Overlapping Communication with Computation in Distributed Parallel FDTD Codes Implemented in Java

C. G. Biniaris[1], A. I. Kostaridis[1] and D. I. Kaklamani[1]

Summary

This paper presents some key aspects regarding the overlapping of communication with useful computations in distributed parallel FDTD codes implemented in the Java programming language. This overlapping can be achieved due to the formulation of the FDTD method, which allows the inner mesh points in every sub-region of the computational domain to be updated in every time step of the algorithm, without any need for communicating neighboring field values. The process of overlapping communication with useful computations is made easy in a numerical code implemented in Java due to the language's inherent multithreaded nature which allows flexible thread management mechanisms.

Introduction

The Finite-Difference Time-Domain (FDTD) method is undoubtedly the most popular numerical method for the numerical solution of electromagnetic problems. FDTD, as was first proposed by Yee in 1966 [1], constitutes a simple and elegant way of discretization of the differential form of Maxwell's equations.

One of the more important advantages of the FDTD method is the ease of its parallelization. Many parallel codes have been developed in the past, demonstrating excellent speed- ups [2]. Nevertheless, for many years, the adaptation of the parallel code in new systems was extremely difficult because these codes were developed taking into consideration their execution in specific target systems.

Moving away from custom parallel architectures, the current common practice is to exploit the computing resources of network connected computers. Towards this approach, several standards and libraries [3],[4] have been proposed to enable the development of parallel codes with minimum effort. These tools, indeed, take away much of the complexity related to code parallelization by hiding the underlying inter-process communication mechanisms from the code developer. However, future distributed applications should exploit the computing resources of not only a set of workstations connected to a LAN, but ideally of any PC connected to the Internet.

Towards this goal, there is a strong need for the scientific community to adopt new tools, architectures and computing paradigms. The Java programming language is the ideal candidate to cover these demands due to its pure object orientation, its platform

[1] National Technical University of Athens, 9, Iroon Polytechniou Str., Zografos Athens, GR-15780 GREECE

independence and its multithreading and networking capabilities. These capabilities can be enhanced by distributed object frameworks, which enable the implementation of a Java application as a set of distributed objects. Such frameworks provide object location transparency, hide the communication details from the programmer and enhance Java serialization mechanisms [5-7]

In the following sections we present some basic concepts regarding the parallelization of FDTD method. Next, we present aspects related to the overlapping communication with computation in a distributed parallel two dimensional numerical code implemented in Java, focusing on thread notification and synchronization mechanisms. Finally, we give some concluding remarks.

Basic Concepts of FDTD Parallelization

The FDTD algorithm is "data - parallel" and an efficient parallel solution can be achieved using domain decomposition techniques. A one dimensional decomposition of a two dimensional FDTD computational domain is depicted in Figure 1 for the case of TMz mode.

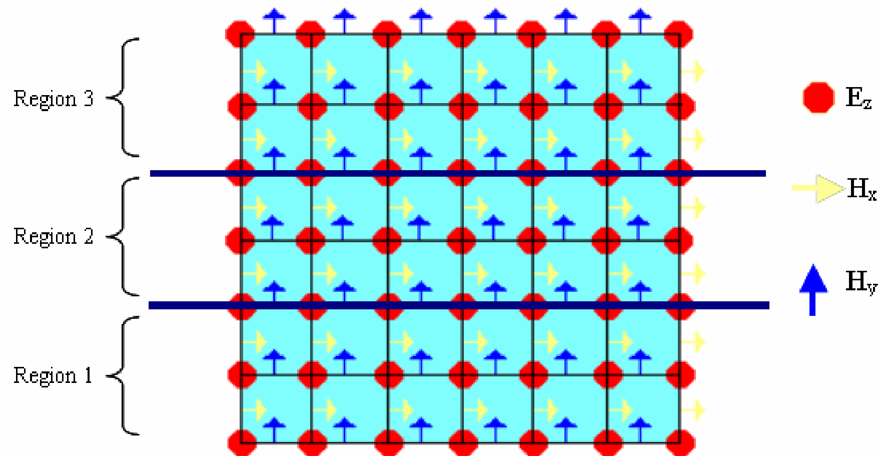

Figure 1. One dimensional domain decomposition of a two dimensional FDTD mesh

The calculation of the field values in the sub-regions is assigned to a set of distributed objects, each one of which is responsible to handle the field update process of its allocated sub-region. The size of every sub-region is chosen by a load balancing algorithm, in order to minimize the hosts idle time during each time step.

Due to the formulation of FDTD algorithm, each tangential E component located on the shared boundary between two sub-regions is updated in the usual explicit manner using the values of the neighboring H components. In Figure 1, for instance, the update process of the E_z component in every time step requires the neighboring H_x and H_y

components' values. Each object has direct access to both H_y components, however one of the two needed H_x components belongs to the neighboring sub-region. Hence, in every time step there is a need for communication between objects, in order to exchange the required data. On the other hand, each object can update independently the H components in every sub-region without the need for any communication [2].

Overlapping of communication with useful computations

Given that the process of requesting the required H components from the neighboring sub-regions is independent of the update process of the rest E (inner) components in the sub-region, these tasks can be assigned to independent threads. As a result, the process of communication and update of the boundary E components can be executed independently of the field update process of the inner E components.

For the case of the two dimensional problem depicted in Figure 1, each distributed object responsible to update the fields in its sub-region creates two inner classes named UpperCommunicator and LowerCommunicator. The object instantiates the objects of these two classes according to its needs. For instance, the object responsible to update the fields in the top sub-region instantiates only the LowerCommunicator object in order to communicate with its lower neighbor. In the same way, the object responsible to update the fields in the bottom sub-region instantiates only an UpperCommunicator object, while every other object instantiates both an UpperCommunicator and a LowerCommunicator object.

These objects extend the class java.lang.Thread, consequently their functionality is included in their run() method. Each object of the aforementioned classes is responsible to perform in every time step the following tasks:

- To call the appropriate method to the proxy of the neighboring object in order to receive the H required values.
- To receive these values when they become available.
- To update the corresponding E components which lie tangentially on the shared boundary.
- To inform the main object's thread, which handles the field update process of the sub-region, that the update process of these boundary E components has been completed.

The pseudocode depicted in Figure 2, describes the functionality of each thread. It has to be ensured that the body of the loop will be executed once in each time step. For this reason, once each thread enters the body of the loop, it is set to a waiting state, waiting for the notification by the object to perform its work. Since such a notification takes place once in each time step, the thread will execute the body of the loop also once in a time step.

```
public void run() {
    while(true) { // the body of this loop is executed once in every time step
                  // the thread is in waiting state
                  // the thread is notified by the object
                  // the thread performs the required tasks
                  ...
    }
}
```

Figure 2. The pseudocode of a thread which requests values from neighboring sub-regions

Thread notification mechanisms

In order to enable each thread to exit the waiting state and perform its task in every time step, a notification mechanism has to be used in order to enable each thread to execute its task immediately after it is requested. This mechanism is based on the thread notification capabilities provided by Java.

For the purpose of thread notification, an inner class with the name Synchronizer was implemented. For each thread, an object of the class Synchronizer is instantiated, serving as the notification mechanism for the thread in the following manner:

The thread is set in the waiting state by calling the block() method of the corresponding Synchronizer object. When this method is called, the thread waits on the Synchronizer object's monitor due to the fact that the body of the block() method contains the call to the Thread.wait() method.

In order to cause the thread's exit from the waiting state, the object's main thread calls the wake() method of the Synchronizer object. As soon as this method is called, the object, that is waiting on the Synchronizer object's monitor, is notified, consequently the thread's block() method returns and the thread performs the rest of its tasks included in the while loop.

The code of the Synchronizer class is depicted in Figure 3.

```
class Synchronizer{
    public synchronized void block() {
        System.out.println("Blocking started");
        try {
                wait();
        }
        catch(InterruptedException iex) {}
    }
    public synchronized void wake() {
        System.out.println("Notified");
        notify();
    }
}
```

Figure 3. The code of the Synchronizer class

Thread synchronization mechanisms

In order to be considered by the object's main thread that the E field update process for the current time step has been completed, all the E components in the sub-region (internal and boundary) have to be updated. Since these processes have been delegated to independent threads, there has to be a synchronization mechanism among these threads.

According to this mechanism, if the E field update process completes before the H values from the neighboring sub-regions are received and the corresponding E components are updated, the object's main thread does not perform any other tasks and the method which updates the inner E components returns. In this case, the main object's method, which monitors the received results from the calls to remote objects, is responsible to decide when the object will proceed to the remaining tasks for the current time step, which mainly involve the update of the H components in the sub-region.

In any other case, where the boundary E components have been updated by the other threads during the update process of the inner components, the object's main thread is ready to proceed to the other tasks for the current time step

Conclusions

In this paper we have presented some basic aspects regarding the overlapping of communication with useful computations in distributed parallel FDTD codes implemented in the Java programming language. The communication tasks are assigned to independent threads which are responsible to receive the required data in order to

update the field components which lie on the shared boundary between the sub-regions, while the update process of the inner components takes place.

Our main purpose was to demonstrate that the inherent characteristics of Java, relevant to thread management and notification, can lead to a straightforward implementation of a mechanism to overlap communication between distributed objects with useful computations.

References

1 K. S. Yee (1966): "Numerical Solution of Initial Boundary Value Problems involving Mawxell's Equations in Isotropic Media", *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302-307.

2 A. Taflove (1995): *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House. Inc.

3 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam (1994): *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press.

4 W. Gropp, E. Lusk, and A. Skjellum (1994): *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press.

5 OMG CORBA (1995) Common Object Request Broker Architecture and Specification, Revision 2, August 1995, www.omg.org

6 T. Megedanz (1997): " Mobile Agents – Basics, Technologies, and Applications (including Java and CORBA Integration)", *Invited Tutorial in IEEE IN Workshop, Colorado Springs, USA*.

7 C. G. Biniaris, A. I. Kostaridis, D. I. Kaklamani and I. S. Venieris (2002): "Implementing distributed FDTD codes with Java mobile agents", *IEEE Antennas and Propagation Magazine*, 44 (6), pp.115-119.