Concurrent programming and composite Newton methods

Craig D. S. Thompson cdt830@mail.usask.ca
Raymond J. Spiteri spiteri@cs.usask.ca

Numerical Simulation Laboratory
Department of Computer Science, University of Saskatchewan
Saskatoon, Saskatchewan, Canada, S7N 5C9

June 26, 2008

Abstract

The most widely used, robust, and general-purpose numerical methods for approximating the solution to systems of nonlinear algebraic equations (NAEs) are based on Newton's method. Many variants of Newton's method exist in order to take advantage of problem structure. However, no Newton variant converges quickly for all problems and initial guesses. It is generally impossible to know a priori which variant of Newton's method will be effective for a given problem: some variants and initial guesses may not lead to convergence at all, or if they do, the convergence may be extremely slow. New multi-core computer architectures allow the use of multiple Newton variants in parallel to potentially enhance the overall convergence for a given problem. For example, by sharing intermediate results each variant can make use of the best iterate generated thus far. This results in a sequential combination of Newton variants that we call a composite Newton method. In this paper, we survey concurrent programming techniques, describe an implementation of composite Newton methods, and give some experimental results.

1 Introduction

Concurrent programs, programs that have more than one thread of control, have been of interest to computer scientists since the mid 1960s. Concurrent programs can be used to distribute work to multiple processors in order to make a program faster, do more work in the same amount of time, or perform independent tasks simultaneously. These programs can run on single processor computers, which can only do one task at a time, but the processes are interleaved in such a way that at a high level (i.e. from the user's perspective) the processes are perceived as executing at the same time.

There are two major styles of concurrent programs that were created to operate in two distinct hardware environments. The first type is *shared variable programming*, in which multiple processes communicate with each other via shared memory. This usually requires that processes execute on the same physical machine. This machine may have multiple processors, but the processes communicate through the memory of the computer. The second type of concurrent programming makes use of *message passing*. Message passing programs send information over a network to processes running on other computers. It should be noted that it is possible to have a distributed shared memory system, or to use message passing to communicate with local processes.

2 Literature Review

2.1 Independent processes

Before we discuss methods of concurrent programming, it is important to recognize that programs can be completely independent of each other in that they have no interactions. If two programs do not interact then there are no concurrency issues of concern. Bernstein [2] described conditions for two programs to be independent of each other based on the inputs and outputs of the programs. These inputs and outputs are often called the *read set* and the *write set* because they represent the set of files, memory addresses, and devices that are read or written. Bernstein's three conditions are that: (1) the read set of the first program must be disjoint from the write set of the second program, (2) the write set of the first program must be disjoint from the read set of the second program, and (3) both programs write sets must be disjoint. Thus, one program cannot be reading from the same file that another program is writing to, and both programs cannot be writing to the same file. If these conditions are met there is no need to use the methods described below to control the interactions of the programs.

2.2 Shared Variable Programming

When processes are interacting, there are sections of their programs that must be executed as a single operation. However, these sections are usually longer than a single hardware level instruction and thus we need a way to ensure that the whole section is executed as one action; this is also known as *atomic execution*. The standard example is reading or writing a character array containing a word. Whenever we read the word we want to read the whole word atomically. Additionally, if we write a new word to the array we want the whole word to be written atomically. It is undesirable for one process to be reading while the other process is writing. The two actions must happen in sequence rather than in parallel or else the word read may contain part of the original word and part of the newly written word. These sections of code that need to be executed atomically are called *critical sections*, and ensuring that they execute atomically

is called the *critical section problem*. Dijkstra [5] introduced the critical section problem, the requirements that must be met by a solution to the problem, and a solution. Dijkstra's requirements state that a solution should be symmetric in that it does not consistently give preference to one process over another. Additionally, Dijkstra states that processes not wanting to access the critical section should not inhibit others from accessing it. Finally, Dijkstra states that if multiple processes want to access the critical section they should determine which one will enter the critical section in a finite number of steps. Knuth [15] adds to Dijkstra's requirements by showing that one process could wait indefinitely if using Dijkstra's solution; this is known as *starvation*. Knuth's solution takes into consideration the notion of "taking turns" to solve the problem of starvation. If multiple processes want to enter the critical section at one time, one of them is given preference based on whose turn it is. After entering the critical section on one's turn, the preference is then given to another process, which is chosen in a round-robin order. Lamport [16] made further improvements with the so-called "bakery algorithm", which simulates customers taking sequentially numbered tickets as they enter a bakery and being served in the order of the numbers on the tickets. The bakery algorithm does not rely on a shared array to indicate which processes want to enter the critical section, as the previous solutions do. Instead, when a process wants to enter the critical section it takes a number one greater than the number held by any other process, and processes are allowed to enter the critical section in order of the numbers they hold. The main benefit of this algorithm is that processes enter the critical section in a first come, first served order. Lamport also claims that this algorithm is more robust with respect to hardware failure.

There are other methods of controlling access to a specific section of code that, although not more powerful than Dijkstra's solution, can make the programmer's job much easier. *Locks* are the abstract name of Dijkstra's [5] solution to the critical section problem, in which processes acquire a lock, perform some task, and release a lock.

Semaphores [8] are variables that are used for protecting access to shared resources. Semaphores may be incremented with the "S()" operation or decremented with the "P()" operation. Both P and S are atomic, and a P() operation that would give the semaphore a negative value will block until that is no longer the case. As an example, semaphores are used in the classic producers/consumers problem, wherein a producer continually places a value into a shared queue and then increments the semaphore. The consumer will try to decrement the value of the semaphore and then remove the value from the shared queue. Thus, the producer will increment the value of the semaphore each time it creates a new item and the consumer will attempt to decrement the value of the semaphore before consuming an item. The advantage of using semaphores for this purpose is that the producer and consumer are each able to work at their own speed. If the producer is producing faster than the consumer is consuming, the values are just stored in the queue. Conversely, if the consumer's speed increases it can use up any values that the producer previously added to the queue. Thus, semaphores may be used to ensure mutual exclusion

or to signal events.

Dijkstra [6] shows solutions to numerous concurrent programming examples by using semaphores and introduces the now classic "sleeping barber" problem. Courtois, Heymans, and Parnas [4] are credited with describing the "readers" "writers" problem, in which several processes want to read a shared variable and several processes want to write to the shared variable. In the readers writers problem multiple readers are allowed in the critical section at once, but only one writer may be in the critical section at a time. Additionally, readers and writers are not allowed to be in the critical section at the same time. Courtois and colleagues present two solutions to the problem using semaphores, including one that gives preference to readers in the case where both a reader and a writer want access to the shared variable, and another where the writers are given preference.

Monitors are a higher level abstraction of these same principles of protected access in which the programmer does not need to explicitly indicate how to protect access, but rather what should be protected [12]. A monitor is a class that has private variables and methods to manipulate them. Access to the methods of the monitor is mutually exclusive, so only one process can be invoking one monitor method at a time. Monitors also have condition variables that allow processes to be suspended until they are signaled and to resume action of other processes by signaling that some condition has occurred. Having outlined how processes share data via shared memory and how their access can be controlled using locks, barriers, semaphores, and monitors, we now move to a discussion of correctness.

The issue of proving program correctness is an important one, especially with respect to concurrent programs, where the order of execution can greatly affect the outcome. Hoare [11] is credited with devising a formal logic for proving partial correctness of sequential programs; that is, if a program terminates, its results will be correct. Additionally, Orwicki and Gries [21] [20] are credited with extending Hoare's logic for proving correctness to concurrent programs and describing the *at-most-once property.* An assignment statement in a program that meets the at-most-once property will execute atomically, regardless of whether this is explicitly stated, whereas assignment statements not meeting the at-most-once property will not be executed atomically, unless specifically stated using one of the above techniques.

2.3 Distributed Programming

Distributed programming is a technique primarily used to communicate between processes that cannot share variables, such as processes located on separate computers, connected via a network. There are four major techniques of distributed programming: synchronous and asynchronous message passing, remote procedure call (RPC), and Rendezvous.

Message passing consists of functions to send and receive information. When using asynchronous message passing, the *send* operation is non-blocking in that the process sends its message and continues executing. The "receive" oper-

ation, on the other hand, is a blocking operation. If there is currently no message ready to be received, the process waits until there is a message to receive. However, if there is a message waiting to be received then it will be received, and execution will continue. When using synchronous message passing the send operation will send the data and then wait for the message to be received. Once it is received, execution of the sending process will resume. Brinch Hansen [10] provides an early message passing system, similar to current models, designed for the RC 4000 computer. Brinch Hansen's description allows for client/server asynchronous communication. Lauer and Needham [17] show the duality between monitors and message passing, proving that a program based on one method can be implemented using the other method, and thus monitors and message passing are equal in their power over synchronization and mutual exclusion. Lauer and Needham's findings are foundational in that if message passing is of equal power to monitors, then the set of problems solvable on a shared memory system can be solved on a distributed system and vice versa.

Hoare [13] introduced *synchronous communication* and *guarded communication*. Guarded communication allows a process to conditionally accept an incoming message based on the value of a Boolean expression, known as the *guard*. Additionally, this guarded acceptance of messages allows for non-deterministic choice of which message to accept when there are multiple messages available.

Carriero and Gelernter [3] provide a good description of three different types of parallelism that can be achieved using concurrent programming techniques, and provides implementations using message passing in Linda. They describe *result parallelism* as assigning a process to each part of the final result and having them compute that portion. Additionally, they describe *specialist parallelism*, in which each process is designed to do a certain task and performs all instances of that task. Finally, they describe *agenda parallelism*, in which a list of tasks is generated and all processes work on the same task until it is complete, then they all move onto the second task. Furthermore, Cariero and Gelernter note that these conceptualizations do not fit all tasks, and that the distinction between techniques can become blurred.

Remote Procedure Call (RPC) [19] is a system that allows for a server consisting of remotely callable procedures and local processes. When the client invokes one of the server procedures, a process is dynamically created to serve the request. Additionally, multiple clients can invoke server procedures at one time. When a new server process is invoked, the client process waits for the server process to terminate before continuing.

By contrast, *Rendezvous*, as implemented in ADA, consists of a single server process that receives requests and performs the desired actions in sequence. Thus, with Rendezvous only one client's request is serviced at a time; there is no dynamic creation of processes to server requests. The concepts of synchronous and asynchronous message passing can be combined with rendezvous and RPC, resulting in the "multiple primitives notation" [1]. Multiple primitives allow for RPC and rendezvous to be invoked by either a synchronous send, resulting in the actions described previously, or, invoked by an asynchronous send, resulting in dynamic process creation, in the case of RPC, or asynchronous message passing

in the case of rendezvous.

2.4 Newton's Method

Mathematical models used in science and engineering often require the solution of Nonlinear Algebraic Equations (NAEs). Some areas where NAEs are used include sound or heat flow, fluid flow, electromagnetism, astronomy, chemical reactions, and wave propagation. We denote a system of NAEs by

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}, \tag{1}$$

where $\mathbf{F}(\cdot) : \Re^m \rightarrow \Re^m$ is called the *residual*, $\mathbf{x} \in \Re^m$ is the vector of *unknowns*, and $\mathbf{0}$ is a vector of zeros.

The most widely used, robust, and general-purpose numerical methods for approximating the solution to systems of nonlinear algebraic equations (NAEs) are based on Newton's method. The classical Newton's method is defined as

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \mathbf{J}_{\mathbf{F}}^{-1}(\mathbf{x}^{(n)})\mathbf{F}(\mathbf{x}^{(n)}),\ n = 0, 1, 2, \ldots, \tag{2}$$

where $\mathbf{x}^{(n)}$ is the nth approximation to the solution of (1), and $\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})$ is the *Jacobian matrix* evaluated at $\mathbf{x}^{(n)}$.

Inversion of the Jacobian matrix in (2) is not done in practice. Instead we solve

$$\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})\mathbf{d}^{(n)} = -\mathbf{F}(\mathbf{x}^{(n)}), \tag{3}$$

where $\mathbf{d}^{(n)}$ is called the *Newton direction*. Then,

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \mathbf{d}^{(n)}. \tag{4}$$

There are many variants of Newton's method that take advantage of the structure of a particular problem. The variants of Newton's method are formed by choosing a termination criterion for (2), an algorithm for computing the Newton direction, a forcing term, a globalization strategy, and an initial iterate. However, no Newton variant converges quickly for all problems and initial guesses, and it is generally impossible to know beforehand which variant of Newton's method will be optimal for a given problem. Furthermore, some variants and initial guesses will have disastrous results because they may not lead to convergence or the convergence may be extremely slow.

3 Composite Newton Methods

The physical limits of semiconductor based processors have been reached. The response has been to produce multiple processors on a single chip. Dual core systems are now standard and quad core or 8-core systems are becoming common for workstations and servers. It is expected that the number of processor cores will continue to increase. Given that the current trend is toward many-processor machines, one of the ways to make use of multi-core processors to

quickly solve NAEs is to use multiple Newton variants (solvers) in parallel to attempt the same problem, thereby decreasing the likelihood that all variants will fail or converge extremely slowly. Furthermore, there may exist common measures of progress between Newton variants that could allow for some degree of co-operation, wherein intermediate results produced by one variant could be used by another. By sharing intermediate results each variant can make use of the closest approximation generated thus far. This process of sharing intermediate approximations results in a sequential combination of Newton variants called a composite Newton method [22].

With the composite Newton method we aim to solve the same set of NAEs with several different variants simultaneously. We hypothesize that by sharing intermediate information (e.g., $\mathbf{x}^{(n)}$) a problem can be solved faster than is possible with a non composite newton method.

We investigate three criteria to determine if a given solver should share its partial solution with other solvers, or if the solver should proceed by using another solvers partial solution. These criteria are based on the residual norm, rate of convergence, and a combined residual norm and rate of convergence term.

Previous work by Ter, Donaldson, and Spiteri [22] investigated a greedy algorithm to form a composite Newton method based on minimum residual norm. Their work involved a serialized simulation of parallel execution due to hardware limitations at the time of experimentation. Ter, Donaldson, and Spiteri found that composite Newton methods can be faster than the fastest Newton variant from which they are formed.

The present research focuses on how the computational overhead of sharing partial results between variants can be minimized while maximizing the increase in speed of convergence. A *specialist parallelism* shared variable approach making use of locks is explored. Although there is an overhead associated with sharing partial results, the composite method may converge more quickly than Newton variants in parallel without cross communication.

4 Design and Methodology

4.1 pythNon Problem Solving Environment

pythNon is a Problem Solving Environment for NAEs Developed in the Numerical Simulation Laboratory at the University of Saskatchewan. It includes a range of solution methods, and allows for quick coding of solution methods and problem definitions. The Newton variants used are all defined within pythNon.

4.2 Coordination of Newton variants

We used a coordination program written in the python programming language to share information between variants. pythNon includes facilities to write the current iterate to a file so the "best" iterate was stored using a file. In order

to minimize the performance effects of file I/O, a RAM disk was used to store the current iterate file. RAM disks allow a portion of RAM to be treated as though it were a file system on a hard disk drive, thereby greatly improving performance.

Each pythNon solver is an independent process and there is one multi-threaded coordinator process. The coordinator process has one thread per pythNon solver. Each coordinator thread communicates via message passing with one pythNon process. The coordinator threads make use of locks and shared variables to communicate with each other. Python has a feature called the Global Interpreter Lock (GIL) that protects threads within a process from concurrent access to shared objects. The impact of the GIL is that a python process is limited in its scaling abilities via threading to full utilization of one CPU core. This is not a problem for the coordinator process because the coordinator threads do very little work; full utilization of one CPU core is more than enough compute power for coordination in our case. Threads are more lightweight in terms of their resource needs than processes and threading allows for use of shared variables, rather than broadcasting messages to all other coordinator threads. If coordinator processes where used, rather than threads, there would be many more context switches when running the program, which may impact performance. The use of both multiple threads and multiple processes is justified by our need for more compute power than one process can provide, while minimizing the amount of resources required for coordination.

4.3 Selection Criteria

We explored three selection criteria: residual norm, rate of convergence, and a combined residual norm and rate of convergence factor formed by multiplying the two other criteria together. The residual norm is a measure of the "distance" between the current iterate and the actual solution. We have defined the rate of convergence as the current residual norm, divided by the residual norm of the previous iterate.

$$||\mathbf{F}(\mathbf{x}^{(n)})|| \tag{5}$$

$$\frac{||\mathbf{F}(\mathbf{x}^{(n)})||}{||\mathbf{F}(\mathbf{x}^{(n-1)})||} \tag{6}$$

$$\frac{||\mathbf{F}(\mathbf{x}^{(n)})||^2}{||\mathbf{F}(\mathbf{x}^{(n-1)})||} \tag{7}$$

4.4 Problem Set

We examined 14 problems available from the pythNon problem set. Many of these problems are synthetic benchmark type problems, such as the Tridiagonal system (8) [18] used by Ter, Donaldson, and Spiteri. Of the problems used, the most realistic are the Bratu problem (9) [9] and the Two-dimensional,

steady-state, convection-diffusion equation (10) [14]. The NAEs for the Bratu are formed by discretizing the spatial operators by centered differences and incorporating the boundary conditions. The NAEs for the convection-diffusion problem are formed by discretizing on a uniform mesh with 500 interior grid points in each direction using centered differences.

$$\begin{aligned} F_1(x) &= 4(x_1 - x_2^2), \\ F_i(x) &= 8x_i(x_i^2 - x_{i-1}) - 2(1 - x_i) \\ &\quad + 4(x_i - x_{i+1}^2),\ i = 2, 3, \ldots, m-1, \\ F_m(x) &= 8x_m(x_m^2 - x_{m-1}) - 2(1 - x_m), \end{aligned} \tag{8}$$

$$\begin{aligned} \Delta u + \kappa \frac{\partial u}{\partial x_1} + \lambda e^u &= 0 \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \\ \Omega \equiv [0,1] \times [0,1] &\subseteq \mathbb{R}^2 \end{aligned} \tag{9}$$

$$\begin{aligned} -\Delta u + 20u(u_x + u_y) &= f(x,y) \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \\ \Omega \equiv [0,1] \times [0,1] &\subseteq \mathbb{R}^2 \end{aligned} \tag{10}$$

5 Results

All tests where performed on a system with 8 cores and 128GB RAM. Each problem was solved 10 times, with the shortest time recorded. By taking the shortest time we obtain the value with the least noise caused by other users or system processes. We have reported the fastest time recorded for the classical methods in seconds. The timing of the composite methods are reported as % faster or slower than the fastest classical solver tested. Scores in red indicate that the composite solver was 10% or more slower than the fastest classical solver, while scores in blue indicate that the composite solver was 10% or more faster than the fasted classical solver. The columns are labeled R, C, and RC corresponding to the residual norm, rate of convergence, and combined selection criteria, respectively.

Problem	m	$\mathbf{x}^{(0)}$	R	C	RC	Classical
Tridiagonal	5000	−2	12.90	19.35	13.49	3.41
		0	−1.69	9.28	8.44	2.37
		2	2.37	9.48	11.85	2.11
Pentadiagonal	5000	−2	−133.28	−163.28	−161.72	5.80
		0	6.49	3.90	8.66	4.62
		2	−2.35	−46.08	11.76	5.10
Chandrasekhar	5000	-2	-1.85	2.54	0.69	4.33
		0	0.88	0.22	0.88	4.52
		2	−3.75	−1.04	−3.96	4.80
Rosenbrock	5000	−2	15.88	13.24	16.76	3.40
		0	−0.41	18.18	9.92	2.42
		2	−5.71	11.43	12.86	2.10
Broyden Tri.	10000	−2	0.00	0.00	0.00	0.61
		0	10.98	8.54	9.76	0.82
		2	–	–	–	–
Discrete BVP	100000	−2	−0.31	−0.78	−0.16	6.42
		0	−0.78	0.16	−0.47	6.45
		2	0.16	0.16	0.47	6.45
Struct. Jacob.	100000	−2	−1.75	−2.19	−1.32	2.28
		0	−15.02	−14.19	−15.51	6.06
		2	–	–	–	–
Powell Badly S.	100000	−1	−6.87	−4.58	−3.82	1.31
		1	−5.88	−5.88	−5.88	1.19
		5	−3.85	−2.88	−3.85	1.04
Premult. Diag.	6000	1	−2.11	−6.34	−5.63	1.42
Ext. Rosen.	100000	−5	−2.91	−3.88	−6.80	1.03
		1	−3.76	−5.38	1.61	1.86
		5	0.00	7.14	3.57	1.12
Trigonometric	6000	1	1.49	0.00	0.00	0.67
Ornstein-Zernike	10000	0	33.14	31.98	32.27	6.88
Bratu	500	0	21.23	21.42	12.91	198.89
Convect.-Diff.	500	0	19.71	14.06	13.49	275.56

5.1 Analysis and Observations

Results vary greatly depending on problem, initial guess, and method of composition. In some cases the fastest classical solver is "thrown off" by another solver when using the composite Newton method. For example, in the case of the Pentadiagonal problem, the fastest classical solver starts slowly and then speeds up. When used in the composite method, it obtains an iterate from another solver after only a few iterations, but progresses extremely slowly after taking the iterate from another solver.

The fastest solution may not always be the same combination of variants. For example, one test run may find two iterations from solver A, followed by 8 iterations of solver B to be the fastest solution, whereas, another test run may result in three iterations from solver C, followed by 7 iterations of solver A to be the fastest solution. This is because the global best solution is stored in a shared variable, and solvers compete to obtain a lock on the variable before reading or writing. The order in which solvers obtain the lock on the shared variable can change the outcome.

6 Future Work

In this work three selection criteria were explored for forming the composite Newton method. However, none of the studied selection criteria consistently out-performed the non-composite method. Therefore, other criteria for determining which solver has the "best" solution at a given time should be explored. In particular, one selection method that should be explored is the minimization of the difference between the residual norm and the local linear model.

We ran experiments on an eight core machine using six solvers, thus two cores where available for operating system processes, as well as other users. Presently, the pythNon problem solving environment is a single-threaded application, so each solver uses at most 100% of one CPU core for its computations. When we attempted to solve a problem we initially had good overall cpu utilization, 75% (six out of eight cores); however, as solvers failed the overall CPU utilization also dropped because each solver was limited to a single core. By rewriting pythNon to be a multi-process application we could ensure a higher cpu utilization. As solvers fail, the remaining solvers could spawn new processes to aid in their computations. moreover, by creating additional processes more than one processor core could be used by a single solver.

Presently, we wait for a solver to fail before it is terminated. However, there are some cases where a solver is progressing towards a solution at an extremely slow rate. By implementing algorithms for early detection and pruning of poor performing solvers we could make more CPU resources available to other solvers. This assumes that pythNon is a multi-threaded application, otherwise removal of poor performing solvers would be ineffectual.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[2] A. Bernstein. Analysis of programs for parallel processing. *IEEE transactions on Computers*, EC-15(5):757–763, 1966.

[3] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, 1971.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[6] E. W. Dijkstra. Cooperating sequential processes. published as [7], 1968.

[7] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[8] E. W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[9] R. Glowinski, H. B. Keller, and L. Reinhart. Continuation-conjugate gradient methods for the least squares solution of nonlinear boundary value problems. *SIAM Journal on Scientific and Statistical Computing*, 6(4):793–832, 1985.

[10] P. B. Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4):238–241, 1970.
[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[12] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
[13] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
[14] C. T. Kelley. *Solving nonlinear equations with Newton's method*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics (SIAM), 2003.
[15] D. E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, 1966.
[16] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
[17] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.
[18] G. Li. Successive column correction algorithms for solving sparse nonlinear systems of equations. *Math. Program.*, 43(2):187–207, 1989.
[19] B. J. Nelson. *Remote procedure call*. PhD thesis, Pittsburgh, PA, USA, 1981.
[20] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
[21] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
[22] T.-P. Ter, M. W. Donaldson, and R. J. Spiteri. Observations on greedy composite newton methods. *hpcs*, 00:11, 2007.