11th International Conference on Vibration Problems
Z. Dimitrovová *et al.* (eds.)
Lisbon, Portugal, 9-12 September 2013

HIGH PERFORMANCE COMPUTING OF ELASTODYNAMIC RESPONSE ON COMPLEX DOMAINS USING CELLULAR AUTOMATA

Michael J. Leamy*[1], Wesley Emeneker[2]

[1]GWW School of Mechanical Engineering, Georgia Institute of Technology, USA
michael.leamy@me.gatech.edu

[2]Office of Information Technology, Georgia Institute of Technology, USA
wesley.emeneker@oit.gatech.edu

Keywords: Cellular Automata, Wave Propagation, Computational Mechanics, High Performance Computing, Message Passing Interface.

Abstract. *This article presents a High Performance Computing (HPC) implementation of a* Java*-based elastodynamic Cellular Automata simulator. Elastodynamic simulation using Cellular Automata (CA) has recently been presented as an alternative, inherently object-oriented technique for accurately and efficiently computing linear and nonlinear wave propagation in arbitrarily-shaped geometries. The local, autonomous nature of the method makes it ideal for straight-forward and efficient parallelization. As such, using a* Java*-based implementation of the Message Passing Interface (MPI), the existing CA simulator has been rewritten to run on multicore desktop machines and multicore, multinode computer clusters. This is accomplished by optimally assigning each processor a subdomain of cells, together with their neighbours, with the objective of minimizing interprocessor send-receive data. On a single multicore machine the communication costs are very low (as expected) and thus near-linear speedup is documented. On multiple node clusters, scaling studies demonstrate that as the number of cells in the domain increases, so does the speedup advantage. For a domain with nearly one million cells, 128 processors correctly compute the elastodynamic response over 75 times faster than a single processor. This demonstrates both the effectiveness of the presented HPC implementation approach and the potential for* Java*-based HPC.*

1 INTRODUCTION

The cellular automata (CA) paradigm [1, 2] has recently been adopted to simulate wave propagation in two-dimensional linear and nonlinear elastic domains of arbitrary shape [3, 4]. CA models are distinguished by their use of simple, local interaction rules to compute complex global behavior – this global behavior is often termed 'emergent.' The rules are derived from known relationships (such as geometrical constraints, conservation of energy, etc.) and statistical relationships/models, as well as other intuitive relationships. The elastodynamic CA approach shares an idea central to all cellular automata modeling, which is domain discretization using autonomous cells (usually rectangular or hexagonal) whose state is updated via simple rules. Each cell is treated as an autonomous state machine storing pointers to its local neighbors' state, avoiding the need for global, or centralized, control. This lends itself well to object-oriented (OO) programming practices implemented using modern computing languages (*C++*, *Java*, etc.) and is compatible with parallel processing. The elastodynamic CA approach is discontinuous like the finite difference time domain (FDTD) method, and thus accurately captures propagating wave fronts. In fact, for a uniform rectangular grid, the method reproduces the same interior update equations as the central difference method [4]; however, it differs from FDTD in that easily generalizes to cells of varying shape, much like the finite element (FE) or finite volume (FV) methods.

Non-uniform triangular cells allow domains of multiply-connected, arbitrary shape to be simulated efficiently and accurately. In fact, the method has been shown to effectively avoid spurious oscillations at the front of sharp wave fronts without the need for specialized treatment (unlike other methods – e.g., the finite element method). The triangular case serves as the starting point for a High Performance Computing (HPC) implementation of a *Java*-based elastodynamic simulator described herein. The paper begins by reviewing the CA elastodynamic formulation and then discusses the parallelization approach using the Message Passing Interface (MPI) [5]. Speedup results are presented for both shared memory (e.g., multicore machines) and distributed memory (e.g., cluster) systems, followed by concluding remarks.

2 ELASTODYNAMIC CA OVERVIEW

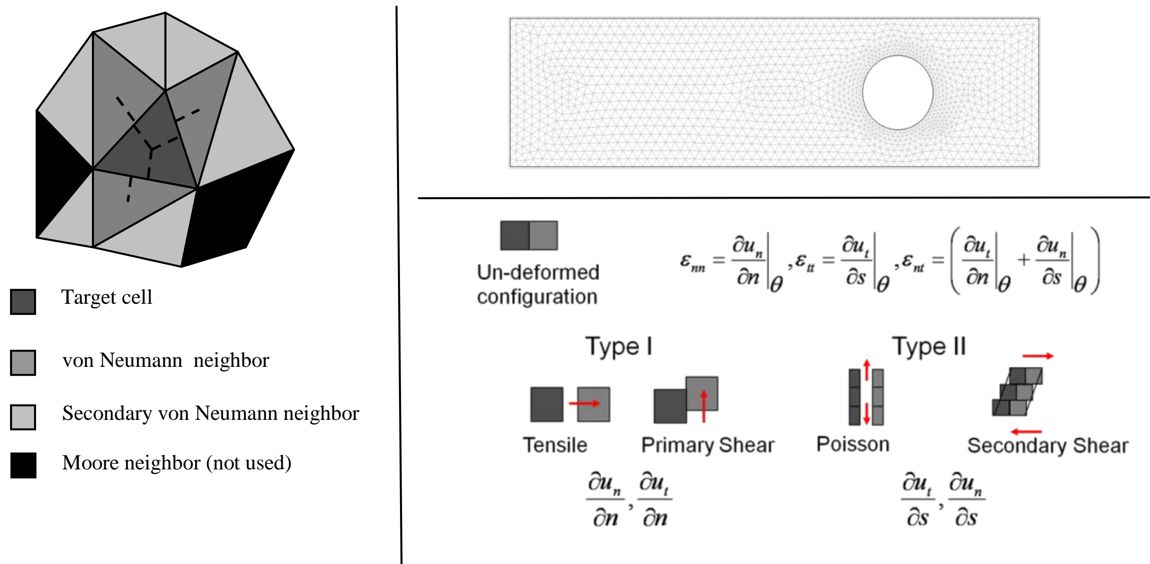

Figure 1: (Left) target cell geometry with neighbors identified. (Top Right) multiply-connected domain with automata mesh. (Bottom Right) rectangular automata illustrating strains needed for rule set development.

Traditional computational approaches for structural dynamics can be considered top-down (or reductionist) as they start with a general law or principle, usually in partial differential equation (PDE) form, and work downwards to determine the behavior of subsequent discrete entities. Cellular automata, on the other hand, starts at the bottom with discrete entities (cells) holding individual states governed by a small set of rules, without the need to introduce further a continuum, a PDE, and subsequent discretization. The global behavior emerges from the assembly. Figure 1 provides a graphical overview of the elastodynamic CA method. Non-uniform triangles are employed to discretize a domain into multiple state machines termed automata. The rule set governing the temporal update of each cell is first arrived at using a balance of momentum applied to a target cell which sums forces present on each face. In doing so, the computation of strains is necessary, which is done by categorizing the strains as either Type I (derivatives in the normal direction) or Type II (tangential derivatives). Numerical evaluation of the spatial derivatives then follows from simple finite difference expressions using the appropriate states of neighboring automata. By choice, we use only von Neumann and what we term secondary von Neumann neighbors. Finally, a forward-Euler time integration of the momentum equation yields the target cell's explicit rule set. Simulation proceeds by requesting that each automata update their state (displacement and velocity components) at each time step. Note that the method relies solely on local interactions, avoids partial differential equations and their complexity, and is fully object-oriented. In fact, the traditional process of assembling and solving a matrix set of equations is traded for requests to the automata objects to update their state.

State variables stored by each triangular automata include the displacement, velocity, and applied tractions. In addition, cells store parameters such as material properties and cell geometry (centroid, face angles and lengths, area). All state information and applicable parameters reference a global x-y coordinate system. The displacements and applied tractions are determined at each step of the simulation and updated based on a local rule set (to be detailed) using the previous state of each automaton and that of its von Neumann and Moore neighbors. This simple, local nature makes the approach massively parallel, and is responsible for the method's high degree of scalability. As illustrated in Figure 1, von Neumann neighbors share an edge with the cell of interest, while Moore neighbors share a vertex. It is important to note that not all Moore neighbors are used in the state update - only those that are also von Neumann neighbors of the cell's von Neumann neighbor. These will be referred to herein as 'secondary' von Neumann neighbors.

The cells are treated as autonomous entities, and so it suffices to describe the update of any given cell, which is termed herein the 'target' cell. At the start of each time step, the previous state is transformed into tangential and normal components for each of the target cell's three faces. Strains are obtained by numerically estimating derivatives of displacements across each face by using the target cell and its local neighbors. These derivatives are classified as either Type I (across the face) or Type II (parallel to the face). Type I models tensile strain and 'direct shear,' whereas Type II models Poisson effects and 'indirect shear.' Note that only von Neumann neighbors are required to evaluate Type I derivatives, while Type II require inclusion of Moore neighbors which have been pre-identified as secondary von Neumann. To facilitate strain computations, an angle θ is identified which measures the angle between the x-axis and the face normal ($-\pi \leq \theta \geq \pi$). Type I derivatives are approximated as,

$$\left.\frac{\partial u_t}{\partial n}\right|_\theta \Rightarrow \frac{u_t^N - u_t^T}{\Delta n}; \text{ similarly, } \left.\frac{\partial u_n}{\partial n}\right|_\theta \Rightarrow \frac{u_n^N - u_n^T}{\Delta n}, \tag{1}$$

where superscripts identify the location of a cell (T: Target cell, N: cell sharing the target face, N^+: cell sharing the target cell's positive-theta face, N^-: cell sharing the target cell's negative-theta face, M^+: secondary von Neumann cell sharing the target cell's positive-theta vertex, M^-: secondary von Neumann cell sharing the target cell's negative-theta vertex) while subscripts denote a direction (n: normal, t: tangential, +: positive theta, -: negative theta). Type II derivatives are determined by calculations spanning differences across both the target cell and the primary and secondary von Neumann neighbors (see Figure 1 for a physical rationale). These calculations are then averaged to find the difference in the tangential direction at the face. The Type II derivatives are therefore approximated as (e.g.),

$$\left.\frac{\partial u_t}{\partial s}\right|_\theta \Rightarrow \frac{1}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_t^{M+} - u_t^{M-}}{\Delta s_+^N + \Delta s_-^N}\right). \tag{2}$$

The Type I and Type II derivatives can be substituted into the strain relationships and then Hooke's Law (or other constitutive relationship) to yield discrete stresses σ_{nn}, σ_{tt}, and σ_{nt} at each face. The stresses and external body loading are used in the balance of momentum applied to the entire target cell (i.e., three faces), followed by a forward-Euler approximation to all time derivatives. This yields discrete, local velocity and position update equations governing the state update. Nuemann and Dirichlet boundary conditions have been treated in full - details can be found in [3, 4].

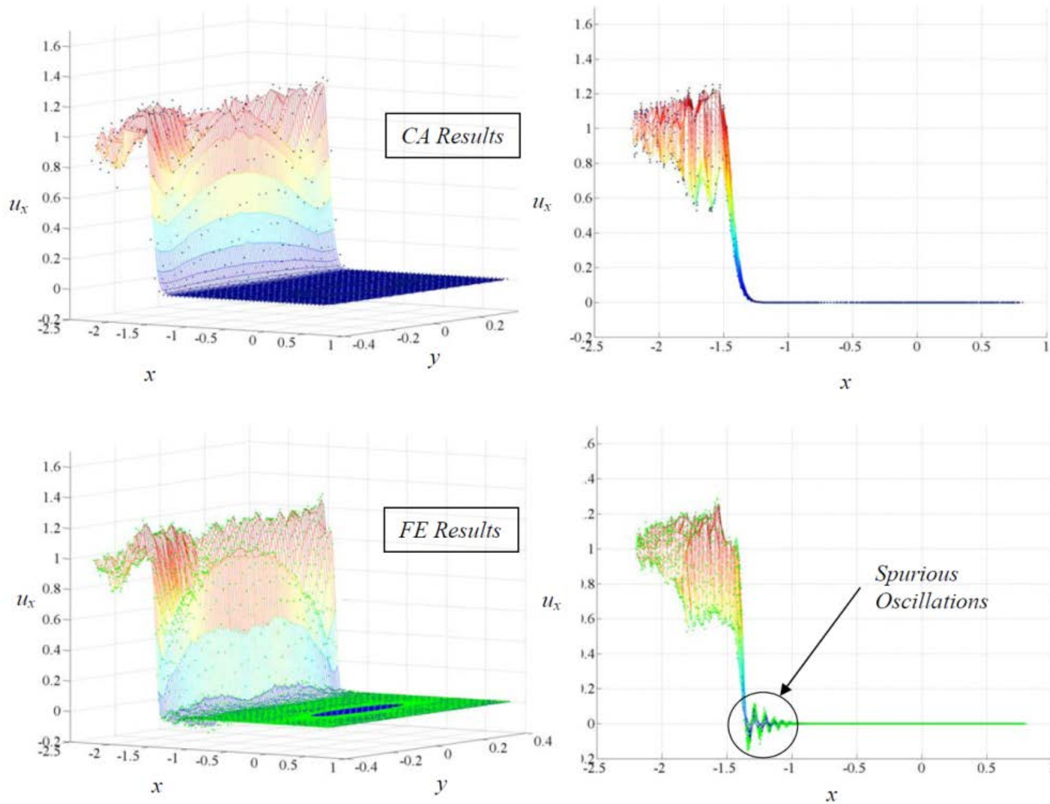

Figure 2: Comparison of x-displacement results at a snapshot in time (finite element [bottom] vs. cellular automata [top]) for a sharply loaded domain with an interior hole – see Figure 1 for the meshed domain. The left subfigures employ an isoperimetric perspective, while the right subfigures employ a perspective from the y-axis. The material simulated is aluminum, and the loading occurs on the left boundary in the form of an imposed Dirichlet boundary condition at the start of the simulation. Figure reproduced from [3].

Simulation results from the elastodynamic CA approach have been compared to those from other methods, including commercially-available finite element simulators, and excellent agreement has been documented [3, 4]. For smooth loading time-histories, the CA and finite element methods yield nearly identical results. However, for sharp loading, the two approaches exhibit qualitative and quantitative differences. Figure 3 documents the x-component of displacement at a snapshot in time when an aluminum two-dimensional domain responds to a sudden, uniform displacement of its left edge starting at time zero. This challenging case results in the propagation of sharp, broad-band wave-fronts (pressure, shear, etc.) primarily in the x-direction. Both methods exhibit broad frequency content, as expected, with displacement profiles exhibiting similar behavior envelopes. However, significant differences are present. In particular, the FE results exhibit spurious oscillations (also termed Gibb's phenomena) in front of the wave-front. These oscillations are well-known to occur in continuous-Galerkin finite element approaches [6]. It is important to note that they are not an artifact of the time-stepping algorithm employed by the commercial code, but rather a result of the spatial discretization employed - i.e., exact solution of the semi-discrete FE equations will exhibit the same Gibb's phenomena. The CA simulation results, in comparison, do not exhibit these spurious oscillations. Instead, the solution more-faithfully captures the sharp, nearly-discontinuous wave-front. This positive feature of the CA method makes it attractive for studying the long-term behavior of propagating elastic waves, particularly in scattering problems and nonlinear media where mode and frequency conversion can be erroneously identified if spurious oscillations are present.

3 PARALLELIZATION APPROACH

The Message Passing Interface (MPI) was chosen for parallelizing the CA-based elastodynamic simulator on both symmetric multiprocessor (SMP) shared-memory and distributed-memory clusters. MPI is the *de facto* standard for passing messages (i.e., data), which is the central enabler of distributed computing. While *C* and *C++* implementations of MPI are common, *Java* implementations are not; furthermore, the *Java* language has received little notice as a serious High Performance Computing (HPC) language. To justify using *Java*, a preliminary assessment of its HPC potential is carried-out with the example of computing Pi using a Monte Carlo technique. Attention then turns to parallelizing the elastodynamic CA simulator.

3.1 Assessment of *Java* HPC

The original elastodynamic CA simulator was written in the *Java* programming language without consideration for the possibility of later parallelization. While *Java* is commonly employed as a computational language, its use in HPC applications is sparse. However, it was decided not to port the CA simulator to another language, such as *C++*, for two main reasons. The first is that *Java* offers a combination of attributes taken together that are not presently offered by other HPC languages: a high degree of platform portability due to its interpretative nature; large and growing user-base promoted at universities and by the software industry; higher programming abstractions including OO features; robust compile and runtime error checking leading to fast development times; automatic garbage collection; support for multithreading (which is exploited by MPJ Express [7] on SMP architectures); and availability of a rich collection of support libraries. Moreover, the performance of the latest version of *Java* is comparable to the performance achieved by programs written in *C/C++* [8]. The second reason for remaining with *Java* is the growing availability and increasing performance [9, 10] of *Java* implementations of the MPI standard, to include the afore-

mentioned MPJ Express and FastMPJ [11]. The present work employs MPJ Express as it has been in existence and proved to be more stable than FastMPJ.

A Monte Carlo simulation of Pi was first written in both *Java* and *C* to compare the runtime performance of both *C*-based MPI and MPJ Express. The approach employs randomly-generated numbers on the interval [0,1] for both *x* and *y* coordinates, and then tallies a one if the pair lies inside of the top right quadrant of the unit circle, and a zero otherwise. Note that the probability of being inside the circle is ¼ of Pi. After *n* iterations, the estimate for Pi is therefore four times the total tally divided by *n*. By performing 'inner' iterations in which each processor generates *n* random (*x*,*y*) pairs, and *m* 'outer' iterations in which each processor reports back to a master processor their Pi estimate after performing an inner iteration, the ratio of time computing Pi to the time passing messages can be controlled. Thus the comparison captures both the inherit calculation speeds of C and *Java*, and the message passing overheads (latency and bandwidth).

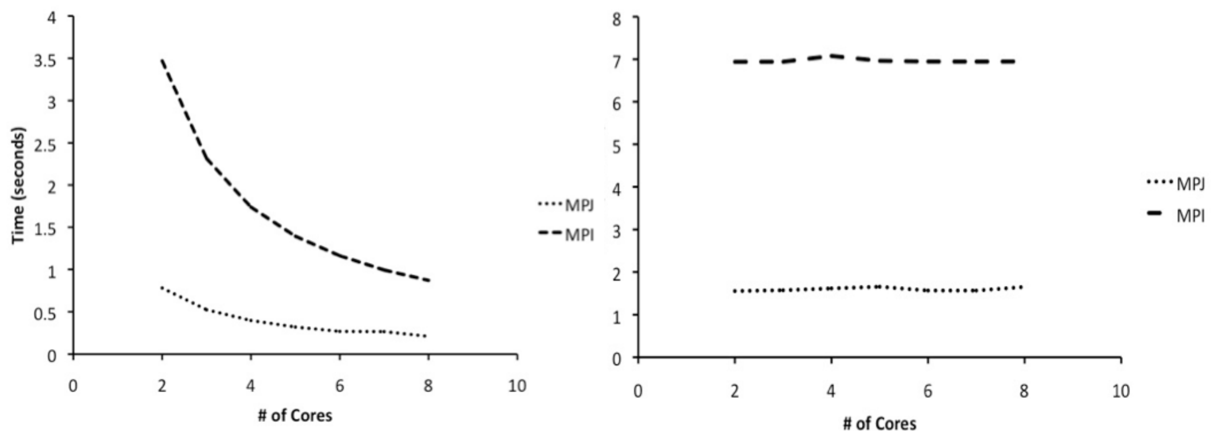

Figure 3: Strong and weak scaling, *C* vs. *Java*, using a parallel Monte Carlo algorithm for computing Pi.

The Monte Carlo estimates of Pi were run on a 64-bit Windows 7 machine with two Intel Xeon E5506 quad-core processors and 24GB of available RAM. The *C* simulation employed the MPICH2 implementation of MPI as well as Cygwin, a Unix-like environment and command-line interface for Windows [12]. The *Java* version utilizes MPJ Express and runs natively through the NetBeans IDE. The coded algorithms are nearly identical in terms of the number of lines of code needed, variable types employed, and overall program flow. Figure 3 illustrates scaling results from the simulation. The graph on left shows the strong scaling comparison (fixed problem size) using 10 million iterations. Weak scaling is also provided and is used to illustrate the manner in which the solution time varies with the number of cores employed (10 million iterations per core). The results indicate that, as implemented, *Java* performs on the same order, or better, than *C*. This conclusion has also been reached in studies conducted by the MPJ Express developers [13].

3.2 HPC Implementation of Elastodynamic CA

The elastodynamic CA calculation flow is straight-forward in the absence of parallel computation: at each time step, the main routine requests that each cell in the domain update its state. As described in Section 2, this requires each cell to gather the previous time step state information of neighboring cells, and to then calculate strains, stresses, and then ultimately changes in its velocity and displacement. When parallelization is employed, this calculation flow is unchanged except that at the end of each time step, cells on one processor with point-

ers to cells on another processor must exchange state information. The calculation flow described is captured in Figure 4. The figure assumes the presence of *NP* number of processors residing on an SMP or cluster architecture. The Master processor is given the identification (ID) 0, while the other processors are given IDs 1 through *NP-1*. In order to reduce memory usage, only the Master processor reads the input file, which contains geometry, mesh topology, material properties, loading, and boundary conditions. The Master processor instantiates each simulation cell and stores them in a single array termed the *cellList*. This array is reordered (described below) to minimize the amount of state information later passed between processors. After reordering, the Master processor divides the *cellList* into *NP*-number of equal-sized partitions and uses MPI send/receive calls (also described later) to send subordinate processors the cells it is responsible for computing state updates, and any neighbor cells it will need to accomplish the state updates. Finally, the Master processor has the task of establishing a series of neighbor sharing arrays that keep track of what cell information each processor must share with other processors. It does this by interrogating the neighbor pointers held by each cell, and ensuring that no duplication of information leads to extraneous message passing. For example, it is probable that more than one cell on processor *m* shares the same neighbor (von Neumann or Moore) on processor *n*, and thus this neighbor's state information need only be sent once to processor *m*. A series of arrays encodes this information for each state variable shared (displacement and velocity), to what cell and what face it belongs, and the cell IDs involved in the sharing and distributing of this information. These sharing arrays are then used to decode tightly packaged information broadcast between the processors.

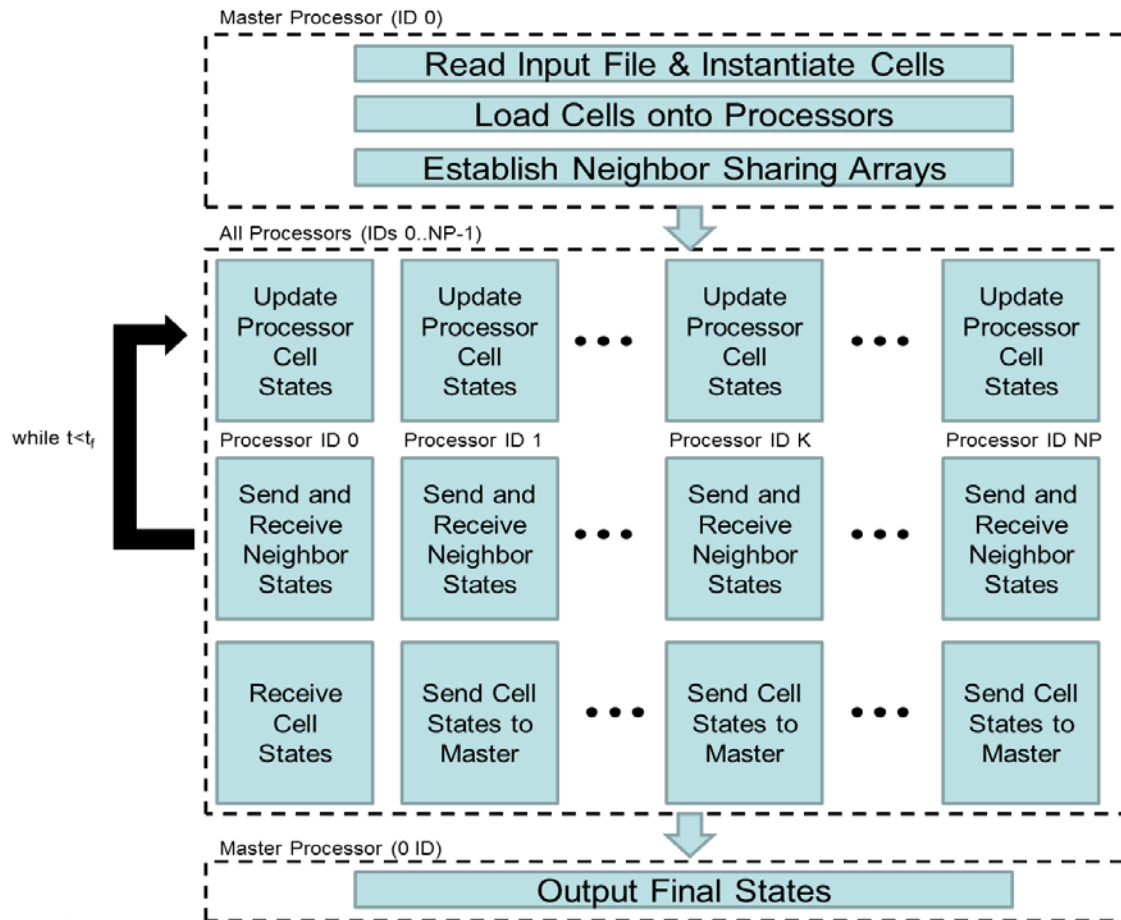

Figure 4: Parallel calculation flow for the elastodynamic CA simulator.

Following the setup of the simulation, all processors (including the Master) proceed in a time-stepping loop (see Figure 5) where they update the state of their assigned cells, send and receive neighbor state information, and then proceed to perform another update until the end time is reached. This process continues until the simulation time has been exceeded, at which point all of the processors send their state data to the Master processor so that results can be outputted to data files. Note that the send-receive operations required to share neighbor states are 'blocking' – i.e., the processor waits for an expected message to be received before proceeding to follow-on calculations. Thus it is critical to 'load balance' the processors such that one is not computing significantly more than any other processor, and thus dictating/limiting the speed of the solution. The load balancing requirement is met nicely in this work by the trivial task of ensuring the *cellList* is split as equally as possible to each processor, but also by ensuring one processor does not have an excessive communication burden compared to another processor, as described next.

```
while (stepNumber <= endStep) {
   me = MPI.COMM_WORLD.Rank();

   // Update state of each cell accountable to 'me' processor
   cellCount = -1;
   for (int i = beginVector[me]; I <= endVector[me]; i++) {
       cellCount++;
            cellList.get(cellCount).step(stepsPerUnitTime);
   }

   // Share neighbor states with other processors
   shareNeighborStatesAcrossProcessors();
   stepNumber++;
}
```

Figure 5: Calculation flow pseudo-code. Note that the ID of the processor running the code in an MPI implementation can always be determined with a rank request, as indicated in the pseudo-code. The global arrays *beginVector* and *endVector* stores each processor's starting and ending cell location in the *cellList*. The call to the `shareNeighborStatesAcrossProcessors()` method is described in more detail in Figure 7.

Communications are often the bottleneck in a parallelization strategy due to their non-negligible latency. In order to decrease the number of communications per processor, we populate cells on a processor by exploiting the neighbor information already stored by the CA cells in such a way that when a cell is added to a processor, its neighbor cells are next added to the processor. This is illustrated graphically in Figure 5. Starting with the first cell stored in the *cellList*, a cell is assigned to the first processor while its neighbors are placed in a first-in, first-out queue awaiting assignment to the same processor. Each cell in this queue is removed, assigned to the processor, and its neighbors (if not already in the queue) are added to the queue. The process is repeated for the first processor until its allotted cells have been assigned, at which point the assignment and queuing continues for the next processor. The described straight-forward and load-balanced CA parallelization approach can be contrasted with traditional elastodynamic simulation methods, such as most finite element approaches, which require a sparse linear system solver and complex decomposition schemes prior to HPC deployment [14].

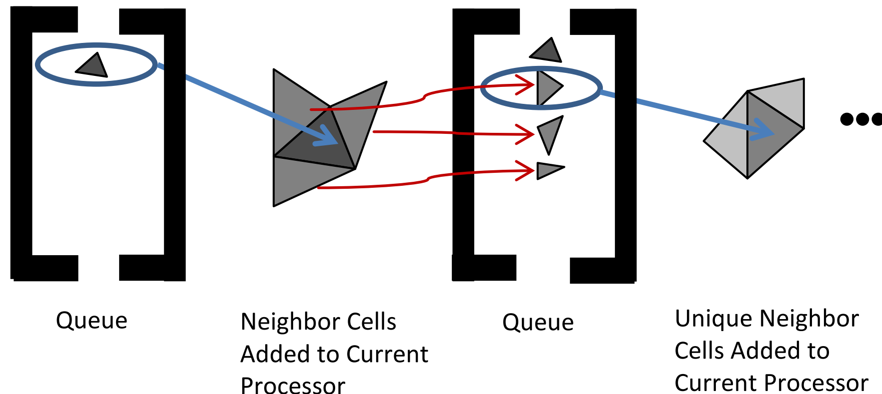

Figure 6: Depiction of processor loading for minimizing interprocessor communications.

Efficiently scheduling of send/receive pairs for passing neighbor information is a critical task for achieving good speedup, especially when employing many processors. The send/receive pairs must also avoid hanging processes where one processor expects a blocking message from a second processor, who is also awaiting a blocking message. Figure 7 provides pseudo-code implemented to share neighbor information and thus parallelize the elasto-dynamic CA simulator. After completing state updates, each processor executes a loop sending/receiving with all other processors. If the rank of the current processor equals the loop iterator, it expects to receive neighbor information, otherwise it sends information. When sending information to another processor, all neighbor displacements and velocities are stored in a single array holding doubles-type data termed *sendDoubleBuffer*, which when decoded using the neighbor sharing arrays, limits the communication overhead to only a single send/receive pair between each processor per time step. The result is minimal communications and hence strong performance. Note that each send/receive MPI call requires the following arguments: pointer to a communication buffer, a buffer offset (in all cases zero), the size of the buffer, the type of data stored in the buffer, the intended target or source, and an integer ID for tagging the message. Note further that MPJ Express allows entire objects (i.e., cells) to be passed as opposed to primitive data types, which if employed would greatly decrease the complexity of the neighbor sharing arrays, but was found to be prohibitively slow and error-prone when passing cells with pointers to other cells.

```
for (int i = 0; i < NP; i++) { // i ranges over all processors
   if (me == i) { // receive from all other processors
      for (int j = 0; j < NP && me!=j; j++) {
         MPI.COMM_WORLD.Recv(recvDoubleBuffer,0,bufferSize,
            MPI.DOUBLE,j,j*NP + me);
         updateNeighbors(me,j,recvDoubleBuffer);
      }
   }
   else { // send appropriate neighbor states to processor i
     sendDoubleBuffer = getNeighborStates(me,i);
     MPI.COMM_WORLD.Send(sendDoubleBuffer, 0, bufferSize,
        MPI.DOUBLE,i,me*NP + i);
   }
}
```

Figure 7: Neighbor sharing pseudo-code. The call to the method *updateNeighbors* uses the neighbor sharing arrays to update the requisite neighbor states.

4 SPEEDUP RESULTS

This section assesses the performance of the parallelization strategy presented in Section 3 on both SMP and distributed-memory clusters. In the case of a single multicore machine, MPJ Express automatically limits communication costs by employing *Java*'s innate threading capability, thereby avoiding the need for interprocessor communications. As a result, the speedup performance can be anticipated to be very good. On distributed memory clusters, MPJ Express employs the full communication protocols and performance can be anticipated to suffer in comparison.

4.1 Shared memory multicore systems

Results are presented in Figure 6 for representative speedup on symmetric multicore, shared memory systems. As in the Monte Carlo Pi study, the simulations plotted in the figure were carried out on a 64-bit Windows 7 machine with two Intel Xeon E5506 quad-core processors and 24 GB of available RAM. Each data point represents the ratio of the best sequential running time to the observed parallel run time while simulating just over 20,000 cells for 10,000 time steps. The physical problem solved is that depicted in Figure 1 and discussed further in Figure 2 – a two-dimensional domain with an offset hole acted upon by a unit displacement applied to its left end. For comparison purposes, the ideal speedup curve is also provided. For low numbers of processors, the CA speedup is nearly ideal. With increasing number of processors, the burden of processor synchronization (or lock-stepping) increases, and the achieved speedup deviates from ideal. However, even when employing the full 8 processors, and thus competing for computing resources with the operating system, the speedup is a very respectable 6.7 (ideal being 8.0).

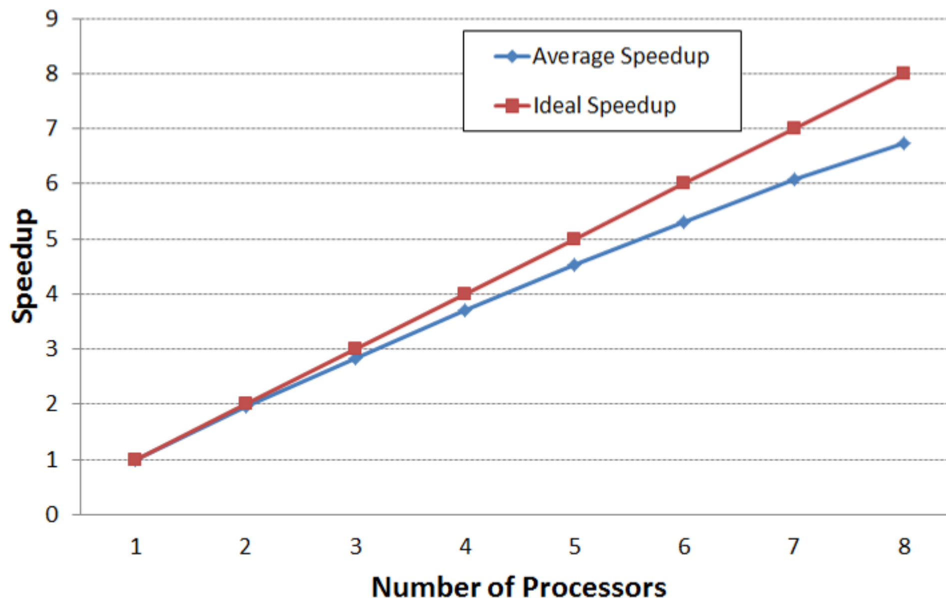

Figure 8: Speedup as a function of cores employed on a multicore shared memory system.

4.2 Distributed memory systems

Speedup studies of the parallel elastodynamic CA simulator were also carried out on the FoRCE cluster housed at Georgia Tech (more information available at http://pace.gatech.edu/force-cluster). This cluster is a heterogeneous mixture of compute

nodes, which at the time the simulations were performed consisted of 42 nodes, each with 24 AMD Opteron 8431 processors operating at 2.4 GHz. Each node had anywhere from 64 to 128 GB of RAM. The cluster is equipped with Infiniband interconnects; however, the slower Gigabit Ethernet (GigaE) interconnects were utilized since MPJ Express does not use Infiniband to its full potential, and speedup results suffered in comparison to results obtained using the GigaE interconnects. FastMPJ purports to be fully compatible with Infiniband and thus may be employed in the future when it exhibits better stability.

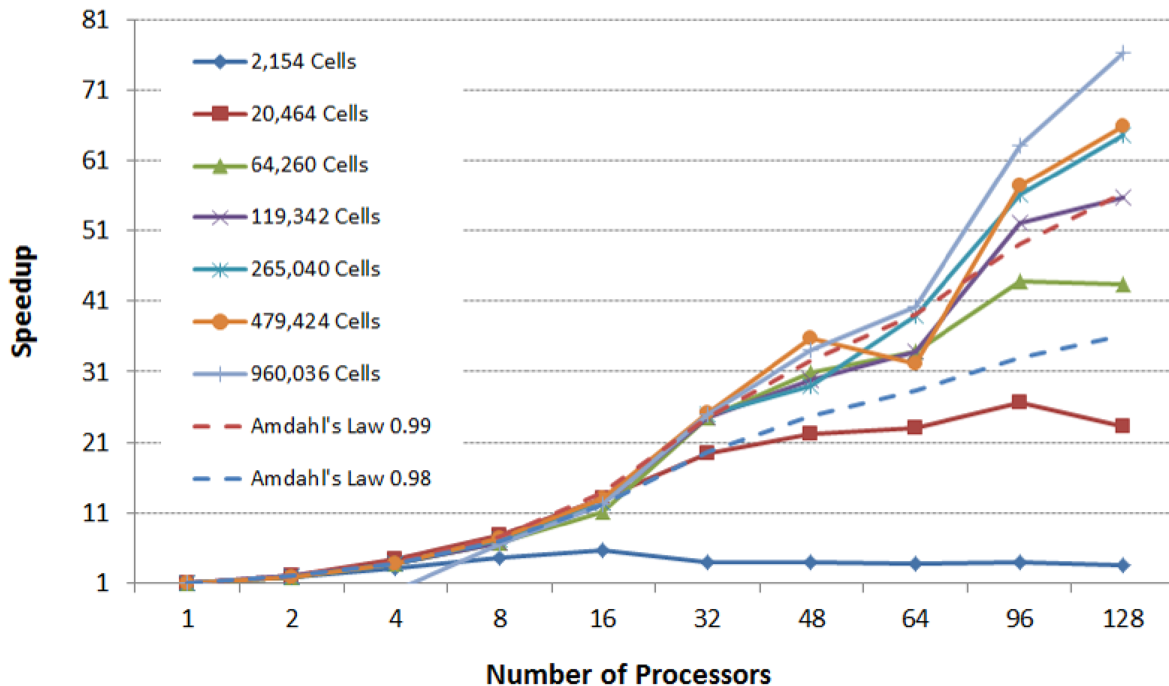

Figure 9: Speedup as a function of cores employed on a distributed memory cluster.

Figure 9 presents speedup results for a variety of cell counts and processors employed. The same physical problem considered in the shared memory multicore study is again simulated, with mesh refinement resulting in cell counts from just over 2,000 to nearly 1,000,000 cells. The commercial meshing package used restricted the number of cells to under 1,000,000. Note that a single node was employed for simulations using up to 16 processors, three nodes for processor counts from 32 to 64, and six nodes for processor counts of 96 and 128. In addition to the speedup observed for various cell counts, two curves (dashed lines) are presented which depict the maximum theoretical speedup using Amdahl's Law [15]. This law states that the maximum theoretical speedup as a function of processors employed is

$$S(n) = 1\Big/\left((1-p)\frac{p}{n}\right), \tag{3}$$

where S denotes the speedup, n denotes the number of processors, and p denotes the proportional of the program that is strictly running in parallel. Figure 9 plots Amdahl's Law for $p = 0.98$ and $p = 0.99$. Since no progress is made solving the problem during interprocessor communications, this proportion of the simulation effort can be thought of as that which accounts for the deviation of the actual (or achieved) p from 1. Thus comparing the achieved speedup curves to the Amdahl's curves provides some perspective on the overall efficiency of the described processor load-balancing and interprocessor communication approaches.

Several trends appear in Figure 9. First, the number of processors that can be employed effectively is highly dependent on the problem size. For cell counts on the order of 2,000 the speedup begins decreasing beyond 16 processors; for 20,000 cells the speedup increases out to 96 processors, and for nearly 1,000,000 cells the speedup curve indicates there is likely speedup beyond 128 processors. For this final processor count, the speedup for nearly 1,000,000 cells is 76.2. Note that this speedup is significantly greater than that predicted by Amdahl's Law using a parallel proportion of 99%.

The second significant trend in Figure 9 is the appearance of superlinear speedup. Speedup is said to be superlinear when an application runs faster on more processors than a linear speedup relationship would predict [16]. In Figure 9 there are two cases of this superlinear behavior. The first occurs with the 20,464 cells when increasing the number of processors from 2 to 4. This shows a speedup of 2.2 instead of 2. The second case occurs with 64,260 cells when increasing from 16 to 32 processors. This also shows a speedup of 2.2. Here, this superlinear scaling occurs when the amount of memory needed per processor transitions from being too large to fit into CPU cache, to an amount where it can fit entirely into cache. Once this occurs, each processor can spend more time computing state updates instead of waiting for memory to be delivered, and thus speedup increases beyond the linear regime.

5 CONCLUDING REMARKS

This paper describes a High Performance Computing implementation of a cellular automata simulation technique for predicting elastodynamic response on complex domains. It exploits the local, autonomous nature of the CA method to simply and effectively parallelize an existing *Java* code. Processor load balancing takes advantage of the neighbour information stored by each cell to stack processors with cells requiring a minimal set of neighbours on opposing processors. This also minimizes the necessary number of interprocessor calls, which combined with efficient packaging of state information passed between processors, results in a single call from one processor to another after each time step. Speedup results document very high efficiency of the presented approach. On shared memory multicore systems, the speedup follows closely the ideal speedup curve until the full processor count of the system is reached, at which point compute resources are in competition with the operating system. On distributed memory clusters, the speedup is a strong function of the number of compute cells. For very large problem sizes on the order of 1,000,000 cells, the speedup on 128 processors is over 75 times that of a single serial simulation. This indicates a parallel proportion of the compute burden in excess of 99% as predicted by Amdahl's Law. Finally, it is observed that the parallelization approach can occasionally achieve superlinear speedup when increasing numbers of processors allow more memory to be stored in CPU cache.

REFERENCES

[1] Chopard, B., and Droz, M., *Cellular Automata Modeling of Physical Systems*. Cambridge University Press, Cambridge, England, 1988.

[2] von Neumann, J., *Theory of Self-reproducing Automata*. University of Illinois Press, Urbana, IL, USA, 1966.

[3] Hopman, R. K., and Leamy, M. J., Triangular Cellular Automata for Computing Two-Dimensional Elastodynamic Response on Arbitrary Domains. *Journal of Applied Mechanics*, **78** (2), 021020, 1-10, 2011.

[4] Leamy, M. J., Application of Cellular Automata Modeling to Seismic Elastodynamics. *International Journal of Solids and Structures*, **45** (17), 4835-4849, 2008.

[5] Gropp, W., Lusk, E. L., and Skjellum, A., *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[6] Idesman, A., Samajder, H., Aulisa, E., and Seshaiyer, P., Benchmark Problems for Wave Propagation in Elastic Materials. *Computational Mechanics*, **43** (6), 797-814, 2009.

[7] Shafi, A., Carpenter, B., Baker, M., and Hussain, A., A Comparative Study of Java and C Performance in Two Large-Scale Parallel Applications. *Concurrency and Computation: Practice and Experience*, **21** (15), 1882-1906, 2009.

[8] Nikishkov, G., Nikishkov, Y. G., and Savchenko, V., Comparison of C and Java Performance in Finite Element Computations. *Computers & Structures*, **81** (24), 2401-2408, 2003.

[9] Taboada, G. L., Ramos, S., Expósito, R. R., Tourino, J., and Doallo, R., Java in the High Performance Computing Arena: Research, Practice and Experience. *Science of Computer Programming*, **78** (5), 425-444, 2013.

[10] Taboada, G. L., Touriño, J., and Doallo, R., Java for High Performance Computing: Assessment of Current Research and Practice. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, Calgary, Alberta, Canada, 30-39, 2009.

[11] Taboada, G. L., Touriño, J., and Doallo, R., F-MPJ: Scalable Java Message-Passing Communications on Parallel Systems. *The Journal of Supercomputing*, **60** (1), 117-140, 2012.

[12] Noer, G., Cygwin: A Free Win32 Porting Layer for Unix Applications. *http://cygwin.com*, 1998.

[13] Baker, M., Carpenter, B., and Shafi, A., MPJ Express: Towards Thread Safe Java HPC. *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 1-10, 2006.

[14] Paszynski, M., Kurtz, J., and Demkowicz, L., Parallel, Fully Automatic HP-Adaptive 2D Finite Element Package. *Computer Methods in Applied Mechanics and Engineering*, **195** (7-8), 711-741, 2006.

[15] Amdahl, G. M., Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities (Reprint from a 1967 AFIPS Conference Proceedings). *Solid-State Circuits Newsletter, IEEE*, **12** (3), 19-20, 2007.

[16] Gustafson, J., Fixed Time, Tiered Memory, and Superlinear Speedup. *Proceedings of the Fifth Distribued Memory Computing Conference (DMCC5),* Charleston, SC, USA, 1255-1260, 1990.