A Parallel Implementation of the Finite-Domain Time-Difference Algorithm using MPJ Express

Aamir Shafi
NUST Institute of Information Technology
Pakistan
aamir.shafi@niit.edu.pk

Aftab Hussain
NUST Institute of Information Technology
Pakistan
aftab.husain@niit.edu.pk

Jamil Raza
NUST Institute of Information Technology
Pakistan
jamil.raza@niit.edu.pk

Abstract

This paper presents and evaluates a parallel Java implementation of the Finite-Difference Time-Domain (FDTD) method, which is a widely used numerical technique in computational electrodynamics. The Java version is parallelized using MPJ Express—a thread-safe messaging library. MPJ Express provides a full implementation of the mpiJava 1.2 API specification. This specification defines a MPI-like binding for the Java language. This paper describes our experiences of implementing the Java version of the FDTD method. Towards the end of this paper, we evaluate and compare the performance of the Java version against its C counterpart on a 32 processing core Linux cluster of eight compute nodes.

1 Introduction

Soon after its release in 1996, Java became one of the "mainstream" programming languages of the software industry. Various computer scientists have argued [3] that Java could also make an excellent language for developing scientific codes. To date this argument has not convinced too many practising computational scientists. The scarcity of high-profile number-crunching codes implemented in Java does not help the case.

Perhaps a reason behind skepticism about Java for High Performance Computing (HPC) is the fact that Java initially executed as an interpreted language. The performance suffered in this case because the Java Virtual Machine (JVM) interpreted the bytecode generated by the Java compiler (`javac`). An alternative and more efficient approach is to generate and execute the native machine code as one would with C or Fortran. The pre-requisite for this is to convert the bytecode into the native machine code. This is what Just-In-Time (JIT) compilers do. The JVM first converts the bytecode into the native machine code and later executes the native code. Currently, there are at least two production-quality and free-to-download implementations of the JVM—one from Sun and one from IBM. Both of these JVMs are equipped with JIT compilers.

Compared with C or Fortran, the advantages of the Java programming language include higher-level programming concepts, improved compile time and runtime checking, and, as a result, faster problem detection and debugging. In addition, Java's automatic garbage collection, when exploited carefully, relieves the programmer of many of the pitfalls of lower-level languages. The Java language includes a large set of libraries that can be reused by application developers for rapid application development. Code type safety is yet another feature that helps a beginner to avoid syntactic bugs in their code.

The most attractive feature of applications written in Java is that they are portable to any hardware or operating system, provided that there is a JVM for that system. The contribution of the JVM is significant, keeping in mind that it allows new programmers and scientists to focus on issues related to their application and domain of interest, and not on system heterogeneity.

In the context of "Java for HPC", one of the authors was involved in an effort [8] to develop a Java version of massively parallel cosmological simulation code Gadget-2 [9]. Versions of the C Gadget-2 code have been used in the "Millennium Simulation" [10], which is heralded as the largest ever simulation of the Universe. It evolved 10^{10} dark matter particles from the early Universe to the current day. The Java version was developed as an experiment to help understand where Java stands in comparison to C—an already established HPC language. The performance evaluation of the Java version revealed that it could achieve comparable performance to the original C code.

In this paper, we implement a parallel version of the Finite-Difference Time-Domain (FDTD) method in Java. We used MPJ Express [2]—a thread-safe implementation of Message Passing Interface (MPI) [6] bindings in Java—to parallelize the implementation in Java. The FDTD method is a widely used and increasingly popular method for the study of electromagnetic wave propagation. The FDTD method has been successfully applied to a broad range of applications. These applications include antenna design, radar and photonic crystals. In addition there are emerging applications of the FDTD method in areas such as biophotonics and nanophotonics [11].

We also compare the performance of the parallel Java FDTD with its C counterpart, which was developed as part of our earlier study [4]. The C code is already parallelized using the MPI library. A goal of the performance comparison between C and Java codes is to understand the relative performance in a popular scientific algorithm like the FDTD.

Section 2 presents the FDTD algorithm. Section 3 discusses the design and implementation of MPJ Express. This is followed by the parallel implementation of the FDTD method in Section 4. Section 5 evaluates the performance of the parallel Java FDTD code on a Linux cluster of eight compute nodes. We conclude and discuss future work in Section 6.

2 The Finite-Domain Time-Difference (FDTD) Method

The numerical foundation of the FDTD method is the discretization of Maxwell's curl equations through the use of central difference approximations to both the space and time differentials. These discretizatons result in coupled finite difference equations that govern the propagation of electric and magnetic fields on a discrete numerical grid. With appropriate boundary conditions these difference equations are then iterated on a computer in a time and space marching sequence to study the propagation of electromagnetic waves. The Maxwell's curl equations for a source less and homogeneous medium are:

$$\nabla \times \bar{E} = -\mu \frac{\partial \bar{H}}{\partial t} \tag{1}$$

$$\nabla \times \bar{H} = \epsilon \frac{\partial \bar{E}}{\partial t} \tag{2}$$

Where ϵ is the electrical permittivity and μ is the magnetic permeability.

For 2D Transverse Magnetic (TM) waves the Maxwell curl equations reduced to the following form:

$$-\mu \frac{\partial H_x}{\partial t} = \frac{\partial E_z}{\partial y} \tag{3}$$

$$\mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x} \tag{4}$$

$$\epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \tag{5}$$

Where E_z is the electric field component in the z direction, and H_x and H_y are the magnetic field components in the x and y directions, respectively. After discretizing equations (3)-(5) in both time and space, we obtain the following discrete update equations for 2D TM waves [7]:

$$\begin{aligned} H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] = \\ H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}\right] - \\ \frac{\Delta t}{\mu \Delta y}\left(E_z^q[m, n+1] - E_z^q[m, n]\right) \end{aligned} \tag{6}$$

$$\begin{aligned} H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n\right] = \\ H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n\right] - \\ \frac{\Delta t}{\mu \Delta x}\left(E_z^q[m+1, n] - E_z^q[m, n]\right) \end{aligned} \tag{7}$$

$$\begin{aligned} E_z^{q+1}[m,n] = E_z^q[m,n] + \\ \tfrac{\Delta t}{\epsilon \Delta x}\left\{H_y^{q+\frac{1}{2}}\left[m+\tfrac{1}{2}, n\right] - H_y^{q+\frac{1}{2}}\left[m-\tfrac{1}{2}, n\right]\right\} - \\ \tfrac{\Delta t}{\epsilon \Delta y}\left\{H_x^{q+\frac{1}{2}}\left[m, n+\tfrac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m, n-\tfrac{1}{2}\right]\right\} \end{aligned} \tag{8}$$

Here m and n are the spatial indices while q is the temporal index. Δt is the size of the time step in the numerical FDTD grid and Δx and Δy are the sizes of the spatial steps in the x and y directions, respectively. The electric and magnetic fields evolve in time and space on a staggered FDTD grid. If we study electromagnetic wave propagation in a structure with a complex geometry we must make the spatial grid finer to capture the fine detail of the structure. However if we make the spatial steps smaller the Courant stability criterion [11] says we must also make the time step smaller. This results in a greatly increased simulation time.

$$\Delta t \leq \frac{1}{c\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}}} \qquad (9)$$

Equation (9) is the Courant stability criterion for the two dimensional FDTD method. Here c is the speed of light. If the Courant stability condition is not satisfied the simulation will rapidly blow up and generate incorrect results.

3 Overview of MPJ Express

Initial interest in using Java for HPC led to the formation of the Java Grande Forum [5] that was a platform for HPC community to voice their opinion about Java and how to improve it. The Java Grande Forum was a group of leading researchers from academia and industry that aimed to exploit Java for *Grande* applications. The forum proposed various improvements to Java for numerical computing—some of these were introduced into the standard Java language. The Message Passing Working Group of this forum defined Java bindings for the MPI standard. The two commonly used bindings, namely the MPI and mpiJava 1.2 APIs were published by the Java Grande Forum.

MPJ Express is a new implementation of Java bindings for the MPI standard. The system provides thread-safe communication in a Java messaging system. It addresses potentially contradictory issues of performance and portability by supporting pluggable transport devices based either on Java NIO (pure Java) or high performance interconnects like Myrinet. Another implementation of the Java bindings is the mpiJava [1] software. It uses JNI to interact with the underlying native MPI library. The project started in 1997 at NPAC (Syracuse University), later moved to the University of Florida, and then Indiana University.

MPJ Express has a layered design that allows incremental development, and provides the capability to update and swap layers in or out as needed. Figure 1 is a layered view of the messaging system that shows MPJ Express levels: high-level, base-level, *mpjdev*, and *xdev*.

Interested readers are refered to [2] for a detailed discussion on the MPJ Express software. MPJ Express is available as an open source software under Lesser GNU Public License (LGPL) from `http://mpj-express.org`.

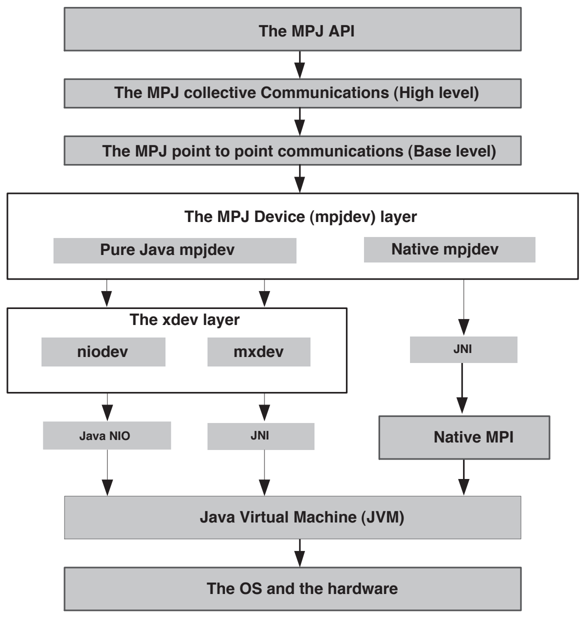
The MPJ API
The MPJ collective Communications (High level)
The MPJ point to point communications (Base level)
The MPJ Device (mpjdev) layer
Pure Java mpjdev
Native mpjdev
The xdev layer
niodev
mxdev
JNI
Java NIO
JNI
Native MPI
Java Virtual Machine (JVM)
The OS and the hardware

Figure 1. MPJ Express Design

4 Implementation of the FDTD Method

In this section we discuss the parallel implementation of the FDTD method using MPJ Express.

The FDTD method is amenable to parallelization using the message passing paradigm. The reason is that the update of the field values on the FDTD computational grid are dependant on the immediate neighboring field values.

In the parallel version, the domain decomposition is achieved by dividing the computational grid equally amongst the processors—this allows each processor to concurrently execute its computational domain. For instance, imagine a computational grid of $size_x$ and $size_y$ grid points in x and y directions, respectively. In addition, if there are P number of processes, then we can distribute the grid into $(size_x \times size_y)/P$ sub-computational grid between P processes equally.

While updating the boundary electric and magnetic field data structures, each process requires field values from its neighboring process. For this purpose, processes exchange field values at the boundaries. This exchange of values can be done by communicating the respective field values in a point by point fashion. This kind of fine grain communication normally leads to an excessive communication cost. In order to avoid this, the entire column of *ghost values*—the required column on neighboring processor—is exchanged between the processes leading to less communication cost. In our implementation, the ghost values are communicated using non-blocking point-to-point MPI functionality.

Figure 2 shows the pseudocode for MPI version of the

FDTD code.

4.1 Cavity Resonator Simulation using the Parallel FDTD Method

In this section we describe the FDTD simulation used for the comparison of Java and C parallel FDTD codes.

Our parallel codes are based on the sequential FDTD code given in [7]. The simulation is of a Ricker Wavelet propagating in free space surrounded by Perfectly Electrically Conducting (PEC) walls, which reflect impinging electromagnetic waves.

The computational problem domain consists of a 8192×8192 grid. The source is a Ricker wavelet, which is a pulse-like waveform. It is equal to the second derivative of a Gaussian. Pulse-like sources are used in FDTD simulations when it is desired to excite a structure with a broad range of frequencies. The Ricker wavelet emerges from the middle of the computational grid and propagates towards the walls of the cavity resonator. When it reaches the walls of the grid, it is reflected back because of the Perfectly Electrical Conducting (PEC) boundary conditions. As the simulation proceeds, the Ricker wavelet is confined to propagate within the resonator and we observe interference patterns generated by the confined waves.

Figure 3 shows snapshots from the simulation of a cavity resonator at three different times. In this case, the grid size is 256×256. The snapshots are shown for E_z field components. The entire simulation ran for 1000 time steps.

5 Performance Evaluation

In this section we report evaluation results of the Java and C MPI codes.

We begin by describing our test environment, which consisted of a 32 processing core Linux cluster at our institute. The cluster consists of eight compute nodes. Each node contains a quad-core Intel XEON processor. The nodes are connected via Myrinet and Fast Ethernet and we configured the OpenMPI runtime to use Myrinet for communication. The compute nodes run the SuSE Linux Enterprise Server (SLES) 10 Operating System and GNU C Compiler (GCC) version 4.1.0. Each compute node has 2 GBytes of main memory. We used OpenMPI version 1.2.4 as the C MPI library. For the parallel Java version, we used the latest development version of MPJ Express with the Sun Java Development Kit (JDK) 1.6 Update 4.

Figure 4 presents the performance comparison of the Java and C versions. Both Java and C versions of the FDTD method almost achieve similar performance. Also the two versions roughly scale in a similar fashion. The C code was executed with the GCC compiler version 4.1.0 and optimization switch -03 was specified.

```
Hx, the magnetic field component along x axis
Hy, the magnetic field component along y axis
Ez, the electric field component
rank, process rank in the world communicator
size, total number of processes

initialize field components

for initial time:maxtime {

  if(rank != 0) {
    Asynchronously send Ez edge column to
                            process rank-1
  } else if(rank != size-1) {
    Asychronously receive Ez edge column
                       from process rank+1
  }

  for (i=1 to imax) {
    for (j=1 to jmax) {
      Hx(i,j) = Hx(i,j) -
                  (Ez(i,j+1) - Ez(i,j))
    }
  }

  Wait for the Ez edge column communication
                                  to complete

  for (i=1 to imax) {
    for(j=1 to jmax) {
      Hy(i,j) = Hy(i,j) +
                  (Ez(i+1,j) - Ez(i,j))
    }
  }

  if(rank != size-1) {
    Synchronously send Hy edge column to
                             process rank+1
  } else if(rank !=0) {
    Synchronously receive Hy edge column from
                             process rank-1
  }

  for(i=1 to imax) {
    for(j=1 to jmax) {
      Ez(i,j) = Ez(i,j) +
                  (Hy(i,j) - Hy(i-1,j) -
                  (Hx(i,j) - Hx(i,j-1))
    }
  }
}
```

Figure 2. Pseudocode for the parallel FDTD method

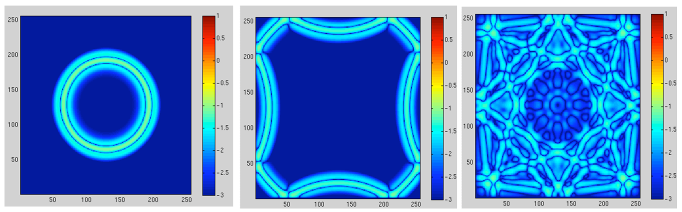

Figure 3. Snapshots Taken from the Simulation at Different Times

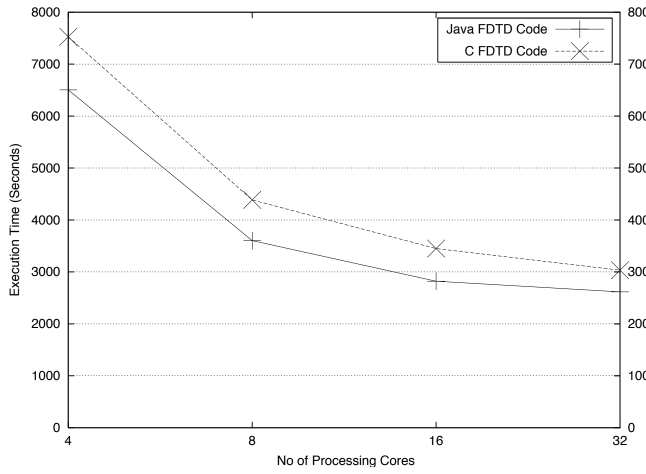

Figure 4. Java and C FDTD Codes Comparison

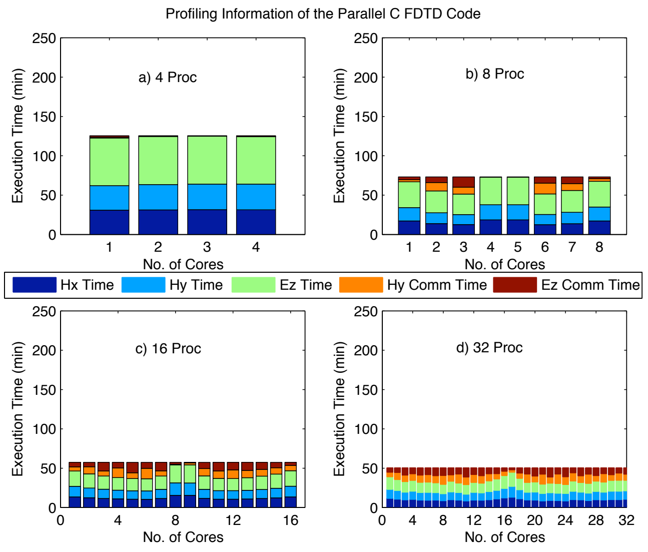

Figure 5. Profiling Information for the C FDTD Code

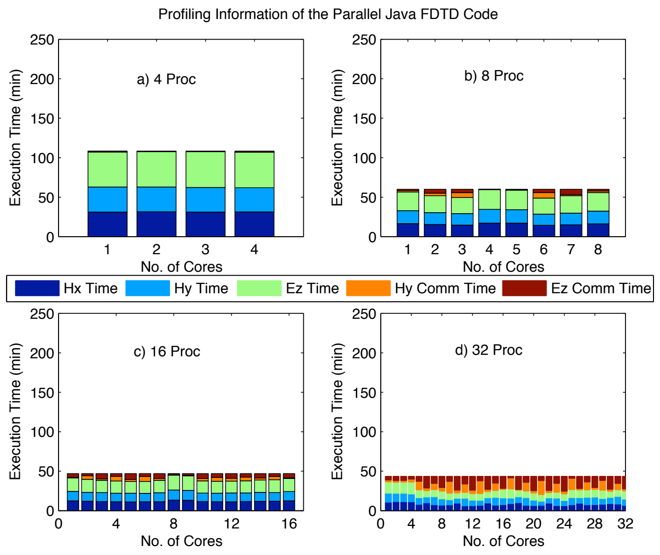

Figure 6. Profiling Information for the Java FDTD Code

To understand the comparative performance of the two versions of the FDTD code, we plotted the profiling information in Figure 5 and 6. Figure 5 plots the profiling information for the C FDTD code; here 5a, 5b, 5c, and 5d corresponds to execution results with 4, 8, 16, and 32 MPI processes. For all cases, the graph shows the time each process spent in various stages of the FDTD computation. These stages include H_x computation, H_y computation, E_z computation, H_y communication, and E_z communication. Similarly Figure 6 plots profiling information for the parallel Java FDTD code; here 6a, 6b, 6c, and 6d corresponds to execution results with 4, 8, 16, and 32 MPJ Express processes.

The profiling information in Figures 5 and 6 show that the Java code performs better in the computation stages. Modern JVMs are equipped with JIT compilers, which first convert the Java bytecode to native machine code. Also we observe that with 32 MPI processes, the communication cost for exchanging edge columns for H_y and E_z computation are higher for the Java code. This is understandable because of higher latency of the MPJ Express Myrinet device than its OpenMPI counterpart.

6 Conclusions and Future Work

This paper presented a parallel version of the FDTD algorithm for computational electrodynamics. An aim of this paper was to assess performance concerns of using Java in a real-world scientific algorithm. In this context, we produced a parallel Java version and studied its performance on our Linux cluster of eight compute nodes. We found that the Java and C versions scale in a similar fashion. Also the Java code achieves similar performance as the C code.

The performance shown by the parallel Java FDTD code shows that Java is a viable option for developing scientific applications. In general, Java encourages good software engineering by being an object-oriented language that is more portable than its precursors.

Currently MPJ Express, our implementation of Java bindings, provides communication devices using Java NIO and Myrinet. We have plans to develop drivers for Infiniband, Quadrics, and shared memory platforms. MPJ Express can be downloaded from `http://mpj-express.org`.

Acknowledgments

The authors would like to thank Bryan Carpenter and Mark Baker and acknowledge that some of the work—development of MPJ Express and Java Gadget-2—described in this paper was carried out at University of Portsmouth and University of Reading.

References

[1] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. An Object-Oriented Java interface to MPI. In *Proceedings of the International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[2] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster 2006)*, pages 1–10, Barcelona, Spain, September 2006. IEEE Computer Society.

[3] Geoffrey Fox. Editorial: Java for Computational Science and Engineering - Simulation and Modeling. *Concurrency: Practice and Experience*, 9(6):413–414, June 1997.

[4] Aftab Hussain, Jamil Raza, and Aamir Shafi. A Comparative Performance Study of Message Passing and Distributed Shared Memory Implementations of the Finite-Difference Time-Domain Algorithm. Technical Report NIITHPC-TR-01-08, January 2008.

[5] The Java Grande Forum. http://www.javagrande.org.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, 1995. www.mcs.anl.gov/mpi.

[7] Schneider webpage. http://www.eecs.wsu.edu/ schneidj/ee535.

[8] Aamir Shafi. *Nested Parallelism for Multi-core Systems Using Java*. PhD thesis, Distributed Systems Group, University of Portsmouth, November 2006.

[9] Volker Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105, 2005.

[10] Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Joerg Colberg, and Frazer Pearce. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629, 2005.

[11] Allen Taflove and Susan Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, volume 3. Artech House Publishers, June 2005.