error Analysis

The backward stability analysis of the algorithm yields results similar to LU-factorization. This coupled with empirical findings provides evidence that the algorithm yields accuracy comparable to that of LU-factorization. As with any computer calculations, rounding errors affect the accuracy of the algorithm's solution. Backward stability analysis shows that the solution provided is the exact solution to a slightly perturbed problem. The typical notation for this concept is

$$(A + F)\hat{x} = b + \delta b. \tag{4.1}$$

Where A denotes the original matrix, b gives the constant values, and $\hat{x}$ gives the solution calculated using the algorithm. F represents the adjustments to A, and δb the adjustment to b that provides a problem that would result in the calculated solution if exact arithmetic was possible. In this analysis the simplest case is given, namely where the algorithm uses $M = 1$.

In the first stage of condensation, $A^{(2)}$ is computed from $A^{(1)}$, which is the original matrix. It should be noted that each condensation step also incurs error on the right-hand side due to the algorithm carrying those values along during reduction. This error must also be accounted for in each step, so in the first stage of condensation, $b^{(2)}$ is computed from $b^{(1)}$ just as A.

Before Chio's pivotal condensation occurs, the largest determinant is moved into the lead position. Since $M = 1$, the determinant is simply the value of the element. This greatly simplifies the conceptual nature for conveying this analysis. Normally $M \times M$ determinants would need to be calculated, and then all the rows comprising the largest determinant moved into the lead. Here, the row with the largest lead value $a_{i,1}$ is moved to row one, followed by each element in column one being divided by a_{kk}. This normalizes the matrix so that the absolute values of all row leads are smaller or equal to one, such that

$$a_{k,j} = \frac{a_{kj}}{a_{kk}}(1+\eta_{kj}), \quad where\ \eta_{kj} \leq \beta^{-t+1} \tag{4.2}$$

In this case, β^{-t+1} is the base number system used for calculation with t digits. This is equivalent to machine epsilon, ϵ. The computed elements $a_{ij}^{(2)}$are derived from this basic equation $a_{ij}^{(2)} = a_{ij} - a_{ik} \times a_{kj}$ when the error analysis is then added

$$a_{ij}^{(2)} = \left[a_{ij}^{(1)} - a_{ik} \times a_{kj} \times \left(1+\gamma_{ij}^{(1)}\right)\right]\left(1+\alpha_{ij}^{(1)}\right) \quad |\gamma_{ij}^{(1)}| \leq \beta^{-t+1}\ and\ |\alpha_{ij}^{(1)}| \leq \beta^{-t+1} \tag{4.3}$$

$$a_{ij}^{(2)} = a_{ij}^{(1)} - a_{ik} \times a_{kj} + e_{ij}^{(1)} \tag{4.4}$$

where

$$e_{ij}^{(1)} = \frac{a_{ij}^{(2)} \times \alpha_{ij}^{(1)}}{\left(1+\alpha_{ij}^{(1)}\right)} - a_{ik} \times a_{kj} \times \gamma_{ij}^{(1)} \quad ij = 2, ..., n. \tag{4.5}$$

This then provides the elements for $E^{(1)}$ such that $A + E^{(1)}$ provides the matrix that would condense to $A^{(2)}$ with exact arithmetic. The lead column is given by $e_{ij}^{(1)} = a_{ik} \times \eta_{kj}$. This follows for each condensation step $A^{(k+1)} = A^{(k)} + E^{(k)}$ and similarly for the right-hand side, $b^{(k+1)} = b^{(k)} + E_b^{(k)}$, where E includes an additional column to capture the error incurred on b. In this case, the E matrix will capture the variability represented by δb found in equation 4.1. If taken through all steps of condensation, then $E = E^{(1)} + ... + E^{(n-1)}$, giving

$$(A+E)\hat{x} = b. \tag{4.6}$$

Bounds on E need evaluation, since this controls how different the matrix used for computation is from the original matrix. It's important to note that, due to the use of Cramer's rule, the algorithm can normalize a matrix and simply discard the value used to normalize. Cramer's rule is a ratio of values so as long as both values are divided by the same number the magnitude of that number is unimportant. This is a crucial attribute, since needing to retain and later use these values would cause instability.

Consider $a = max|a_{ij}|$, $g = \frac{1}{a}max|a_{ij}^{(k)}|$ and equation (9). If these are combined along with the knowledge that $|a_{1j}| \leq 1$, the following is obtained

$$|e_{ij}^{(k)}| \leq \frac{\beta^{-t+1}}{1-\beta^{-t+1}}|a_{ij}^{(k+1)}| + \beta^{-t+1} \times |a_{ij}^{(k)}| \leq \frac{2}{1-\beta^{-t+1}}ag\beta^{-t+1} \tag{4.7}$$

In essence the E matrix yields the following

$$|E| \leq ag\Upsilon \left\{ \begin{bmatrix} 0 & 0 & ... & ... & 0 \\ \beta^{-t+1} & 2 & ... & ... & 2 \\ : & : & & & : \\ : & : & & & : \\ \beta^{-t+1} & 2 & ... & ... & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & \beta^{-t+1} & 2 & ... & 2 \\ : & : & : & & : \\ 0 & \beta^{-t+1} & 2 & ... & 2 \end{bmatrix} + ... \right\} \tag{4.8}$$

where $\Upsilon = \frac{\beta^{-t+1}}{(1-\beta^{-t+1})}$, such that

$$|E| \leq ag\Upsilon \begin{bmatrix} 0 & 0 & ... & ... & 0 \\ \beta^{-t+1} & 2+\beta^{-t+1} & ... & ... & 2 \\ : & : & 4+\beta^{-t+1} & ... & 4 \\ : & : & : & & : \\ \beta^{-t+1} & 2+\beta^{-t+1} & 4+\beta^{-t+1} & ... & 2(n) \end{bmatrix}. \tag{4.9}$$

The bottom row of the matrix clearly provides the largest possible value, whereby the summation is roughly n^2. When combined with the other factors, it yields the equality $\|E_\infty\| = 2n^2ag\frac{\beta^{-t+1}}{1-\beta^{-t+1}}$. If it's assumed that $1-\beta^{-t+1} \approx 1$ and a is simply a scaling factor of the matrix, two values of interest are left: n^2 and the growth factor g. The growth factor is the element that has the greatest impact on the overall value, since it provides a measure of the increase in value over numerous condensation steps. Fortunately, this value is bound because all multipliers are due to the pivoting and the division performed before each condensation, such that

$$max|a_{ij}^{(k+1)}| = max|a_{ij}^{(k)} - a_{ik} \times a_{kj}| \leq 2 \times max|a_{ij}^{(k)}|. \tag{4.10}$$

The value of $a_{ij}^{(k)}$ is at most the largest value in the matrix. The value of $a_{ik} \times a_{kj}$ is also at most the largest value in the matrix. Since a_{ik} is the row lead, it's guaranteed to be one or less. The value of a_{kj} could possibly be the largest value in the matrix. Therefore, the greatest value that could result from the equation $a_{ij}^{(k)} - a_{ik} \times a_{kj}$ is twice the maximum value in the matrix or $2max|a_{ij}^{(k)}|$. This can then repeat at most n times, which results in a growth factor of 2^n. The growth factor given for LU-factorization with partial pivoting in the literature is $g \leq 2^{n-1}$ [36]. The slight difference being that this algorithm computes a solution directly, whereas LU-factorization analysis must still employ forward and backward substitution to compute a solution vector. As with LU-factorization, it can be seen that generally the growth rate will less than double each step. In fact, the values tend to cancel each other leaving the growth rate around 1 in actual usage.

Mirroring does not affect the stability analysis of the algorithm. The matrices that are used to calculate the answers may have been mirrored numerous times. Since no calculations take place during the mirroring, and it does not introduce an additional condensation step, the mirroring has no bearing on the accuracy.

4.2 Backward Error Measurements

One of the most beneficial attributes of LU-factorization is that, although it has a growth rate of 2^{n-1}, in practice it generally remains stable. This can be demonstrated by relating the relative residual to the relative change in the matrix, giving the following inequality:

Matrix Size	$n\epsilon_{machine}$	Cramer's $\frac{\|b-A\hat{x}\|}{\|A\|\cdot\|\hat{x}\|}$	Matlab $\frac{\|b-A\hat{x}\|}{\|A\|\cdot\|\hat{x}\|}$
1000 x 1000	2.22E-13	5.93E-14	8.05E-16
2000 x 2000	4.44E-13	5.42E-14	1.04E-15
3000 x 3000	6.66E-13	9.62E-14	1.95E-15
4000 x 4000	8.88E-13	3.32E-13	2.49E-15
5000 x 5000	1.11E-12	8.12E-14	3.05E-15
6000 x 6000	1.33E-12	5.52E-14	3.35E-15
7000 x 7000	1.55E-12	7.46E-14	3.55E-15
8000 x 8000	1.78E-12	8.12E-14	4.28E-15

Table 4.1: Relative residual measurements for Cramer's rule algorithm

$$\frac{\|b - A\hat{x}\|}{\|A\| \cdot \|\hat{x}\|} \leq \frac{\|E\|}{\|A\|} . \tag{4.11}$$

The symbol $\|\hat{x}\|$ represents the norm of the calculated solution vector. When the residual found from this solution set is divided by the norm of the original matrix multiplied by the norm of the solution set, an estimate is produced of how close the solved problem is to the original problem. If an algorithm produces a solution to a problem that is very close to the original problem then the algorithm is considered stable. A reasonable expectation for how close the solved and given problems should be is expressed as [24].

$$\frac{\|E\|}{\|A\|} \approx n\epsilon_{machine} \tag{4.12}$$

A pragmatic value of $\epsilon_{machine} \approx 2.2E-16$ reflects the smallest value the hardware can accurately support, and n represents the size of the linear system. Table 4.1 shows this relative residual calculations when using Cramer's Rule in comparison to those obtained with Matlab and for the target values given by equation (4.12). The infinite norm is used for all norm calculations and Cramer's Rule used a condensation step size (M) of 8. As shown in table 4.1, both Matlab's implementation of LU-factorization and Cramer's Rule deliver results below the target level to suggest a stable algorithm for the test matrices considered. The latter were created by populating the matrices

Matrix Size	$\kappa(A)$	Matlab $\|x-\hat{x}\|_\infty$	GSL $\|x-\hat{x}\|_\infty$	Avg Matlab	Avg GSL
1000 x 1000	506930	2.39E-9	1.93E-10	1.03E-10	5.38E-12
2000 x 2000	790345	4.52E-9	5.36E-9	1.01E-10	7.27E-12
3000 x 3000	1540152	1.95E-8	1.84E-8	1.12E-10	2.09E-11
4000 x 4000	12760599	4.81E-8	5.62E-8	1.43E-10	7.91E-11
5000 x 5000	765786	2.92E-8	4.39E-8	1.18E-10	3.46E-11
6000 x 6000	1499430	8.67E-8	8.70E-8	1.37E-10	6.04E-11
7000 x 7000	3488010	9.92E-8	8.95E-8	1.27E-10	5.15E-11
8000 x 8000	8154020	9.09E-8	9.43E-8	1.86E-10	7.85E-11

Table 4.2: Cramer's rule algorithm relative error when compared to Matlab and GSL solution sets

with random values between -5 and 5 using the standard C language random number generator. Results produced by the proposed algorithm for these matrices were measured over numerous trials.

4.3 Forward Error Measurements

The forward error is typically defined as the relative difference between the true values and the calculated ones. Here, the actual answers are generated by Matlab and GSL (GNU scientific library). The solution vector provided by the algorithm was compared to those given by both software packages. Table 4.2 details the observed relative difference between the software packages and the solutions provided by the proposed algorithm.

Matlab includes a number of test matrices in a matrix gallery that were used for further comparison. In particular, a set of dense matrices from this gallery were selected. Each type had four samples of 1000x1000 matrices. In many cases, Cramer's Rule resorted to a condensation size of one ($M = 1$) for improved accuracy. The relative residuals were then calculated in the manner shown in section 3.2. Table 4.3 provides a summary of those findings.

Matrix Type	Cramer's $\frac{\|b-A\hat{x}\|}{\|A\|\cdot\|\hat{x}\|}$	Matlab $\frac{\|b-A\hat{x}\|}{\|A\|\cdot\|\hat{x}\|}$
chebspec — Chebyshev spectral differentiation matrix	1.95E-07	1.13E-16
clement — Tridiagonal matrix with zero diagonal entries	3.66E-16	2.80E-17
lehmer — Symmetric positive definite matrix	2.67E-15	6.46E-18
circul — Circulant matrix	6.39E-14	8.35E-16
lesp — Tridiagonal matrix with real, sensitive eigenvalues	1.22E-16	1.43E-18
minij — Symmetric positive definite matrix	2.44E-15	2.99E-18
orthog — Orthogonal and nearly orthogonal matrices	1.41E-17	6.56E-17
randjorth — Random J-orthogonal matrix	1.40E-09	7.24E-16
frank - Matrix with ill-conditioned eigenvalues	5.59E-03	1.52E-21

Table 4.3: Comparisons using Matlab matrix gallery

4.4 PDE Application

A further test for the accuracy of the algorithm involved applying the algorithm to a problem where a known solution exists. The chosen example was a simple implementation of the heat equation:

$$\frac{\partial\phi}{\partial t} = \alpha\frac{\partial^2\phi}{\partial x^2} \quad 0 \leq x \leq L,\ t \geq 0 \tag{4.13}$$

An initial condition of $\phi(x, 0) = sin(\pi x) + sin(3\pi x)$ and a precise solution of $\phi(x, 0t) = sin(\pi x)e^{-t\pi^2} + sin(3\pi x)e^{-9t\pi^2}$ [31] were applied to the partial differential equation (PDE). The Crank Nicolson method [40] was used to transform the PDE into a large linear system of equations with a size of 2000 x 2000. The Crank Nicolson method was run for a single time step from time zero to time .001. Matlab and Cramer's rule algorithm were then used to solve the system of linear equations for a solution. The computed solutions were compared to the exact solution and norms of the difference were recorded, as shown in Table 4.4. The results of both Matlab and Cramer's rule algorithm show an accurate answer. It should be noted that the method for transforming the example into a system of linear equations will introduce some small amount of error. This example simply provides more evidence of the algorithm's accuracy and stability.

Linear Solver	$norm_2$	$norm_\infty$
Cramer's Rule Algorithm	2.34E-8	7.39E-10
Matlab	2.34E-8	7.41E-10

Table 4.4: Application of Cramer's rule algorithm to a PDE

Chapter 5

Serial Implementation Results

This chapter presents implementation results for the proposed algorithm. The latter has been implemented on a single processor platform. Compiled in the C programming environment, it has been compared to LAPACK (Linear Algebra PACKage) on a single core Intel Pentium machine to provide a baseline for comparison.

5.1 Optimization Efforts

Several optimization techniques were applied to the implementation, including the SIMD (single instruction multiple data) parallelism that is standard on most modern processors. The program code also employs memory optimizations such as cache blocking to reduce misses. No multi-processor parallelization has been programmed into the implementation such that the algorithm itself could be evaluated against the industry standard prior to any parallelization efforts.

5.1.1 Extended Condensation

Software packages such as LAPACK that leverage BLAS routines are optimized to do multiple calculations on data while it's in cache memory as opposed to moving the data in and out of memory multiple times as the algorithm traverses the matrix. This doesn't follow the mathematical representation of LU-factorization but is much more efficient on a computer.

For the proposed algorithm a similar technique was employed with the extended Chio's condensation. Instead of traversing a matrix N times for an $(N \times N)$ matrix the algorithm can reduce the repetitions. If a Chio's condensation step size of four is used, the algorithm only traverses the matrix $\frac{N}{4}$ times. In theory this could decrease memory accesses by 75%, although in practice the memory calls are not reduced that significantly. It does, however, make an improvement in run times.

This improvement, however, must be balanced with the accuracy of the algorithm. The pivoting scheme, as discussed earlier, depends on identifying the largest lead determinant. As the Chio's condensation step size increases so does the size of the determinants that need to be evaluated to find the lead determinant. The gain in memory accesses must be balanced with the additional computation required for the determinant identification.

5.1.2 Memory Re-use

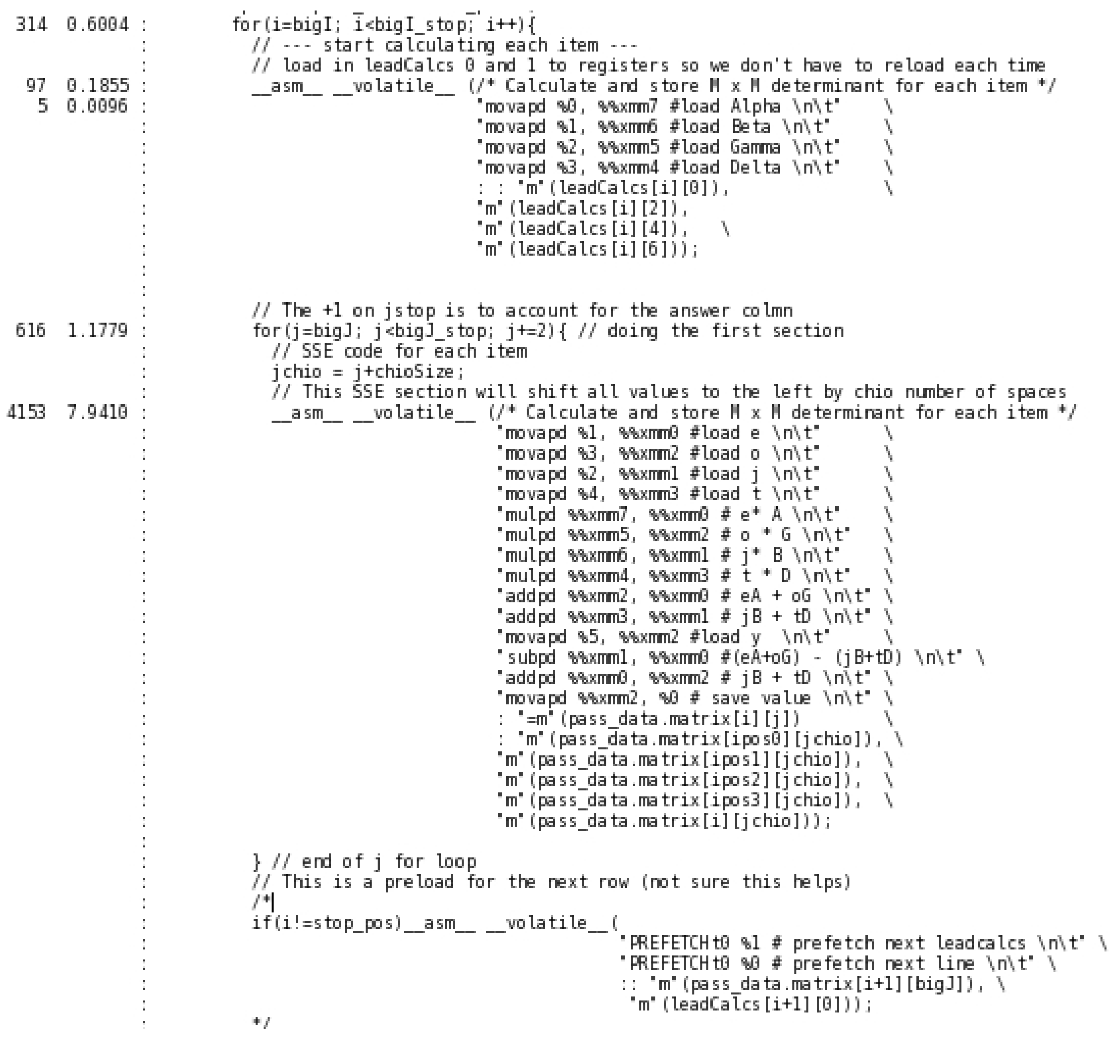
The mirroring function of the algorithm requires a copying of the matrix. While the copy operation itself is not particularly time consuming, setting up the target memory requires significant time. When a program allocates memory it generally must make a call to the operating system to check permissions and a number of other items. This forces a context switch in the processor and generates a great deal of overhead. Making this call to the operating system each time a algorithm mirrors leads to significant idle time.

The solution to eliminate this overhead entails allocating all the memory needed at the outset of the process. In order to keep the memory requirements to a minimum and reduce the probability of page faults, the memory used to store matrices is reused continually. As a matrix is condensed it vacates memory. When the matrix reaches a mirroring size, the newly formed mirror takes the memory recently vacated. In addition, those memory locations may still be stored in cache, particularly when the matrices reach the end of their condensation.

The act of mirroring requires a memory copy of the matrix. This can be combined with the first condensation step of the new mirrored matrix. If when a matrix is being copied the computations for the first condensation step are applied it saves moving the same data through memory twice. Once for the memory copy of the matrix and once for the first condensation step. Instead both are done together to reduce the number of times the new matrix must be accessed.

5.1.3 Profiling

One very common tool for optimizing an algorithm is a profiler. This helps identify hot spots in computer code where a programmer can target their efforts. The profiling tool 'Oprofile' [30] provided detailed information on the execution of the proposed algorithm. The profiling first level report provides a breakdown of run times for each method. The report shows the condensation method with the most amount of processing time as expected. The profiling tool can then provide a breakdown of the run times for specific lines code. The profiler as configured for these tests samples the CPU at a set interval to see what instruction the computer is processing at each sample. The profiler then tallies the number of samples that fell on each line of code. Lines that collect more samples spent more time using the CPU, statistically. Figure 5.1 shows an example for the proposed algorithm. This particular section shows the core loop of Chio's condensation that represents the bulk of the run time. The left margin shows the number of samples and percentage of time spent on particular lines of code.



```
  314  0.6004 :        for(i=bigI; i<bigI_stop; i++){
              :          // --- start calculating each item ---
              :          // load in leadCalcs 0 and 1 to registers so we don't have to reload each time
   97  0.1855 :          __asm__ __volatile__ (/* Calculate and store M x M determinant for each item */
    5  0.0096 :                               "movapd %0, %%xmm7 #load Alpha \n\t"    \
              :                               "movapd %1, %%xmm6 #load Beta \n\t"     \
              :                               "movapd %2, %%xmm5 #load Gamma \n\t"    \
              :                               "movapd %3, %%xmm4 #load Delta \n\t"    \
              :                               : : "m"(leadCalcs[i][0]),               \
              :                               "m"(leadCalcs[i][2]),
              :                               "m"(leadCalcs[i][4]),   \
              :                               "m"(leadCalcs[i][6]));
              :
              :
              :          // The +1 on jstop is to account for the answer colmn
  616  1.1779 :          for(j=bigJ; j<bigJ_stop; j+=2){ // doing the first section
              :            // SSE code for each item
              :            jchio = j+chioSize;
              :            // This SSE section will shift all values to the left by chio number of spaces
 4153  7.9410 :            __asm__ __volatile__ (/* Calculate and store M x M determinant for each item */
              :                                 "movapd %1, %%xmm0 #load e \n\t"      \
              :                                 "movapd %3, %%xmm2 #load o \n\t"      \
              :                                 "movapd %2, %%xmm1 #load j \n\t"      \
              :                                 "movapd %4, %%xmm3 #load t \n\t"      \
              :                                 "mulpd %%xmm7, %%xmm0 # e* A \n\t"    \
              :                                 "mulpd %%xmm5, %%xmm2 # o * G \n\t"   \
              :                                 "mulpd %%xmm6, %%xmm1 # j* B \n\t"    \
              :                                 "mulpd %%xmm4, %%xmm3 # t * D \n\t"   \
              :                                 "addpd %%xmm2, %%xmm0 # eA + oG \n\t" \
              :                                 "addpd %%xmm3, %%xmm1 # jB + tD \n\t" \
              :                                 "movapd %5, %%xmm2 #load y  \n\t"     \
              :                                 "subpd %%xmm1, %%xmm0 #(eA+oG) - (jB+tD) \n\t" \
              :                                 "addpd %%xmm0, %%xmm2 # jB + tD \n\t" \
              :                                 "movapd %%xmm2, %0 # save value \n\t" \
              :                                 : "=m"(pass_data.matrix[i][j])        \
              :                                 : "m"(pass_data.matrix[ipos0][jchio]), \
              :                                 "m"(pass_data.matrix[ipos1][jchio]),  \
              :                                 "m"(pass_data.matrix[ipos2][jchio]),  \
              :                                 "m"(pass_data.matrix[ipos3][jchio]),  \
              :                                 "m"(pass_data.matrix[i][jchio]));
              :
              :          } // end of j for loop
              :          // This is a preload for the next row (not sure this helps)
              :          /*
              :          if(i!=stop_pos)__asm__ __volatile__(
              :                                               "PREFETCHt0 %1 # prefetch next leadcalcs \n\t" \
              :                                               "PREFETCHt0 %0 # prefetch next line \n\t" \
              :                                               :: "m"(pass_data.matrix[i+1][bigJ]), \
              :                                                "m"(leadCalcs[i+1][0]));
              :          */
```

Figure 5.1: Example of opannotate results from 'Oprofile' for the proposed algorithm

5.1.4 Assembly Code

The profiling tool provided the target areas for code optimization. These areas were changed to assembly code using gcc-inline-assembly. This gave opportunity to fully utilize the SSE2 instruction set [35]. These instructions allow the proposed algorithm to leverage the hardware supported floating point operations, similar to BLAS routines. These instructions allow for multiple floating point operations in parallel on one processor. This provided a significant speed-up.

Converting the core loop to assembly also allowed for optimization of the SSE registers. The algorithm now keeps the lead minors, discussed earlier, in the registers to streamline memory access. Four of the eight registers hold the lead minors while the other four shuttle matrix data in and out. This prevents the lead minors from being moved back to the cache to make room for new matrix data. For the particular implementation this also drives the Chio's condensation step size to be a multiple of four to match up with the SSE registers holding the lead minors.

5.1.5 Cache Blocking

The final piece of optimization ties to the size of the layer 1 cache. The algorithm focuses on a portion of the matrix that comfortably fits into the layer 1 cache and completes all work on that section before touching the next section. The algorithm works on pieces of a different rows at one time and the returns later to get the other pieces of a row as opposed to simply running the length of each row during each Chio's condensation.

5.1.6 Prefetching

A common optimization technique is prefetching data, or in other words, loading data to the cache before it's actually requested. This should reduce memory misses and the time associated with the miss. Unfortunately, this optimization was unsuccessful. Chio's condensation, and most likely LU-factorization, are very predictable in what

Matrix Size	Algorithm (sec)	MATLAB (sec)	Ratio
1000x1000	2.06	.91	2.26
2000x2000	16.44	6.32	2.60
3000x3000	52.33	19.92	2.63
4000x4000	115.44	45.10	2.56
5000x5000	220.32	86.90	2.54
6000x6000	380.92	142.05	2.68
7000x7000	583.02	242.61	2.40
8000x8000	872.26	334.68	2.61

Table 5.1: Execution comparison to LAPACK

data is needed. Both methods work their way across or down the matrix repeatedly in a deterministic manner. It's likely that the hardware prefetch works well with this type of memory usage and automatically prefetch what's needed. In fact, manual prefetching may actually inhibit this process generating additional cache misses. For that reason prefetch is not part of the algorithm optimization.

5.2 Run Time Results

The processor used for the serial comparison was an Intel Pentium M Banias with a frequency of 1.5GHz using a 32KB L1 data cache and 1MB L2 cache. The manufacture quotes a maximum GFLOPS rate of 2.25 for the specific processor [27]. The Linpack benchmark MFLOPS for this processor is given as 755.35.

As can be seen in Table 5.1, the algorithm runs approximately 2.5 times slower than the execution time of Matlab, independent of matrix size, which closely corresponds to the theoretical complexity analysis presented above. Both pieces of software processed numerous trials on a 1.5GHz single core processor. The results further show that while the algorithm is slower than the LU-based technique, it is consistent. Even as the matrix sizes grow, the algorithm remains roughly 2.5 times slower than state of the art methodologies. Figure 5.2 depicts a comparison between the proposed algorithm and Matlab.

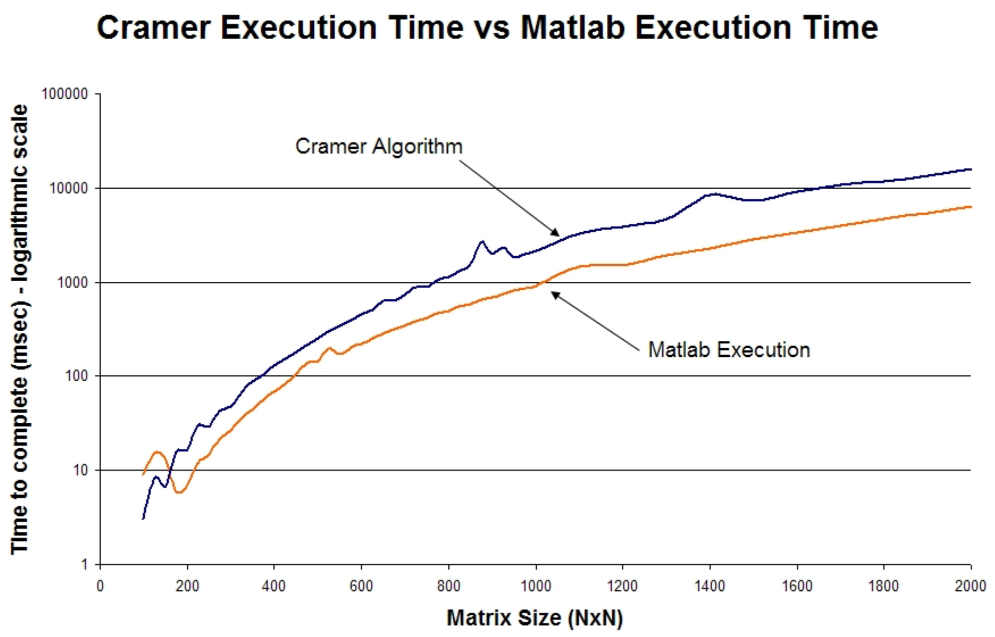


Figure 5.2: Algorithm execution times compared to those obtained using Matlab$^{(TM)}$

The theoretical number of floating point operations (FLOPS) to complete a 1000x1000 matrix based on the complexity calculation is roughly 1555 million. The actual measured floating point operations for the algorithm summed to 1562.466 million. This equates to an estimated 758 MFLOPS. The Matlab algorithm measured 733 MFLOPS based on the measured execution time and theoretical number of operations for LU factorization.

Chapter 6

Parallel Design

With the abundance of parallel processing computing fabrics, in order to determine the scalability attributes of any new algorithm it is natural to consider the implications of realizing it using parallel processing units. The parallel implementation of the proposed algorithm can essentially be broken into two phases. First, the reduction of a given matrix to a number of reasonable sized matrices. Second, the application of Cramer's rule in parallel to the condensed matrices to find the values of the variables. The point to transition from phase one to phase two depends on the original size of the matrix and the number of processors employed. The following sections detail the various implementation aspects of the proposed algorithm over parallel processing platforms.

6.1 Matrix Distribution

One of the key challenges facing LU-factorization is the distribution of a matrix to the various processors using a 2-D block cyclic allocation. For this algorithm a cyclic distribution of the columns to available processors provides a simple and predictable allocation. Figure 6.1 shows the allocation of columns to processors. The first column to P0, the second column to P1 and so on, until all processors are allocated a column at which point the process repeats giving each processor another

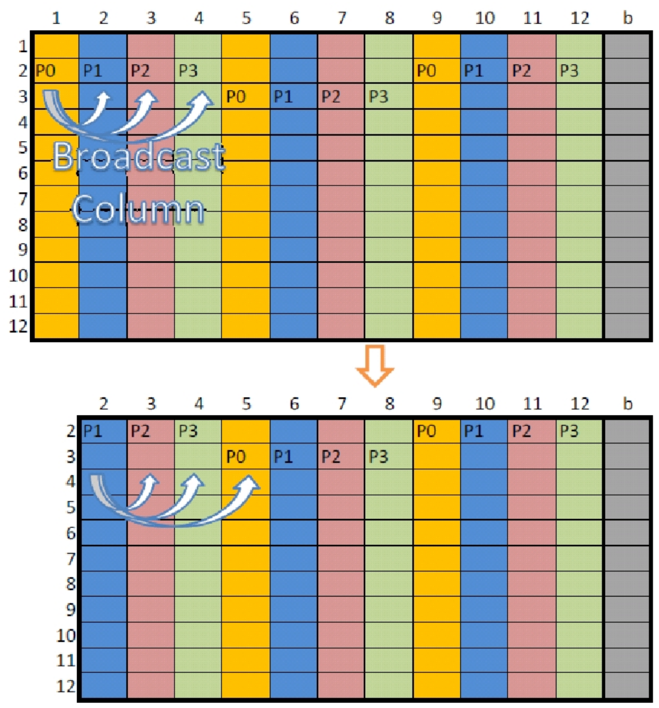


Figure 6.1: Matrix allocation and parallel condensation of proposed algorithm

column. In addition each processor receives the b column that it must include in all condensations. The processor responsible for a particular column can be determined with a simple modulus calculation.

6.2 Parallel Condensation and Balanced Load Distribution

Once the columns have been distributed or loaded to the various processors, the parallel condensation commences. The processor holding the lead column broadcasts the column to all other processors, as depicted in figure 6.1. At the conclusion of that broadcast process, the processor sends a marker indicating which row should be considered as the pivot. The receiving nodes then condense all the rows they posses using Chio's condensation.

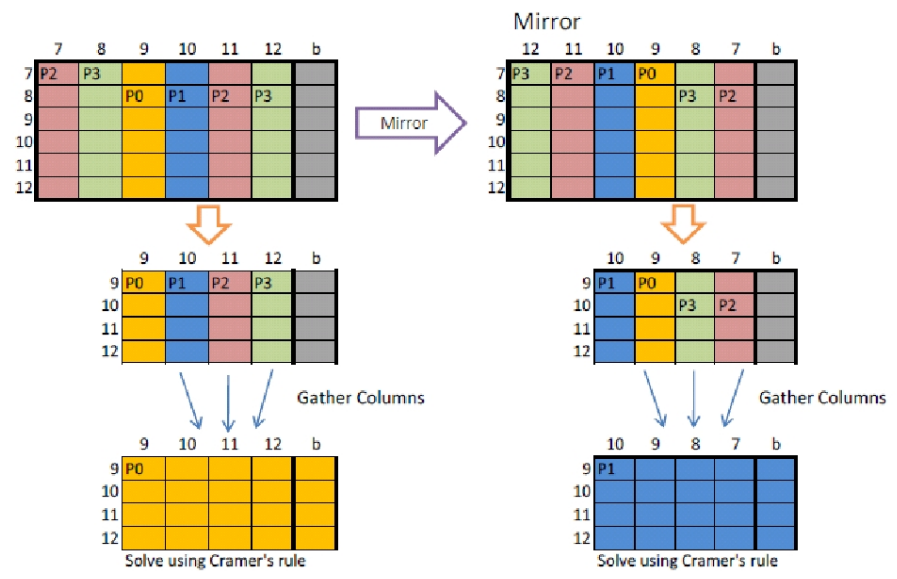


Figure 6.2: Mirroring and Gathering of data for proposed algorithm

The beneficial attribute of doing so is that the lead column is highly predictable. In fact, no communication is required to establish which processor will be broadcasting information. A node will determine if it is going to be the lead node during the next condensation step as it performs current calculations. This allows the future lead processor to begin broadcasting the subsequent lead column prior to the processor's completion of the current condensation step. For example, figure 6.1, the bottom matrix will require P1 to broadcast the lead column. This processor can begin sending column 2 out as soon as it condenses that column in the first step, and then continue condensing column 6 and 10 while it's sending data. This is often referred to as send-ahead or pipelining, allowing communication work to overlap computation work.

The condensation technique used in the algorithm would normally suffer from the same workload distribution problem that LU-factorization exhibits. After a number of condensation iterations are performed, there would be processors that don't have columns left while others do. However, the mirroring step redistributes the work among the processors such that the algorithm inherently re-balances the workload.

As the condensation progresses, the matrix will eventually reduce to a size equal to the number of variables. At this point the algorithm will mirror the matrix and distribute the number of variables each side is responsible for. Figure 6.2 shows the matrix after it is reduced to size 6, which is the number of variables this matrix was assigned. At this point, the columns are logically mirrored, yet it should be noted that the newly mirrored columns remain with the same processors. There is no inter-node communications required, simply a memory copy operation. The left matrix can continue to solve for variables 10, 11 and 12, while the newly created mirror can solve for 9, 8 and 7.

6.3 Distributed Implementation of Cramer's Rule

The second phase of the parallel work is the combination of the columns to one processors to complete the condensation and solve using Cramer's rule. Obviously the strength of Cramer's rule and reason it's so inviting for parallel applications is the ability for the gathered matrices to complete without any communication. The only remaining communication is to report the various calculated variables back to a central process.

Figure 6.2 shows the gathering of the columns to P0 after the matrix reaches a size of four. The specific point that the matrix recombines to one processor is flexible. The recombining of the matrix can also easily leverage optimized communication calls such as the MPI_GATHER command.

Once the columns are gathered to a processor it's best for that node to delay the further reduction and application of Cramer's rule on the small matrix. The processors as a group would then return to the mirrored matrix on the top right of figure 6.2 to condense that matrix and gather it to P1. Once all the mirrors are gathered to individual processors, each node can work independently on the small matrices it has been allocated. This allows for nodes to work completely in parallel, when the matrices are the smallest. Beneficial in two senses. This is where a serial solution is faster, and it's when communication overhead would have its greatest impact due to the reduced amount of computation to hide the communication.

6.3.1 Optimal Point to Gather Columns

The size of the matrix when all columns should be gathered to a single processor is referred to as size F. This is the point when the condensation of a matrix is better handled by a single processor. There are two hard limits on this size. First, it must be larger than the number of nodes in the cluster. If sixteen nodes are used then $F \geq 16$, otherwise nodes will sit idle. Second the gathering should not occur before enough mirrorings have occurred to assign at least one mirror to each node.

This second constraint will depend on the initail size of the matrix. For example a 1000×1000 matrix will mirror initially giving two matrices and then mirror again when it reaches size 500 giving four matrices. If sixteen nodes are being used the gather should not occur at this point because there will only be four matrices to assign. Twelve nodes would sit idle. Instead the algorithm should wait until the algorithm mirrors two more times. This would occur at size 250 and then 125. At this point there would then be sixteen matrices available, fifteen mirrors and the original matrix. If the matrices are gathered when they reach this size then there would be enough for each node to have a matrix to work on independently, thus supporting full utilization of the matrix.

Beyond these two hard limits there are also consideration on the communication. Gathering very large matrices might generate additional idle time because of communication delays. This communication is not broadcast and while many nodes can send at roughly at the same time the receiving node can only take information from one node at a time. For this reasearch, a standard value of $F = 64$ was used.

6.4 Communication Requirements

The theoretical amount of communication for the proposed algorithm should consist of the messaging required to facilitate the two phases. The first phase, where the matrix is condensed amongst all the processors, will require a broadcast of the lead column for each iteration. Keeping in mind that the column size will decrease during each pass. Assuming that N represents the size of the original matrix then equation 6.1 represents the data communicated to reduce the matrix and its mirrors.

$$\sum_{k=0}^{N}(N-k) + \sum_{k=0}^{log_2 N} 2^k \left(\frac{1}{2}\left(\frac{N}{2^k}\right)^2 - \left(\frac{N}{2^k}\right)^2 \right) \tag{6.1}$$

Reducing to a more manageable form gives the following

$$\frac{3N^2}{2} - \frac{N}{2} - \frac{N}{2}\left(1 + log_2 N\right) \tag{6.2}$$

This amount includes the communication volume to completely condense the matrix and it's mirrors. The proposed algorithm, however, only condenses the matrices to the gather size, at which point phase two of the parallel algorithm begins. The fact that the algorithm doesn't condense completely means that a small part of the communication and computation is saved in phase one. If the size at which phase two is initiated is F, and the less significant terms are removed, the complete communication volume is the following

$$\frac{3N^2}{2} - \frac{3F^2}{2} \tag{6.3}$$

The communication for phase two where each condensed matrix is communicated to a single processor requires gathering all the columns. If p represents the number of processors, and F represents the size of the matrix when gathered then, $\frac{F}{p}$ columns are transmitted, times the number of transmitting processors, $p-1$, times F, which is the length of each column. This amount must then be multiplied by the number of leaves on the binary tree giving equation 6.4.

Number of Processors	PCramer Messages	PCramer Volume (kB)	ScaLAPACK Messages	ScaLAPACK Volume (kB)
4	18,579	12,115	8,042	15,560
8	23,703	12,148	13,797	23,608
12	31,387	12,159	19,822	31,692
16	28,367	12,115	25,888	39,696
24	47,831	12,247	38,022	50,880
32	42,767	12,166	50,047	72,000

Table 6.1: Parallel Cramer algorithm (PCramer) communication compared to LU-factorization (1000 x 1000 matrix)

$$2^{log_2 \frac{N}{F}} \times \frac{F}{p} \times (p-1) \times F \tag{6.4}$$

Combining the two phases yields the theoretical bandwidth in equation 6.5, which is overwhelmingly dependent upon the size of the original matrix N, when F is small.

$$8\, bytes/double \times \left(\frac{3N^2}{2} - \frac{3F^2}{2} + 2^{log_2 \frac{N}{F}} \times \frac{p-1}{p} F^2 \right) \tag{6.5}$$

6.5 Communication Volume Results

The proposed algorithm has been coded using the C programming language, using doubles to store and calculate data, with MPICH2 serving as the message passing interface. The implementation uses MPI for inter processor communication and pthreads for timing and parallel work within a single processor. The developed code was tested for accuracy and communication requirements on a simulated MPI cluster using Argonne National lab's *Fast Profiling library for MP*I (FPMPI). Solution sets from MATLAB used for accuracy comparison.

The main focus for this research is the amount of communication required in a true broadcast environment. The measurement of a broadcast communication is counted once. A broadcast message of one double (8 bytes), has a communication volume of 8 bytes, not some factor of $(p-1)$ as discussed in Subsection 2.5.2. This is done even though the MPICH2 implementation will actually convey the message in $log(p-1) \times 8$ bytes.

The test scenario compared the proposed parallel algorithm, referred to as PCramer, to Parallel LU-factorization implemented within the ScaLAPACK library. Both programs used double floating point arithmetic and were run on a randomly created 1000x1000 size dense matrix. The ScaLAPACK was set to a block size of 4, since this gave the lowest communication volume in empirical trials. The profiling software counted a MPI_broadcast message once as discussed earlier. The results of these tests are listed in table 6.1. All tests were simulated on a single core laptop running Fedora's Linux operating system.

A gather size of 10 was used for the parallel Cramer's rule algorithm. This means that when the original matrix or one of the mirrors condenses to a size of 10x10, all the columns are communicated to a single processor for application of Cramer's rule to find the variables. The gather communication is not broadcast messaging. For a gather size of 10, this communication is very small when compared to the total communication, with around 98% of the messaging being broadcast in table 6.1, for eight processors.

The nature of the communication for the proposed algorithm is largely broadcast with a minimal amount of communication in the form of MPI_GATHER calls to

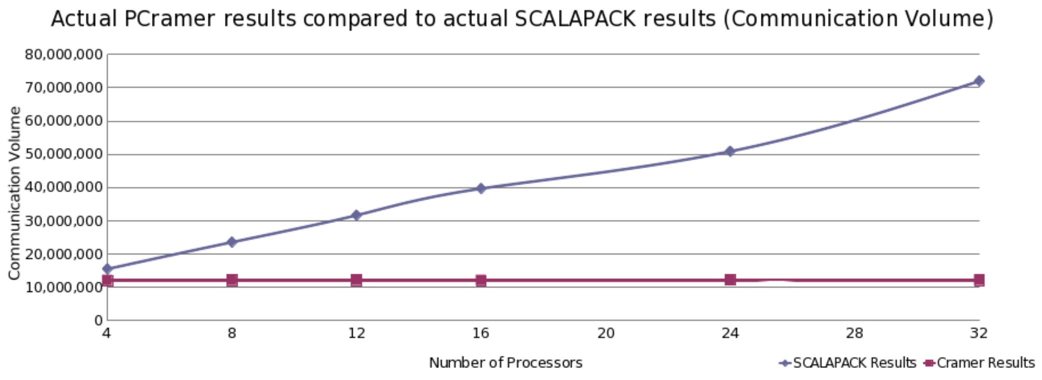


Figure 6.3: Proposed Cramer algorithm (PCramer) Communication Volume

Number of Processors	PCramer Volume (kB)	Predicted Volume (kB)	Relative Error
4	12,115	12,059	0.46%
8	12,148	12,069	0.65%
12	12,159	12,072	0.72%
16	12,115	12,074	0.34%
24	12,247	12,075	1.40%
32	12,166	12,076	0.74%

Table 6.2: Comparison of parallel Cramer's rule algorithm predicted communication volume to actual

recombine a reduced matrix to one processor. Since the broadcast communication is the vast majority of the bandwidth and message count, the overall communication profile follows this pattern when comparing empirical results. The amount of broadcast data is a set amount per condensation step. This means that the actual data transmitted across the network backbone is dependent on the size of the original matrix and not the number of processors. The communication pattern, as can be seen on figure 6.3, stays nearly constant for a specific sized problem. Unlike LU-factorization, represented here by ScaLAPACK's `pdgetrf()` function, the addition of more resources does not drastically increase the communication requirement.

A comparison of the theoretical communication predicted by equation 6.5 to the empirical results shows a very close relationship. Table 6.2 displays the comparison and shows that the predicted amount is within 2% of the empirical findings for all scenarios. In fact the empirical amount is always slightly more than the predicted amount which is expected because of the small amount of communication required to setup the global variables and data structures between the various processors.

Again, these communication results assumes a true broadcast infrastructure. If the broadcast messages were sent as a point-to-point message to all processors the communication volume would be significantly higher and the ScaLAPACK communication volume comparatively less. The benefit of this communication structure is only realized in a true broadcast medium.

Number of Processors	Matrix Size	PCramer Run Time (sec)	ScaLAPACK Run Time (sec)	Run Time Ratio
2	2000x2000	8.92	2.98	2.99
4	2000x2000	4.85	2.52	1.93
6	2000x2000	2.04	2.05	1.00
8	2000x2000	2.05	1.36	1.51
16	2000x2000	1.78	1.19	1.49

Table 6.3: Parallel Cramer algorithm (PCramer) run time compared to LU-factorization (2000 x 2000 matrix)

6.6 Implementation Results

The parallel implementation of Cramer's rule can also be extended using a larger condensation step size as discussed in section 3.2. The same optimal values of condensation size were seen in the parallel version as the serial version. The PCramer implementation also incorporated a version of pipelining to broadcast required data in advance.

The processor that holds the column or columns for the next condensation broadcasts that data as soon as it was ready, instead of waiting to condense the rest of the columns it holds. For example, if processor P_1 holds the lead column for the next condensation step as well as four other columns it would condense the future lead column, find the pivot and broadcast out the column. The processor would then continue condensing its remaining four columns. This would give the message time to arrive at the other processors while they were still working on the current condensation step. Processor P_1 could also convey the future pivot row without any communication or delay since it contains all the data needed to select the appropriate pivot. This is a result of using the 1-D matrix distribution, which also helps minimize the communication overhead when compared to a 2-D matrix distribution.

The run times were collect using the 'mil' cluster in the *Machine Intelligence Lab* at the University of Tennessee. The cluster consists of four servers each with a Quad core Intel Xeon X5472 processor and eight gigabytes of RAM. The CPUs

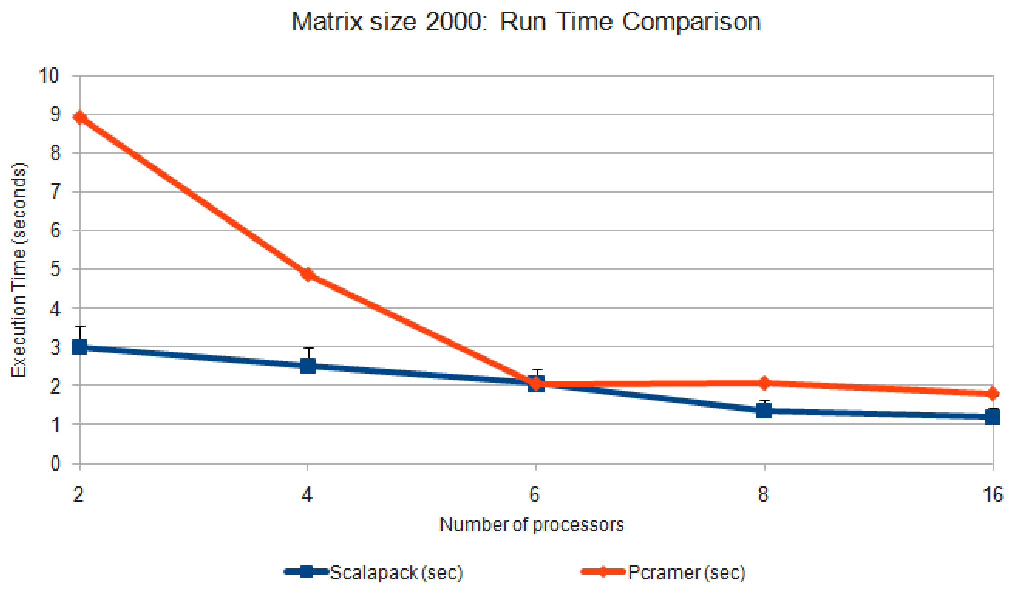


Figure 6.4: Parallel Cramer algorithm (PCramer) run time graphically compared to LU-factorization by processor count

are 3.00 GHZ processors with 12 megabytes of L2 cache. Six megabytes of the L2 cache are shared by a pair of cores and the remaining six megabytes shared by the other two cores. To the MPI daemon this appears as a 16 processor cluster, so tests were run up to 16 processors. However, because two of the cores share the same part of the L2 cache there is local memory contention between the pair of cores. This exacerbates memory constrained operations like the scientific computing being done in this research. For this reason the results obtained for 16 processors are not a true reflection of the parallel algorithms because the local memory bandwidth is most likely the limiting factor for test with more than eight processors.

Table 6.3 and Figure 6.4 show the run times of the parallel Cramer's rule algorithm and a typical parallel LU-factorization implementation provided by ScaLAPACK. The problem size of 2000x2000 was used for this particular test and ten trials were run for each processor count. The average run times for those trials are shown.

The parallel Cramer's rule algorithm compares well at this processor range and matrix size. The workload amount is adequate to hide the communication overhead

Number of Processors	Matrix Size	PCramer Run Time (sec)	ScaLAPACK Run Time (sec)	Run Time Ratio
8	1000x1000	0.49	0.35	1.38
8	2000x2000	2.05	1.36	1.51
8	3000x3000	13.73	3.71	3.70
8	4000x4000	66.21	6.83	9.69

Table 6.4: Parallel Cramer algorithm (PCramer) run time compared to LU-factorization (8 processors)

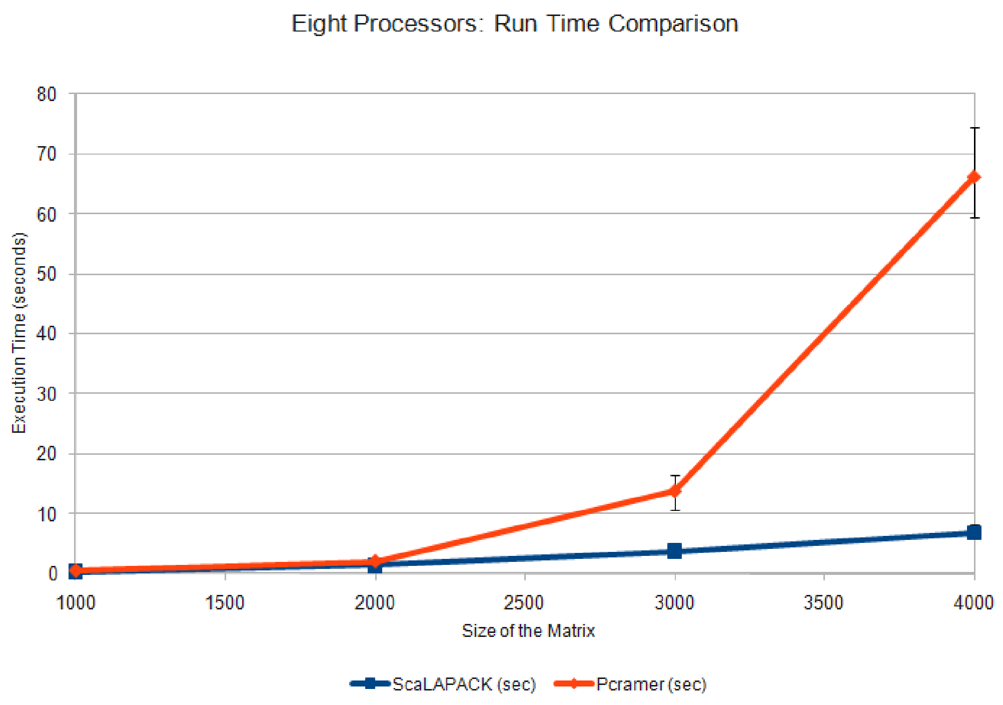


Figure 6.5: Parallel Cramer algorithm (PCramer) run time graphically compared to LU-factorization by matrix size

for the processors and the matrix portions are small enough to keep completely within cache memory. Keeping in mind that the message passing package, MPICH2, does not truly broadcast the messages. The larger number of processors likely begins to create more communication and the run time becomes more dependent on the messaging overhead than the actual computations. As can be seen in Figure 6.4 the run times for 6, 8 and 16 are roughly the same. The amount of communication begins to nullify the extra processing capacity as does the local memory contention.

Table 6.4 shows the comparison of run times as the matrix size increases. Here eight processors is shown because that's the largest true processor count for the *mil* cluster, as discussed earlier. ScaLAPACK uses the BLAS subroutines that are highly optimized for local memory access. The parallel Cramer's rule code, uses SSE code but is not optimized for local memory access within each node doing computation. As each processor's portion of the matrix grows the local memory contention becomes more of a problem. When a size of 4000 x 4000 is reach, Figure 6.5 clearly shows that the algorithm diverges from the baseline. Improved cache blocking and memory optimization, as well as the implementation of a true broadcast message passing scheme would allow the algorithm to scale in a more competitive manner.

Chapter 7

Application to Sparse Matrices

A large number of the matrices used in power systems and other scientific fields are sparse, meaning a vast majority of the matrix is filled with zeros. In order to efficiently handle these types of matrices the proposed algorithm must avoid computations where the outcome is already known. A sparse serial version as well as a sparse parallel version (SPCramer) of the proposed algorithm were developed to test its suitability for handling such matrices.

The sparse versions of the algorithm are based on the same programming code as the previous versions, but with a number of major modifications to handle sparse situations. The code avoids as much computation as possible, although at the cost of additional checks and messaging. These additional overheads are used to identify situations where computation can be variable due to the sparsity of the system.

The data structure used for storing the sparse matrix is a full matrix like those used for dense storage. This is far from optimal but assisted in troubleshooting and analyzing the sparse potential of the algorithm. If a condensed matrix format, like the *compressed-column* form [10] was used there would need to be coding to manipulate the data structure as fill-ins occurred [15]. Fill-in is defined in this research as matrix positions that were previously zero changed to a non-zero value by the algorithm.

Unfortunately, the full matrix storage made the application of SSE instructions in the computations difficult. An SSE instruction performs operations on multiple pieces of data but since the non-zero values are not necessarily contiguous the SSE instructions cannot target the non-zero matrix locations. For this reason SSE optimizations were not used and therefore reduced the overall computational speed of the implementation.

The following sections detail the three main features of the sparse implementation: structure prediction of the variables, computation avoidance, and switching to dense matrix condensation as the fill increases. The final section will give the implementation results in comparison to the dense version.

7.1 Structure Prediction of variables

Section 3.5 discusses the ability of the proposed algorithm to compute a subset of the variables, instead of all the variables. This trait can be exploited in certain sparse situations. If a sparse matrix is coupled with a sparse solution set it's possible that some of the variables will calculate to zero, simply due to the sparse structures. The ability to check which variables will calculate to zero based simply on the sparse structure is referred to as *structure prediction for sparse solve* [21]. The algorithm to find the variables that are non-zero is $\mathcal{O}(N^2)$, so the additional overhead to check the matrix for this condition is worth the potential savings even if one unknown can be avoided.

A simple visual example of the prediction algorithm is displayed in Figure 7.1. In this case a directed graph is built of the sparse matrix. Each non-zero value of the solution vector is marked on the graph. All other nodes that have a directed path to one of the marked nodes will most likely have a non-zero unknown associated with it. The nodes that do not have a path to a marked node are guaranteed to have an x value of zero. In the example shown, nodes 1 and 4 are marked with red since those values are non-zero in the solution vector. It can then be seen that only nodes 2 and

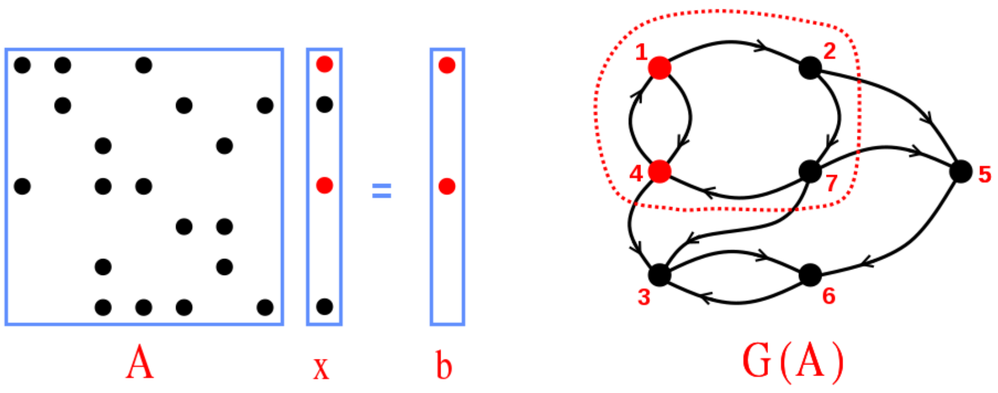


Figure 7.1: Structure Prediction for Sparse Solve

source: http://www.cs.ucsb.edu/~gilbert/talks/SparseDay2.ppt

7 have a path to the marked nodes in graph G(A). Therefore, nodes 1, 2, 4 and 7 will most likely have non-zero values while variables 3, 5 and 6 will equal zero and do not need to be solved for.

The sparse version of the proposed algorithm runs this check before it begins, if the solution vector is not dense. In the parallel version it also runs this check, however it runs completely on one processor since a parallel version of the prediction algorithm was not developed or readily available. This requires the entire matrix to be loaded into memory to solve the prediction algorithm. This is unusual because a parallel solver would not generally load the entire matrix but only the portion needed for computation. The sparse Cramer's rule algorithm loads the matrix, runs the sparse check and then releases the full matrix and reloads the smaller portion of the matrix it needs for computation.

7.2 Computation Avoidance

The similarities of the proposed algorithm with Gaussian elimination allows it to reuse some of the ideas of a sparse Gaussian elimination [19]. A key concept is how the lead row and column affects the values in the resulting matrix during a condensation step. Each matrix entry is calculated by the equation $a'_{ij} = a_{11}a_{ij} - a_{i1}a_{1j}$, as documented in Section 2.4. Since the a_{11} value is always unity the only values that affect the value of a_{ij} are those that are subtracted, namely a_{i1} and a_{1j}. As can be seen, if either a_{i1} or a_{1j} are zero the resulting equation is $a'_{ij} = 1 \times a_{ij} - 0 = a_{ij}$. The value of a_{ij} does not change.

The sparse algorithm takes advantage of this fact by calculating only columns that don't lead with a zero. This is an $\mathcal{O}(N^2)$ check so it does not greatly affect the overall computational complexity. The lead column also only has a certain number of non-zero values. In the sparse version of the algorithm, only the rows with non-zero values in that lead column are calculated. The only values that are even submitted to the processor for computation are those that will change.

In the parallel version these sparse considerations require sending four pieces of information. As in the normal Parallel Cramer's rule algorithm, it sends the values in the lead column and a marker for the pivot row. However, in this case it only needs to send the non-zero values and an integer array telling which rows are non-zero. The receiving nodes can then bypass all rows except those that have non-zero values in the broadcasted column. In order to pass only the non-zero values the first broadcast must be a message telling how many non-zero values the receiving nodes should expect. This way the other nodes determine how to size their buffers.

There is a cost associated with sending the condensed lead column. The broadcasting node must check the first column for non-zeros and store the positions that are non-zero in an integer array. This is an $\mathcal{O}(N^2)$ check. These values must then be broadcast to the other nodes generating additional communication. In cases where the lead column is sparse, this additional integer array is balanced by the reduced number of matrix entries that need to be communicated. However, as the matrix becomes less sparse the communication may exceed that associated with the dense version of the algorithm. It should be noted that if a data structure, such as the *compressed-column* form was used then the checks for non-zero entries would not be needed since this information would be readily available. Although there would be additional overheads in maintaining the data structure.

These checks to avoid computation add some additional overhead, although it is is minimal compared to the overall complexity of the algorithm. In return, it provides the potential to drastically reduce the amount of computation. The following equation shows the computational workload given an initial level of sparsity, α. The overall workload depends on the specific matrix, so it is the combination of work on sparse items in matrix and the work on non-sparse items in matrix. The equation follows this logic with the workload on sparse items multiplied by α and the workload on non-spare items multiplied by $(1-\alpha)$. As before, γ represents the computational complexity.

$$\gamma = \sum_{k=1}^{n} \alpha_k \left[(n-k) + (n-k)\right] + (1-\alpha_k)\left[4(n-k) + 2(n-k)^2\right]$$

$$\gamma = \sum_{k=1}^{n} 2\alpha_k(n-k) + (1-\alpha_k)\left(2(n-k)^2 + 4(n-k)\right) \tag{7.1}$$

$$\gamma = \sum_{k=1}^{n} 2\alpha_k(n-k) + 2(n-k)^2 - 2\alpha_k(n-k)^2 + 4(n-k) - 4\alpha_k(n-k)$$

$$\gamma = \sum_{k=1}^{n} (2-2\alpha_k)(n-k)^2 + (4-2\alpha_k)(n-k)$$

$$\gamma = \sum_{k=1}^{n} (2-2\alpha_k)(n^2 - 2nk + k^2) + (4-2\alpha_k)n - (4-2\alpha_k)k$$

take the most significant terms, discarding the lower order terms and the equation becomes

$$\gamma = \sum_{k=1}^{n} (2-2\alpha_k)n^2 - \sum_{k=1}^{n} (2-2\alpha_k)2nk + \sum_{k=1}^{n} (2-2\alpha_k)k^2 \tag{7.2}$$

At this point the equation clearly shows that the complexity depends on the size of the matrix and the sparsity, but it also depends on how the sparsity changes. Each iteration the sparsity, α, changes as the fill-in increases. In order to capture this concept the sparsity value has a k subscript. The original sparsity of the matrix is α_1 and the sparsity after the first condensation is α_2. The change in sparsity is due to the situation when a row lead and column lead are non-zero ($a_{i1} \neq 0$, $a_{1j} \neq 0$), but the actual matrix entry associated with those two is zero ($a_{ij} = 0$). When the equation $a'_{ij} = a_{11}a_{ij} - a_{i1}a_{1j}$ is applied, a_{ij} changes from zero to some non-zero value. How often this situation arises depends on the specific matrix.

This situation is fill and an expected average for the Cramer's rule algorithm can be captured with the following equation.

$$fill = ((1-\alpha)(n-k-1) \times (1-\alpha)(n-k-1)) \times \alpha \tag{7.3}$$

Once the expected fill is available this can be used to estimate the sparsity for the next condensation step.

$$\alpha_{k+1} = \frac{\alpha_k(n-k-1)^2 - fill_k}{(n-k-1)^2}$$

$$\alpha_{k+1} = \frac{\alpha_k(n-k-1)^2 - \alpha_k(1-\alpha_k)^2(n-k-1)^2}{(n-k-1)^2}$$

$$\alpha_{k+1} = \alpha_k - \alpha_k(1-\alpha_k)^2$$

$$\alpha_{k+1} = \alpha_k - \alpha_k + 2\alpha_k^2 - \alpha_k^3$$

$$\alpha_{k+1} = 2\alpha_k^2 - \alpha_k^3 \tag{7.4}$$

Now that the change in sparsity is captured this can be combined with equation 7.1.

The actual fill for a matrix is highly dependent on the structure of the matrix. Figure 7.2 shows two sparse matrices used to test the algorithm. The left matrix is referred to as rajat12 and represents a circuit simulation problem. It's a size of 1879 rows and columns and a sparsity of .996. The matrix on the right is named bcsstk26 and is a structure problem. It has a size of 1922 and a sparsity of .992. The matrix size and sparsity are similar but the problem types and the appearance of the matrices are different.

These differences cause very different fill amounts for the sparse Cramer's rule algorithm. The left matrix, rajat12, has a fill count of 3,933,818 while the right matrix, bcsstk26, has a fill count of 659,789. As will be seen in section 7.4, this disparity causes the run times for these two similar sized matrices to be drastically different. It's also interesting that while the difference in fill greatly affects the difference in run times for the sparse Cramer's rule algorithm, it has less of an impact when the dense version of the algorithm is used.

The difference between these two matrices is that the right matrix, bcsstk26, is a symmetric matrix, while the other is a general matrix. The sparse Cramer's rule algorithm consistently has less fill and better run times when the matrix structure is symmetric. This is due to the fact that the row and column leads that have non-zero values will generally intersect at the diagonal, rarely causing fill.

7.3 Switching to Dense Solver

Unlike LU-factorization, the proposed algorithm cannot easily rearrange the order of the matrix columns. This prevents the use of techniques that could reduce the fill-ins during the condensation of the matrix. This limitation coupled with the pivoting of the largest value into the lead position generates fill. As the matrix is condensed it

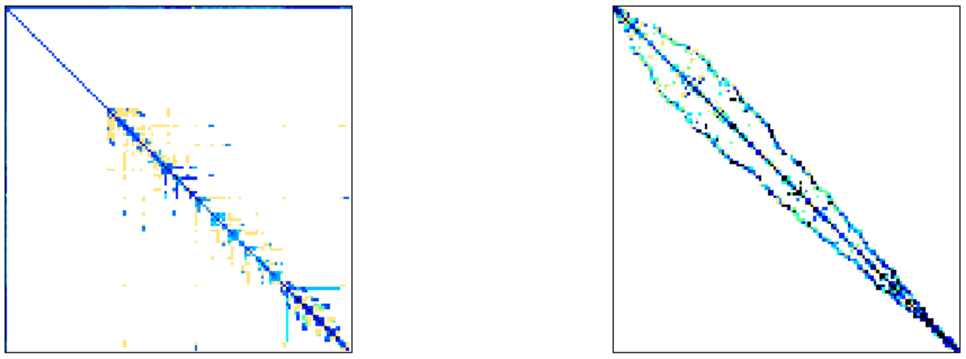


Figure 7.2: Similar sparsity and size matrices with different fill amounts (left: rajat12, right: bcsstk26)

generally becomes more dense, hence at some point it becomes more efficient to use the dense form of the proposed algorithm.

This creates an opportunity for the sparse parallel implementation of Cramer's rule. As described in section 6.3, the algorithm collects all the condensed columns to a single processor to continue the condensation without the need for any additional communication. If this collection point is determined in part by when the matrix reaches a dense level, the algorithm can continue solving the matrix with the non-sparse version of code. This allows for an optimized version using SSE instructions to continue the condensation and compute the variables.

7.4 Sparse Implementation Results

The sparse version of the Cramer's rule algorithm is missing a number of optimizations and is therefore not competitive with a true sparse server such as MATLAB or CSparse [10]. It does not utilize a compressed memory structure or the SSE instructions which would drastically improve the code performance. Despite lacking such optimization steps, it performs better than a dense solver on the same sparse problems.

Matrix Name	Sparsity Level	Matrix Structure	Matrix Size
Schenk_IBMNA/c-20	0.9976	Symmetric	2921x2921
Rajat/rajat12	0.9964	General	1879x1879
HB/bcsstk26	0.9918	Symmetric	1922x1922
Norris/heart2	0.8756	General	2339x2339
Dense Matrix (control)	0.0997	General	1000x1000

Table 7.1: Sparse matrices used for testing sparse Cramer's rule algorithm

Matrix Name	Sparsity Level	Dense Alg Time (sec)	Sparse Alg Time (sec)	Run Time Ratio
Schenk_IBMNA/c-20	0.9976	253.22	43.83	.17
Rajat/rajat12	0.9964	64.11	34.07	.53
HB/bcsstk26	0.9918	73.92	5.61	.08
Norris/heart2	0.8756	122.36	62.67	.51
Dense Matrix (control)	0.0997	8.67	17.32	2.00

Table 7.2: Comparison of Sparse Cramer's rule algorithm and dense Cramer's rule algorithm on sparse matrices

Table 7.1 shows the details of the matrices used to test the sparse version of the proposed algorithm. Please note that the dense implementation of the Cramer's rule algorithm uses a Chio step size of 1 when solving sparse matrices. This is due to the occasional inability to find a satisfactory sorting of rows for the pivot. A larger Chio step size may not find a non-zero lead without adjusting the heuristic that selects the appropriate pivot rows.

The sparse matrices are taken from *The University of Florida Sparse Matrix Collection* [11]. All these matrices are from actual problem sets and are stored in a number of formats. One of the condensed types is the Matrix Market format. The sparse implementations of Cramer's rule algorithm uses this format to load the matrices into memory. As mentioned earlier, the matrices are loaded into a full matrix that has been initialized to zero, and not stored in a compressed format. The sparse matrix collection also stores the matrices as a *mat* file that can be opened in MATLAB.

Matrix Name	Fill Amount	MFLOP Count	Matlab Time (sec)	Sparse Time (sec)	Run Time Ratio
Schenk_IBMNA/c-20	6,604,122	6479	8.67	43.83	5.05
Rajat/rajat12	3,933,818	4410	3.89	34.07	8.76
HB/bcsstk26	659,789	113	2.23	5.61	2.52
Norris/heart2	5,931,208	8470	7.51	62.67	8.34
Dense Matrix (control)	145,085	1562	.91	17.32	19.03

Table 7.3: Fill and Flop counts for test matrices

Each matrix has a specific name and details on the actual problem it represents. The sparsity level of the matrix gives a ratio of how much of the matrix is filled with zeros. A matrix that reaches a sparsity of one is completely sparse with no non-zero elements and a matrix of sparsity zero is completely dense with all non-zero elements. A small number of matrices was selected to test the algorithm. Different structure, sizes and levels of sparsity were selected for comparison.

The sparse version of the algorithm consistently outperforms the dense version except in the case of a dense matrix. The last entry in table 7.2 reports results pertaining to a dense matrix. In this case the sparse algorithm takes nearly twice as long to run as the dense version. This is due to the extra overhead of checking for sparsity and lack of SSE instruction optimizations. The dense time is also with a Chio step size of one. If a step size of four were used the execution times would drop to the range mentioned earlier and create even a larger disparity.

The run times differ greatly even for similar sized matrices. This is a result of the fill discussed earlier. Sparse matrices that have little fill like c-20 and bcsstk26 show reduced run times when compared to the dense run times. These are the matrices with symmetric structures which results in less fill for the sparse Cramer's rule algorithm. Table 7.3 shows the Fill amounts for each matrix, as well as the MFLOPS for these examples. As expected the run times closely track the MFLOP count in the serial implementation. The last two columns compare the run times to MATLAB for the same sparse matrices. Please note that MATLAB was forced to run with full matrices, instead of the typical compressed matrix storage.

Number of Processors	Dense Parallel Run Time (sec)	Sparse Parallel Run Time (sec)	Run Time Ratio
2	30.47	11.97	.39
4	17.07	8.17	.48
8	20.81	8.41	.40

Table 7.4: Parallel sparse Cramer's rule algorithm results on Schenk_IBMNA/c-20 matrix

The final testing was a sparse version of the parallel Cramer's rule algorithm. Table 7.4 shows the run times for the c-20 sparse matrix. This is a symmetric matrix from a non-linear optimization problem. The sparse version of the proposed algorithm shows the run time to solve this problem. Both the dense and sparse run times show a longer run time for 8 nodes than 4 nodes. This is due to the excess communication that cannot be hidden by overlapping computation with communication. As with the dense parallel implementation of the Cramer's rule algorithm a true broadcast message passing scheme would be needed to realize the best run times.

Chapter 8

Conclusions

This dissertation explored a novel approach for solving large linear systems that exhibits low-communication overhead for true broadcast communication platforms. The benefits of the proposed framework are attributed to a highly regular computation and data exchange patterns, when compared to commonly used parallel LU-factorization schemes. The fact that additional processing resources do not significantly impact the amount of communication suggests that this methodology has great potential as a scalable parallel linear systems solver. The use of Chio's matrix condensation, combined with a customized mirroring scheme and Cramer's rule, allow the algorithm to vastly reduce communication overhead at the cost of roughly two and a half the computational workload. In scenarios where communication requirements restrict full utilization of processing capacity and true broadcast communication is available, this algorithm can provide a viable alternative to mainstream approaches. Future work should focus on the implementation of the proposed algorithms on modern massively-parallel computing platforms, such as graphics processing units (GPUs). Moreover, it would be interesting to explore the applicability of condensation-based Cramer's rule to other problems in linear algebra, such as matrix inversion.

8.1 Key Contributions

The original algorithm was developed and documented in [33, 34] but this research further refines the algorithm and examines several different aspects of its function.

The first refinement deals with the mirroring scheme. Previously the algorithm would substitute the solution vector or b column immediately and then proceed with the condensation and mirroring. This meant that a number of values had to be kept in memory so that the b column could be manipulated when solving with Cramer's rule at the end. This research proposes a delay in the substitution of the solution vector till the end of the condensation. This reduces the bookkeeping and provides for an intuitive application of Cramer's rule.

This research also presents the addition of the MxM step size during chio's condensation phase. This allows the algorithm to be competitive with the more mainstream linear solvers that employ matrix blocking to optimize memory access. The larger step size allows the algorithm to drastically reduce the number of memory accesses as well as slightly reduced the computational workload. This optimization helps give run-times that are competitive with algorithms that use blocking to optimize the data transferred between cache and main memory.

One of the key requirements for practical application of an algorithm is a level of confidence that the algorithm returns the correct answer. While this research does not prove that the algorithm is numerically accurate it provides a detailed error analysis that suggests that it's similar to currently used methods. Furthet evidence is presented that the algorithm is as accurate as other commonly used linear solvers. Numerous test results are provided with comparisons to solutions provided by mainstream methods.

The key potential of the algorithm is in parallel utilization. This research proposes a parallel scheme for the algorithm that retains the $\mathcal{O}(N^3)$ characteristic even in a multiprocessor environment. Evaluation of the communication complexity and actual implementations are provided for comparison to parallel implementations of LU-factorization. This proposed implementation is shown to have competitive run-times for smaller matrices, although additional optimizations are needed for the algorithm to scale as well as existing parallel methods.

Finally this research explores the algorithm's suitability for sparse problems. This includes an actual implementation both for a serial version and parallel version. The implementation requires a great deal of refinement and optimization but shows that the algorithm can support reduced computational complexity when presented with sparse problems. The research also identified particular sparse problem sets that lend themselves well to the Cramer's rule algorithm.

8.2 Relevant Publications

- K. Habgood, I. Arel, Revisiting Cramer's Rule for Solving Dense Linear Systems in Proc. *ACM High Performance Computing Symposium (HPC 2010)*, April, 2010.
- K. Habgood, I. Arel, A Condensation-based Application of Cramer's Rule for Solving Large-Scale Linear Systems. Accepted for publication in *Journal of Discrete Algorithms*
- In Preparation: A Parallel Linear Solver Utilizing Cramer's Rule
- Wikipedia update: Clarification that Cramer's rule can be implemented in $\mathcal{O}(N^3)$. (http://en.wikipedia.org/wiki/Cramer's_rule)

8.3 Future Directions

This research provide a theoretical understanding and analysis of the algorithm, and give a solid basis for practical implementation. In order to realize the full potential of the algorithm further development and testing is required in a few areas. The obvious potential of the algorithm is as a low-communication parallel solver and the future directions should focus on that potential.

The algorithm needs an environment that can provide a true broadcast message passing scheme. As discussed in section 2.5.2 the implementations presented use mainstream parallel communication software. The MPICH2 package does not implement messaging that takes advantage of the native broadcasts in networking hardware. Instead the software uses a more reliable form of one-to-one message passing. The algorithm needs to be implemented in a manner where it can leverage a true one-to-many message passing infastructure.

In addition the parallel coding of the algorithm needs further optimization. The tuning of the code for memory access would help reduce the runtimes considerably when compared to other parallel solvers. In addition, the code could futher exploit the difference between parallel nodes that are on the same computer, multicore processors, when compared to nodes across a network. Nodes that are on the same core could take advantage of shared memory instead of the network message passing schemes.

The algorithm also needs further optimizations for sparse matrices. The implementation provided focuses on the theoretical potential to solve sparse matrices rather than the practical application. The coding would need to employ an optimized memory strucutre for the storage of the algorithm. This would reduce the memory access times considerably and thus reduce the overall runtimes of the algorithm. The algorithm should also be Further investigated for opitmal sparse matrices. The research presented clearly shows that the algorithm is well suited to some sparse problems, as shown in table 7.3. However, a better understanding of which matrix types are best stuited would identify when the algorithm could best be applied.

Bibliography

Bibliography

[1] Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing*. Dover. pg. 10. 36

[2] Aitken, A. C. (1956). *Determinants and Matrices, Sixth edition*. Oliver and Boyd. 30

[3] Allan, R. J., Hu, Y. F., and Lockey, P. (1999). Parallel application software on high performance computers. survey of parallel numerical analysis software. 25

[4] Amoura, A. K., Bampis, E., and Konig, J.-C. (1998). Scheduling algorithms for parallel gaussian elimination with communication costs. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(7):679–686. 22

[5] Armistead, R. B. (2010). Mpi-based parallel solution of sparse linear systems using chios condensation algorithm and test data from power flow analysis. Master's thesis, University of Tennessee. 12

[6] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ix, 18, 25

[7] Chapra, S. C. and Canale, R. (1998). *Numerical Methods for Engineers, Second Edition*. McGraw-Hill Inc. 11

[8] Choi, J., Demmel, J., Dhillon, I. S., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D. W., and Whaley, R. C. (1995). Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In Dongarra, J., Madsen, K., and Wasniewski, J., editors, *PARA*, volume 1041 of *Lecture Notes in Computer Science*, pages 95–106. Springer. 26

[9] Ciegis, R., Starikovicius, V., and Wasniewski, J. (2000). On the efficiency of scheduling algorithms for parallel gaussian elimination with communication delays. In Sørevik, T., Manne, F., Moe, R., and Gebremedhin, A. H., editors, *PARA*, volume 1947 of *Lecture Notes in Computer Science*, pages 74–81. Springer. 17

[10] Davis, T. A. (2006). *Direct Methods for Sparse linear Systems*. SIAM. 67, 74

[11] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/. 75

[12] D'Azevedo, E. F. and Dongarra, J. (2000). The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. *Concurrency - Practice and Experience*, 12(15):1481–1493. 8

[13] Desprez, F., Dongarra, J., and Tourancheau, B. (1995). Performance study of LU factorization with low communication overhead on multiprocessors. *Parallel Processing Letters*, 5:157–169. 15

[14] Dianant, S. A. and Saber, E. S. (2009). *Advanced Linear Algebra for Engineers with Matlab*. CRC Press. pp. 70-71. 10

[15] Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct Methods for Sparse Matrices*. Oxford University Press. 67

[16] Dunham, C. (1980). Cramer's rule reconsidered or equilibration desirable. *ACM SIGNUM Newsletter*, 15(4):9. 11

[17] Eves, H. W. (1980). *Elementary Matrix Theory*. Courier Dover Publications. 13

[18] Fu, C., Jiao, X., and Yang, T. (1997). A comparison of 1-d and 2-d data mapping for sparse lu factorization with partial pivoting. In *PPSC*. SIAM. 17

[19] George, J. A. and Ng, E. G. (1985). An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Statist. Comput.*, 6:390–409. 69

[20] George W. Collins, I. (2003). *Fundamental Numerical Methods and Data Analysis*. George W. Collins, II. 3, 9

[21] Gilbert, J. R. (1994). Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 15(1):62–79. 68

[22] Govindaraju, N. K., Henson, M., and Manocha, D. (2005). Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *ACM/IEEE SC05 Conference*, pages 12–18. 8

[23] Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison-Wesley Publishing; 2nd edition, One Jacob Way, Reading, MA 01867-3999, second edition. ix, 24

[24] Heath, M. T. (1997). *Scientific Computing: An Introductory Survey*. McGraw-Hill, pub-MCGRAW-HILL:adr. 43

[25] Hendrickson, B. A. and Womble, D. E. (1994). The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5):1201–1226. 17, 20

[26] Higham, N. J. (1996). *Accuracy and Stability of Numerical Algorithms*. Society for Indus- trial and Applied Mathematics - SIAM. 4, 10, 11

[27] Intel_Corporation (2007). Intel microprocessor export compliance metrics. 52

[28] Karniadakis, G. E. and II, R. M. K. (2003). *Parallel Scientic Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press. ix, 8, 15, 16, 17

[29] Kushilevitz, E. and Nisan, N. (1997). *Communication Complexity*. Cambridge University Press. 6

[30] Levon, J. (2000). *Oprofile Manual*. Victoria University of Manchester. 49

[31] Mathews, J. H. (2003). Module for parabolic p.d.e.'s. http://math.fullerton.edu/mathews/n2003/cranknicolsonmod.html. 45

[32] Moler, C. (1974). Cramer's rule on 2-by-2 systems. *ACM SIGNUM Newsletter*, 9(4):13–14. 11

[33] Nagari, A. (2009). Parallel processing architecture for solving large scale linear systems. Master's thesis, University of Tennessee, http://trace.tennessee.edu/utkgradthes/53. 12, 79

[34] Nagari, A., Elhanany, I., Thompson, B., Li, F., and King, T. (2008). A parallel processing architecture for solving large-scale linear systems. In Arabnia, H. R. and Mun, Y., editors, *PDPTA*, pages 307–312. CSREA Press. 12, 79

[35] Oracle_Corporation (2010). x86 assembly language reference manual. 51

[36] Ortega, J. M. (1972). *Numerical Analysis A second Course*. Academic Press. 42

[37] Palvolgyi, D. (2010). Communication complexity. 15

[38] Press, W. H., Teukolsky, S. A., and Vetterling, W. T. (1996). *Numerical Recipes in Fortran 77 and Fortran 90: The Art of Scientific and Parallel Computing*. Cambridge University Press, pub-CUP:adr. 8

[39] Quarteroni, A., Sacco, R., and Saleri, F. (2000). *Numerical Mathematics*. Springer Verlag. 4

[40] Recktenwald, G. W. (2011). Finite-difference approximations to the heat equation. web.cecs.pdx.edu/ gerry/class/ME448/codes/FDheat.pdf. 45

[41] Saad, Y. (1986). Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Applications*, 77:315–340. Special volume on parallel computing. 20

[42] Sears, C. B. (2000). The elements of cache programming style. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 18–18, Berkeley, CA, USA. USENIX Association. 5

[43] Sridhar, M. K. (1987). A new algorithm for parallel solution of linear equations. *Inf. Process. Lett.*, 24(6):407–412. 11

[44] Wadsworth, D. M. and Chen, Z. (2008). Performance of mpi broadcast algorithms. In *IPDPS*, pages 1–7. IEEE. ix, 20, 21

Vita

Ken Habgood was born in St. Petersburg FL, where he attended the International Baccalaureate (IB) program at St. Petersburg High School. After completing his IB diploma he enrolled at the University of Florida (UF) as a Florida Academic scholar recipient. He was selected for the UF honors program and completed a B.S. in Environmental Engineering. Ken then pursued a Master's of Science in Computer Engineering at the same institution. After graduation he worked in the telecommunications field as a network planner designing public data networks for Sprint. He married Katie Yellen, also from St. Petersburg and settled in Altamonte Springs, FL. In 2005 he and his wife relocated to Tennessee and began a career with Knoxville Utilities Board managing the SCADA system that controls the electric, gas and waste-water systems in Knoxville. Shortly after arriving in Knoxville he enrolled at the University of Tennessee to pursue a doctorate in computer engineering focusing on parallel computing. He has two daughters named Adeline and Alice who have no interest in computer engineering or parallel computing whatsoever but are still willing to hang out with their father occasionally.