Pipeline Implementation of Cellular Automata for Structural Design on Message-Passing Multiprocessors*

Shahriar Setoodeh† David B. Adams‡
Zafer Gürdal§ Layne T. Watson¶

Abstract

The inherent structure of cellular automata is trivially parallelizable and can directly benefit from massively parallel machines in computationally intensive problems. This paper presents both synchronous and pipeline parallel implementations of cellular automata on distributed memory (message-passing) architectures. A structural design problem is considered to study the performance of the various cellular automata implementations. The synchronous parallel implementation is a mixture of Jacobi and Gauss-Seidel style iteration, where it is more Jacobi like as the number of processors increase. Therefore, it exhibits divergence because of the mathematical characteristics of Jacobi matrix iteration for the structural problem as the number of processors increases. The proposed pipeline implementation preserves convergence by simulating a pure Gauss-Seidel iteration. Numerical results for analysis and design of a cantilever plate made of composite material show that the pipeline update scheme is convergent and successfully generates optimal designs.

1 Introduction

The increasing availability of relatively cheap cluster machines creates an opportunity for more researchers to solve computationally intensive problems on parallel machines.

---

* This work was supported in part by the Maui High Performance Computing Center, AFOSR Grant F49620-02-1-0090, and AFRL Grant F30602-01-2-0572.

† Department of Engineering Science and Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061-0203.

‡ Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061-0106.

§ Departments of Aerospace and Ocean Engineering, and Engineering Science and Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061-0203.

¶ Departments of Computer Science and Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061-0106.

However, many traditional techniques do not parallelize well across distributed memory architectures or are too complicated to be accessible to a wide body of researchers. The cellular automata (CA) paradigm, though, is inherently suitable for large scale parallel implementations because of the nature of information flow and data storage at a local level.

The investigation of the cellular automata paradigm focuses on taking advantage of available computing resources in a scalable way. Though the paradigm can not compete with traditional methods on a serial machine, the ease and efficiency by which cellular automata can be implemented on massively parallel distributed memory architectures can provide significant speedup with a relatively low initial time investment. It is this property of cellular automata that has kindled recent research interest.

This work presents three parallel implementations of a cellular automaton and the convergence properties derived from various update styles. The synchronous parallel code for the most trivial implementation (domain decomposition into blocks of cells) is shown to become unstable as the number of processors increases. This instability is attributed to an iteration style that varies with the number of processors used. Most significantly, in the limit of one processor per cell (the pure CA operating in SIMD mode), the computation diverges for the structural design problem being considered. A pipelined implementation of the cellular automata is introduced to emulate a purely serial iteration, which converges. A second pipeline implementation is proposed that is more efficient than the first, where the efficiency gain is achieved by asynchronism. This version closely, but not exactly, emulates the convergent serial CA implementation.

Section 2 provides some background information on the cellular automata paradigm. Section 3 introduces the synchronous parallel version of the work and is followed by an example engineering application in Section 4. A numerical example for the synchronous parallel code demonstrates divergence and reveals the need for a different type of parallelization. Section 5 describes two pipelined parallel implementations that preserve the iteration style of a pure Gauss-Seidel update regardless of the number of processors used. Results and comparisons of the three parallel implementations are detailed in Section 6.

2 Cellular Automata

The cellular automata (CA) paradigm was initially introduced by Wiener et al. [1] in 1946, but remained unpopular until recent times due to limited available computing resources and the difficulty of SIMD programming. Massively parallel machines have made CA an attractive tool for many physical and mathematical applications. Currently, CA applications range from digital image processing [2] to heat transfer [3] to structural design [4] to traffic simulation [5]. A general overview of the CA methodology is presented in [2].

In the cellular automata paradigm, the physical domain of a problem is discretized using a lattice of regular cells that compute and communicate with their neighboring cells in the SIMD model. Essentially, a cellular automaton consists of a set of identical components governed by simple rules acting together to predict complicated behavior of a system. These small components (cells) interact with a group of adjacent cells known as a cell neighborhood. This neighborhood can be any group of cells, but often the classical *von Neumann* and *Moore* neighborhoods are used. The von Neumann neighborhood has five cells — a center and four other cells usually described by the directions S, W, E, and N as shown in Figure 1-(a). The Moore neighborhood shown in Figure 1-(b) is a 3×3 square grid of cells again described by a center cell and the directions SW, S, SE, W, E, NW, N, and NE.

Point-wise quantities associated with each cell specify state information and are updated concurrently (the SIMD model)in discrete time steps using prespecified rules. The state of a cell at iteration t is denoted by $\Phi^{(t)}$ and may consist of numeric, logical, and character entries. In a serial implementation for a regular lattice, cells are updated starting from the lower left corner to upper right corner in a row-wise fashion. During this update process, each cell uses only its own state and state information of the neighboring cells. For instance, for a Moore neighborhood, a Gauss-Seidel (named for the matrix iteration for a discretization of a partial differential equation) update process can be written as follows:

$$\Phi_C^{(t+1)} = \Re[\Phi_{SW}^{(t+1)}, \Phi_S^{(t+1)}, \Phi_{SE}^{(t+1)}, \Phi_W^{(t+1)}, \Phi_C^{(t)}, \Phi_E^{(t)}, \Phi_{NW}^{(t)}, \Phi_N^{(t)}, \Phi_{NE}^{(t)}],$$

where $\Re$ is the update rule. This iterative update process is then terminated when changes in the cell state for all cells in the entire lattice are less than a given small measure ϵ in a relative sense. Such a serial implementation of a CA model is completely equivalent to the classical Gauss-Seidel matrix iteration, with the update rules suitably interpreted as pointwise equilibrium conditions defining a global system of equations ($Ax = b$ or $F(x) = 0$). A CA amounts to an SIMD algorithm for solving those equations.

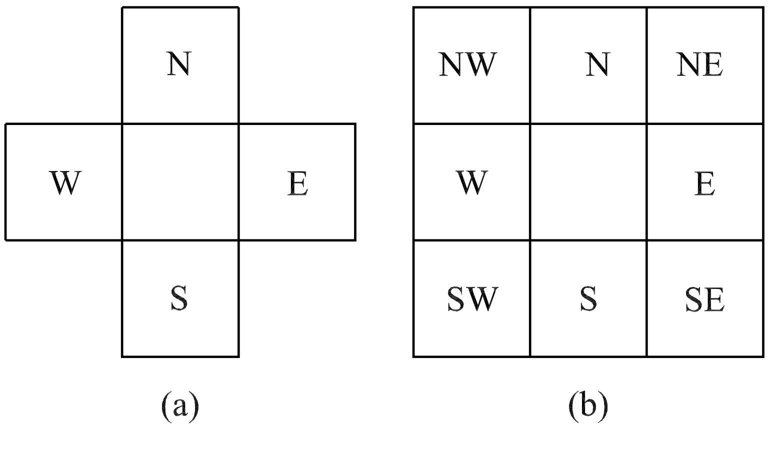

FIG. 1. *CA neighborhoods: (a) von Neumann (b) Moore.*

3 Synchronous Parallel Implementation

As has been stated, CA are inherently parallelizable. The cell domain can be decomposed easily for distribution across a parallel environment since communication patterns are regular across the structure and update rules are based on a local neighborhood. The primary goal of a parallel CA decomposition then is to balance the workload across all available processors while minimizing the communication requirements.

Balancing the workload for a given CA is most commonly done with the assumption that each cell in the domain performs approximately the same amount of work per iteration as every other cell. Equal distribution of cells to available processors then balances the load but the partitioning of the space completely defines the amount of communication required. Cells must communicate state information with all cells in their update neighborhood. For the synchronous implementation in this work, neighboring cells on the same processing node are considered to communicate state information instantaneously, as they share memory space. Those adjacent cells partitioned to separate processors must communicate state information through message passing. The amount of communication traffic is therefore reduced as the number of contiguous cells associated with each processor increases.

The most straightforward application of this balancing information is realized through partitioning the cell space into work units no smaller than a single row of cells. Processing nodes are allocated a roughly equal number of rows of the cell domain given by the following formula:

$$\lfloor (Y/N) \rfloor + \begin{cases} 1, & \text{if } Y \bmod N > n, \\ 0, & \text{otherwise,} \end{cases}$$

where n is the processor rank (0 to $N-1$), N is the number of processors, and Y is the number of rows in the cell data structure. Each processor is given an equal number of rows to work with and the remaining rows are divided equally among low ranking processors (those at the bottom of the domain). Once the number of rows per processor has been determined, the assignment is made in contiguous blocks. An example decomposition of the cell domain into five regions is given in Figure 2, where the shaded rows are shared by adjacent processors.

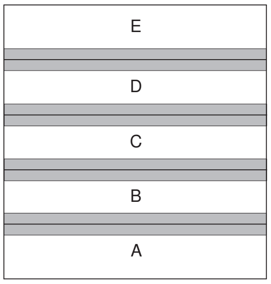

FIG. 2. *Decomposition of the cell domain into five regions. Shaded rows are shared by adjacent processors.*

The initial parallel implementation is modeled after the approach used by Slotta et al. [4]. It is a straightforward application of parallel resources to the cell domain in which each processor iterates independently over the cells in its domain in a Gauss-Seidel iteration style. Processors communicate boundary information with their neighbors (above and below) between iterations in a lock-step communication phase and advance in parallel. As the number of processors used in the calculation increases, the iteration style becomes a hybrid of Gauss-Seidel and Jacobi iterations. Gauss-Seidel updates for the interior of a processor's cell domain and Jacobi updates across communication boundaries cause the rate of convergence for different numbers of processors to vary. The task graph, representing the initiation dependencies between processor dependent iterations, is given in Figure 3.

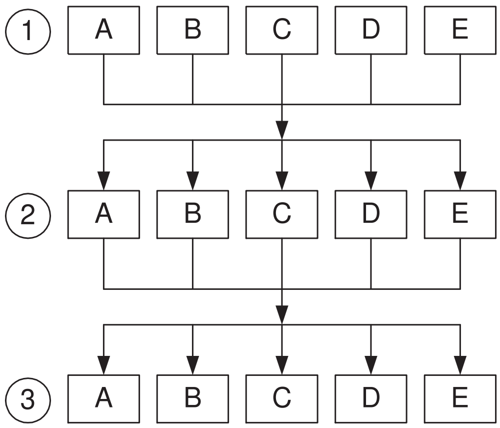

FIG. 3. *Standard parallel implementation. The circles denote computation steps (iteration) between communication. Tasks located on the same horizontal line can be performed in parallel with communication being performed in lock-step after each iteration.*

This implementation, though trivial, has the advantage that each processor must only communicate with two neighboring processors. This eliminates some of the message passing overhead as an entire row of state information can be passed in a single message. The iteration style for the parallel implementation is neither Gauss-Seidel nor Jacobi but rather a mixture of the two. Each processor iterates using new state information where available (Gauss-Seidel) and uses old, previous iteration, state information in a Jacobi style across partition/processor boundaries.

4 An Engineering Application

A structural design problem is considered here to study the performance of the presented CA implementations. Traditional methods of structural design usually require repetitive finite element analyses that are expensive for large scale and nonlinear problems. The inherent structure of cellular automata is trivially parallelizable and can directly benefit from massively parallel machines, whereas nonlinear finite element analysis does not scale easily. Another important feature of the CA application in structural design is the integration of the analysis and design in its numerical implementation. Typically, local cell level rules are implemented for the update of both the field variables and the design variables. Hence, CA methodology can be thought of as an overall simultaneous analysis and design approach, even if the cell level updates are implemented sequentially, thereby eliminating the need for repetitive finite element analyses that are required by the traditional optimization schemes. Moreover, earlier studies [6] showed that the local analysis update rule can be written based on the prior deformed states so that geometric or material nonlinearities can be easily taken into account during the design updates, making the method computationally more efficient compared to a finite element based design optimization method. Finally, it is also envisioned that the local rules from different disciplines may be combined at the cell level easily leading to a multidisciplinary design optimization tool that can be more efficient than trying to integrate multiple models in a global formulation.

The problem of minimum compliance design with a volume constraint is investigated in this study. Compliance of a structure, by definition, is the work done by the external forces. Therefore, to minimize the compliance for a structure with applied loads, the stiffness of the structure should be maximized while at the same time, the total volume must be kept less than a given value. This work considers fiber reinforced composite lamina with curvilinear fiber paths and treats the point-wise fiber orientation angle and density measure as continuous design variables.

The domain of a problem is decomposed into a regular lattice of cells. A *Moore* neighborhood (3×3 array) is used to derive the two level analysis and design update rules. The analysis update rule updates the displacements of each cell given the displacements of the neighboring cells. This rule is governed by the equilibrium of the cell neighborhood and derived by a local finite element model with four quadrilateral bilinear elements. The design update rule updates the cell fiber angle and density based on a local interpretation of the optimality criteria for minimum compliance design of a two-dimensional domain with volume fraction η (ratio of design volume to the volume of a nominal design). Details of the derivation of these update rules are given in [7].

4.1 Numerical Examples

Design of an unsymmetric cantilever plate with geometry and loading as shown in Figure 4 is studied. The magnitude of the downward force F is $10000N$ and the following orthotropic stiffness properties are used in the simulations:

$$E_1 = 135.2\ GPa,\ E_2 = 9.241\ GPa,\ \nu_{12} = 0.318,\ G_{12} = 6.276\ GPa.$$

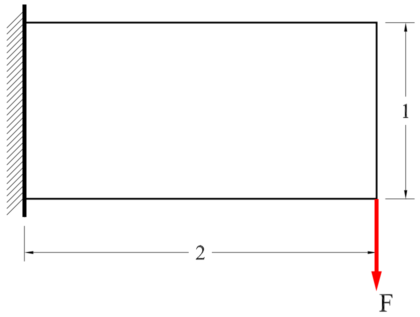

FIG. 4. *Unsymmetric cantilever example.*

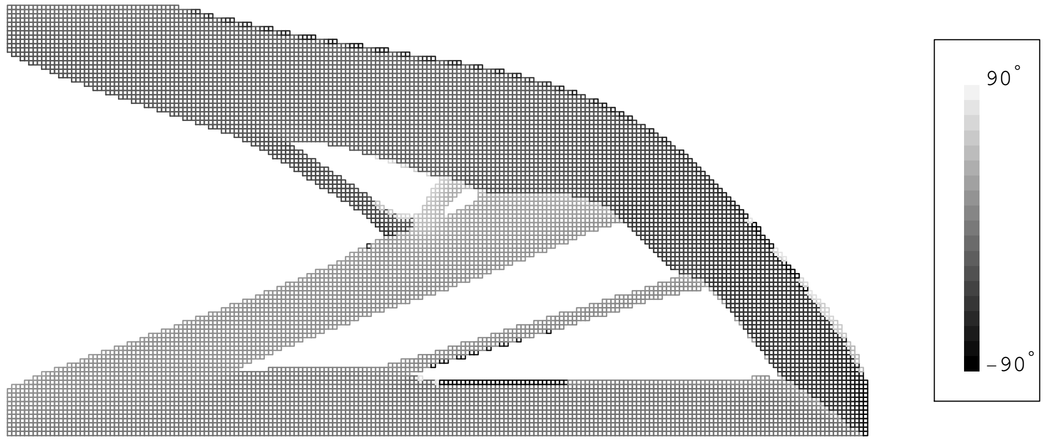

FIG. 5. *CA design for the cantilever problem with* 50% *volume fraction. Cell color represents the optimal fiber angle.*

This problem was modeled by a 201×101 CA lattice and 32 processors were employed to generate the results. Analysis of the structure without design updates converged after $211,938$ iterations and yielded a compliance of $C = 129.30\ N \cdot m$. The optimum topology can be obtained for different volume fractions. As the volume fraction decreases, compliance will increase; therefore, it is helpful to introduce a nondimensional performance index as follows [8]:

$$PI = \frac{C_0 \eta_0}{C_i \eta_i}, \tag{1}$$

where C_0 and $\eta_0 = 1$ are the compliance and volume fraction of the nominal structure, respectively, and C_i and η_i are the counterpart values for an optimal design.

For the problem under consideration, the design for maximum performance index was obtained after $347,764$ CA iterations for $\eta = 0.5$ with a compliance of $C = 62.86 N \cdot m$ as shown in Figure 5 . The performance index for this design is $PI = 4.11$, whereas for the nominal design $PI = 1$.

Attempts to employ more than 32 processors in this case fail due to divergence of the CA iterations. To investigate this issue, consider the following linear system of equations:

$$\mathbf{KX} = \mathbf{F}. \tag{2}$$

The CA analysis in a serial environment is identical to the Gauss-Seidel iterative solution of this system where $\mathbf{K}$ is the stiffness matrix obtained by an equivalent global finite element model, $\mathbf{X}$ is the unknown vector of displacements, and $\mathbf{F}$ is the given vector of applied forces. It is known that the finite element stiffness matrix is symmetric and positive definite for well-restrained structures [9], therefore convergence of a Gauss-Seidel iteration is guaranteed regardless of the starting point [10]. However, a Jacobi iteration may or may not converge. To illustrate this issue, consider a prototype problem as shown in Figure 6 modeled by 6 nodes and two bilinear finite elements. The plate is assumed to have constant thickness of $0.001m$ with the same geometry and loading as in Figure 4 .

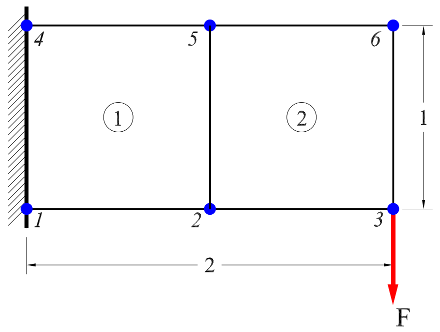

FIG. 6 *The prototype problem.*

The global stiffness matrix of Eq. (2) can be written as the summation of the diagonal, lower triangular, and upper triangular matrices as:

$$\mathbf{K} = \mathbf{L} + \mathbf{D} + \mathbf{U}. \tag{3}$$

The Jacobi and Gauss-Seidel iteration matrices are given by, respectively,

$$\mathbf{J} = -\mathbf{D}^{-\mathbf{1}}(\mathbf{L} + \mathbf{U}), \tag{4}$$

$$\mathbf{G} = -(\mathbf{L} + \mathbf{D})^{-\mathbf{1}}\mathbf{U}. \tag{5}$$

By definition, the *spectral radius* is the maximum absolute value of the eigenvalues of the iteration matrix. For the prototype problem, the numerical values of the spectral radii for the Jacobi and Gauss-Seidel iteration matrices are

$$\lambda_J = 1.46596,$$

$$\lambda_{GS} = 0.834054.$$

The fact that the spectral radius for the Jacobi iteration matrix is larger than one implies that Jacobi iteration for Eq. (2) will not in general converge, and thus neither may the equivalent Jacobi CA iterations (the pure parallel CA [SIMD] implementation). On the other hand, Gauss-Seidel iteration always converges since the spectral radius in that case is less than one.

However, the present synchronous parallel implementation is a mixture of Jacobi and Gauss-Seidel iteration modes, where it is Gauss-Seidel within each processor, and Jacobi between processors. Hence, the iteration is more Jacobi like as the number of processors increases. For a single processor, the update environment is pure Gauss-Seidel, and when there are as many processors as the number of cells, it is a pure Jacobi update scheme. Anywhere between those two extremes, the iteration is a mixture of Jacobi and Gauss-Seidel. Numerical experiments showed divergence for the parallel code as the number of processors exceeded a certain value. The above spectral radii calculations for the prototype problem explain this divergence.

5 Pipeline Parallel Implementation

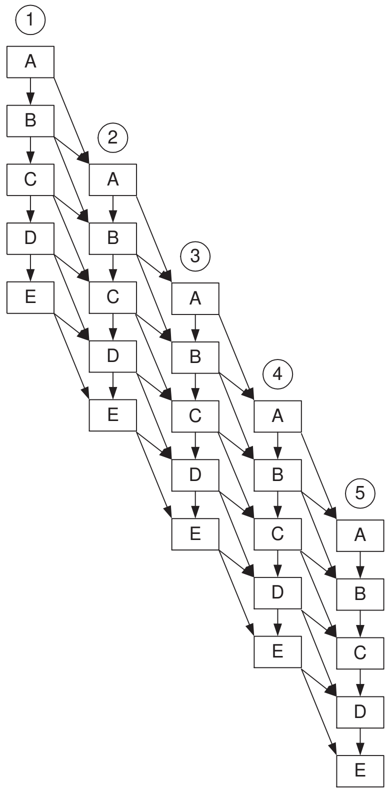

FIG. 7. *Trivial pipe, equivalent to global Gauss-Seidel iteration. Note the oscillation between two and three active processors at successive computation steps (rows).*

The initial parallel implementation given in Section 3 resulted in divergence of the CA as outlined above through hybrid Jacobi Gauss-Seidel update schemes. An additional parallel implementation was developed to preserve the convergence of a pure Gauss-Seidel iteration through pipelined updates. This pipelined parallel code was created in Fortran 90 using the MPI (Message Passing Interface) communication library to guarantee Gauss-Seidel style iteration for the entire cell domain regardless of the number of processors used in the calculation. Cells are distributed, as above, in horizontal contiguous rows but the global iteration of the cell domain is pipelined across the processors. Using the same task model as the above (synchronous) parallel implementation, where a single iteration of the cell domain given to a processor is atomic, results in an efficiency crippling pipelined implementation of even/odd processor oscillation as depicted in Figure 7. The oscillations are a direct result of coupling the send and receive operations that occur smoothly between iterations for the previous (not pipelined) parallel implementation. This coupling represents the requirement for a processor to receive data from both above and below (cf. Fig. 2) neighbors before the computation for a particular iteration begins. Logically, however, this coupling is only required in the most trivial case where a processor is only responsible for updating a single row of the entire cell domain. In all other cases the task of a single iteration on a processor can be divided into three components. The bottom-most row, which requires information from the neighboring processor below before computation can begin, zero or more interior rows, which can be updated once the initial (bottom) row is finished, and a final top-most row that requires data from the above neighbor from the previous iteration before updating can proceed. By dividing the update task per processor into these three categories the pipeline efficiency can be significantly improved through the use of non-blocking send operations. The resulting task graph is shown in Figure 8.

Though the asynchronous pipeline code emulates a serial Gauss-Seidel iteration precisely, the state of the system at a given timestep is in flux and can not be globally queried. Instead, state information for the system is passed along the pipe and aggregated by the last processor in the chain to determine global state. In this way, the convergence criterion for the system is determined at a local level and then collected through the pipe by the last processor to direct state based actions like termination. Each processor uses a greedy approach to possible work that dictates updates for the local domain of cells as soon as the required information is available. A processor need only block once for information from the below neighbor, before starting an iteration, and a second time before updating the top-most row of the local domain, requiring information from the above neighbor from the previous iteration. All sending communications are nonblocking while the corresponding receives are necessarily blocking. This allows maximal time for message transit through the policy of sending information as soon as it is ready and postponing receives until they are required for further computation.

Since state data records for past iterations are not stored in memory for each processor, the final state of the system is a tiered state snapshot of the pipe at termination. The greedy approach, to compute until new information is required, allows the final state of the pipe to partially update a local domain (all but the top-most row) and then terminate, sending that state to output. Partial updates and iteration updates beyond the iteration where convergence is detected are reasoned to further tighten the termination criteria and the need for recalled state information on a per processor basis is discounted accordingly.

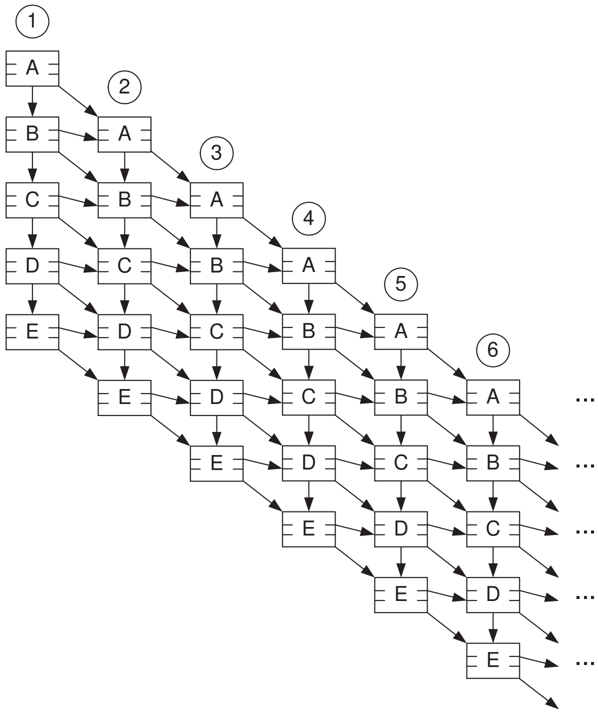

FIG. 8. *3-component task pipe, an asynchronous Gauss-Seidel iteration. Note that after the pipe fills, five processors are active at each component computation step. The tick marks (from top to bottom) within a processor task square represent the completion of the bottom row and interior rows respectively.*

The primary advantage of the pipeline code is that it preserves the convergence properties of a serial Gauss-Seidel iteration, which in general is more likely to converge than a corresponding Jacobi iteration. Convergence results and speedup comparisons are given in the next section.

6 Results and Conclusions

To investigate convergence properties of the proposed pipeline implementation, the same structural problem of Section 4.1 is considered here. The cantilever plate of Figure 4 with symmetric loading is modeled with a 23×12 CA lattice. A serial Gauss-Seidel code converges for combined analysis and design after $12,897$ iterations for a 77% volume fraction design predicting a compliance of 40.63 $N \cdot m$. The parallel synchronous implementation, as described in Section 3, also converges for up to 8 processors to a design similar to that of the serial results in a slightly increased number of iterations. However, as predicted in Section 4.1, the synchronous CA iterations fail to converge when more than 8 processors are employed. The pipelined implementation, which simulates a pure Gauss-Seidel iteration regardless of the number of processors, always converges to the same results as the serial Gauss-Seidel implementation.

Execution time for the asynchronous pipeline code was measured for a cell domain grid of 201×101 cells. The start up time and time required to flush the pipe at the end of the calculation grow as the number of processors increases. Since the pipeline code emulates a serial Gauss-Seidel iteration, the number of iterations to convergence are the same regardless of the number of processors used in the calculation. Timing results giving the speedup for the asynchronous pipeline are given in Figure 9 with computation time till convergence measured in seconds. Performance of the algorithm is expected to drop severely as the number of processors used in the calculation increase beyond half the number of available rows. In the case of the 201×101 grid, using more than 51 processors causes at least one processor in the pipe to have a single row to update and will result in the oscillation behavior depicted in Figure 7. This behavior, even for the simple row partition of the domain, can be avoided until the 100 processor mark simply by rotating the model by 90 degrees. *Additional data will be added to Figure 9 as compute time becomes available.*

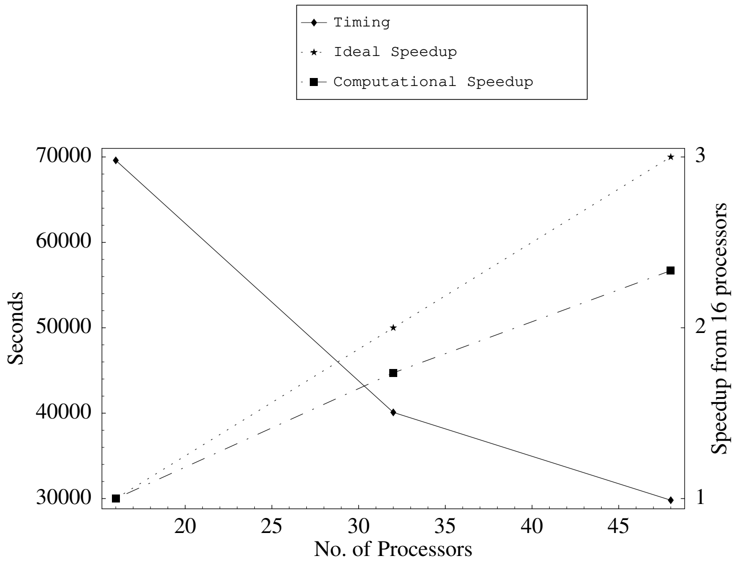

FIG. 9. *Timings performed on the P3 partition of Tempest at the Maui High Performance Computing Center consisting of 16-way, 375Mhz Nighthawk-2 nodes, each with 8Gbytes of shared memory.*

A convergence based speedup comparison for the synchronous CA code holds no meaning since the number of iterations to convergence, and thus the amount of computation performed, changes with the number of processors used. For the 201×101 grid the synchronous CA fails to converge using 64 processors. The justification for a pipelined version of the CA for this application is that the pipelined CA can not fail strictly based on the number of processors used for the computation.

References

[1] J. R. Wiener, A. Rosenbluth, *The mathematical formulation of the problem of conduction of impulses in a network of connected excitable elements, specifically in cardiac muscle*, Arch. Inst. Cardiol. Mexico, 16 (1946) pp. 205–265.

[2] K. Preston, M. J. B. Duff, *Modern Cellular Automata, Theory and Applications*, Plenum Press, New York, 1984.

[3] B. B. Lowekamp, L. T. Watson, M. S. Cramer, *The cellular automata paradigm for the parallel solution of heat transfer problems*, Parallel Algorithms and Applications, 9 (1995) pp. 119–130.

[4] D. J. Slotta, B. Tatting, L. T. Watson, Z. Gürdal, S. Missoum, *Convergence analysis for cellular automata applied to truss design*, Engineering Computations, 19 (2002) pp. 953–969.

[5] J. Wahle, L. Neubert, J. Esser, M. Schreckenberg, *A cellular automaton traffic flow model for online simulation of traffic*, Parallel Computing, 27 (2001) pp. 719–735.

[6] Z. Gürdal, B. Tatting, *Cellular automata for design of truss structures with linear and nonlinear response*, Proceedings of the 41st AIAA/ASME/ASCE/AHS Structures, Structural Mechanics and Materials Conference, AIAA Paper 2000-1580, Atlanta, GA, 2000.

[7] S. Setoodeh, Z. Gürdal, M. Abadalla, *Combined topology and curvilinear fiber path design of composite layers by cellular automata*, to appear in Proceedings of the 45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics & Materials Conference, Palm Springs, California, April 2004.

[8] Q. Q. Liang, G. P. Steven, *A performance-based optimization method for topology design of continuum structures with mean compliance constraints*, Comput. Methods Appl. Mech. Engrg., 191 (2002) pp. 1471–1489.

[9] O. C. Zienkiewicz, R. L. Taylor, *The Finite Element Method*, Butterworth-Heinemann, 2000.

[10] D. Kincaid, W. Cheney, *Numerical Analysis*, Brooks/Cole, Pacific Grove, CA, 1996.