REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
04-02-2008	Journal Article	

4. TITLE AND SUBTITLE

Implementation and Scalability of a Pure Java Parallel Framework with Application to Hyperbolic Conservation Laws (Preprint)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael Kapper & Jean-Luc Cambier (AFRL/RZSA)

5d. PROJECT NUMBER

5e. TASK NUMBER
23040256

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory (AFMC)
AFRL/RZSA
10 E. Saturn Blvd.
Edwards AFB CA 93524-7680

8. PERFORMING ORGANIZATION REPORT NUMBER

AFRL-RZ-ED-JA-2008-038

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory (AFMC)
AFRL/RZS
5 Pollux Drive
Edwards AFB CA 93524-7048

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S NUMBER(S)
AFRL-RZ-ED-JA-2008-038

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited (PA #08069A).

13. SUPPLEMENTARY NOTES
For publication in the Journal of Concurrency and Computation: Practice and Experience.

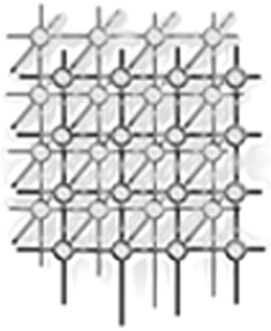
14. ABSTRACT

We introduce a pure Java parallel framework for Single Process, Multiple Dataset (SPMD) applications, intended for time-accurate solutions of hyperbolic conservation laws. The software architecture is based upon an extension of the client-server paradigm, utilizing a tree database abstraction and allowing for multi-tiered network configurations. The framework is designed to be hardware independent, with the ability to handle both shared-memory and distributed-memory hardware alike, allowing execution over heterogeneous networks. Task division is determined through permanent domain decomposition, in which Java threads are created for each computation domain and are then distributed over the available servers. Java Remote Method Invocation (RMI) is used for network-based communication of critical I/O as well as thread communication and cooperation between different Java Virtual Machines (JVM). Parallel efficiency and scalability of the framework for both shared-memory and distributed-memory hardware are evaluated for standardized benchmark problem computed with the Euler equations of gas dynamics. Results show efficient use of multiple process or resources on shared-memory systems with minimal thread overheads and near linear scalability on distributed networks with up to 50 server nodes (100 processors).

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Dr. Jean-Luc Cambier
a. REPORT	b. ABSTRACT	c. THIS PAGE	SAR	21	19b. TELEPHONE NUMBER *(include area code)*
Unclassified	**Unclassified**	**Unclassified**			N/A

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Implementation and scalability of a pure Java parallel framework with application to hyperbolic conservation laws (Preprint)

M. Kapper*,† and J.-L. Cambier†

Air Force Research Laboratory, Edwards AFB, CA 93524 U.S.A.

SUMMARY

We introduce a pure Java parallel framework for Single Process, Multiple Dataset (SPMD) applications, intended for time-accurate solutions of hyperbolic conservation laws. The software architecture is based upon an extension of the client-server paradigm, utilizing a tree database abstraction and allowing for multi-tiered network configurations. The framework is designed to be hardware independent, with the ability to handle both shared-memory and distributed-memory hardware alike, allowing execution over heterogeneous networks. Task division is determined through permanent domain decomposition, in which Java threads are created for each computation domain and are then distributed over the available servers. Java Remote Method Invocation (RMI) is used for network-based communication of critical I/O as well as thread communication and cooperation between different Java Virtual Machines (JVM). Parallel efficiency and scalability of the framework for both shared-memory and distributed-memory hardware are evaluated for a standardized benchmark problem computed with the Euler equations of gas dynamics. Results show efficient use of multiple processor resources on shared-memory systems with minimal thread overheads and near linear scalability on distributed networks with up to 50 server nodes (100 processors).

KEY WORDS: Java; RMI; high-performance computing; scalability

*Correspondence to: M. Kapper, 10 E. Saturn Blvd., Edwards AFB, CA 93524, USA
†E-mail: {michael.kapper,jean-luc.cambier}@edwards.af.mil
‡Distribution A: Approved for public release; distribution unlimited.

Received
Revised

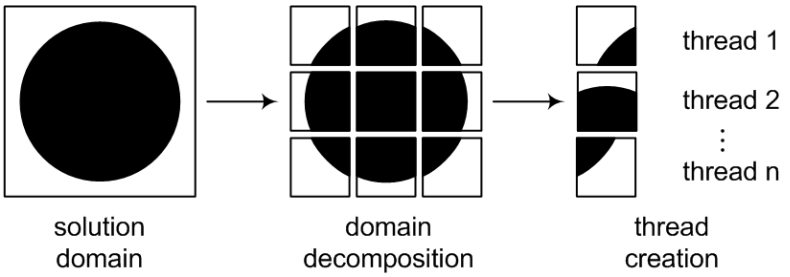


Figure 1. Domain decomposition process. Solution domain is decomposed into multiple computation domains from which new threads are created.

1. Introduction

Java is a highly attractive language for developing high performance computing (HPC) applications. Extensive thread support is built into the language, allowing programs to harness the power of multi-core and symmetric multiprocessing (SMP) hardware. The Remote Method Invocation (RMI) API allows seamless interaction with remote objects, simplifying the development of distributed applications. Java is inherently object-oriented, allowing key abstraction of various network environments. And its platform independence makes it ideal for distributing applications over heterogeneous computing platforms, thus maximizing hardware resources.

Making extensive use of these features, we have developed a distributed framework tailored for explicitly parallel, Single Program, Multiple Data (SPMD) programming applications on clustered networks. In particular, the framework is aimed at solving hyperbolic conservation laws and other problems in which parallelization can be accomplished via domain decomposition—the process by which the physical or solution domain is divided into multiple computation domains (see figure 1). These computation domains encompass instruction sets necessary for advancing the solution in time or achieving a steady-state solution and are executed concurrently and asynchronously through an iterative cycle with only periodic communication and synchronization requirements. Furthermore, the domain decomposition is overlapping such that each domain can communicate by exchanging boundary information with its neighbors.

The framework is implemented 100% in Java, maintaining independence of computing platforms for which the Java Virtual Machine (JVM) is available. This maximizes available computing resources with a minimum of programming effort and required maintainance. The framework relies upon Java threads in order to optimize performance on shared-memory machines. Each computation domain executes within its own lightweight process (LWP), created by spawning new threads. The resulting threadpool is distributed over multiple servers. Communication between different servers is based on the client-server paradigm and relies upon the Remote Method Invocation (RMI) API for network-based communication. RMI is

used to exchange boundary information from one domain object to another over the network. Communication must be synchronized and is implemented using concurrency constructs that encapsulate Java monitors.

The outline of the paper is as follows. Section 2 introduces the underlying hardware architecture, which is based upon an extension of the client-server paradigm for network-based computing. Extension of the client-server model is facilitated through the use of a tree data structure, allowing for a multi-tiered representation of heterogeneous networks. This allows horizontal and vertical scalability over clusters of shared-memory machines. Communication within the framework via RMI is discussed in Section 3 for inter-domain communication. In Section 4, we review the fundamental mechanisms in Java for thread communication and cooperation and show how to construct and implement higher level concurrency constructs from these mechanisms. These constructs serve as the foundation for all synchronization within the framework. Finally, results are presented in Section 5

2. Architecture

Java's distributed computing model is based upon the client-server paradigm. Objects distributed over client and server machines communicate with each other through Remote Method Invocation (RMI) protocols, allowing seamless interaction between applications. Programming with distributed objects greatly simplifies the development process and provides a natural extension of object-oriented programming to distributed environments. Because of its inherent simplicity, RMI is used as the basis for designing the framework around the client-server paradigm. Employing a request-response protocol, orchestrated with the aide of concurrency constructs, the framework allows interaction and communication between computation domains distributed over a network. Here we discuss the roles of clients and servers within the framework and extensions to a multi-tier implementation.

2.1. Client

From the user's point-of-view, the client is the access point into the entire framework. By means of a Graphical User Interface (GUI), the client allows remote-steering of simulations by the user. The user is able to configure, modify, and visualize the problem and solution in real time, all in a user-friendly environment. Before a simulation is initiated, problem-dependent tasks such as the set-up and initialization of boundary conditions, initial conditions, and numerical solvers can be configured through the client.

2.2. Server

Servers are responsible for the relevant program execution. This entails funneling critical I/O, communication between domains, synchronization, and execution of the domain threads. These tasks are divided between two different types of servers, masters and slaves. Master servers accept incoming requests and initialize slave servers for the client. Their primary function is to distribute the computation domains to slave servers and provide periodic barrier

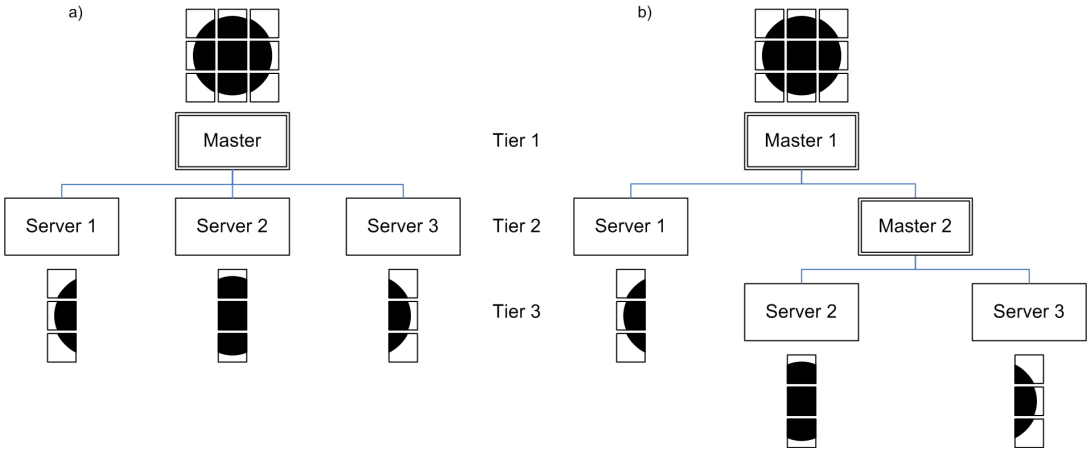


Figure 2. Master-slave configurations. In (a), a single Master node distributes workload among 3 slaves. In (b), Master 2 controls Slaves 1 and 2, but is itself a slave to Master 1.

synchronization between iteration cycles. Masters in turn initialize slave servers that respond to synchronization signals from their master to begin a cycle of program execution, returning signals when completed.

A simple master-slave interaction is demonstrated in figure 2. A master server distributes computation domains among multiple slave machines and requests for the advancement of the solution for a given time step. During this time, the servers may communicate with each other, requesting critical I/O such as domain boundary information. The client then awaits for their successful completion via signals from each of the servers, at which time it may request for solution results. The entire process is then repeated until the final solution is achieved.

This two-tier model demonstrates horizontal scalability, in which the bottom tier is expanded to accommodate the computational load requested by the master. This assumes that the master has direct access to all possible hardware resources. However, this may not always be the case. A master may want to tap into the resources of a Beowulf cluster for example, but may only do so through the head node of the cluster. In such a scenario, the head node assumes a dual role of accepting part of the computational workload from the master as well as distributing the load to its available nodes. This is an example of a three-tier model and is depicted in 3 below.

In order to support both horizontal and vertical scalability, the framework has been generalized to a multi-tiered, client-master-slave model. In a multi-tier network, servers can simultaneously function as both masters and slaves, far more hardware and networking environments can be used. Furthermore, horizontal scalability cannot be sustained indefinitely, and will eventually present a bottleneck for massively-parallel applications. Vertical scalability can be used to funnel I/O through multiple tiers, effectively reducing I/O bandwidth at each

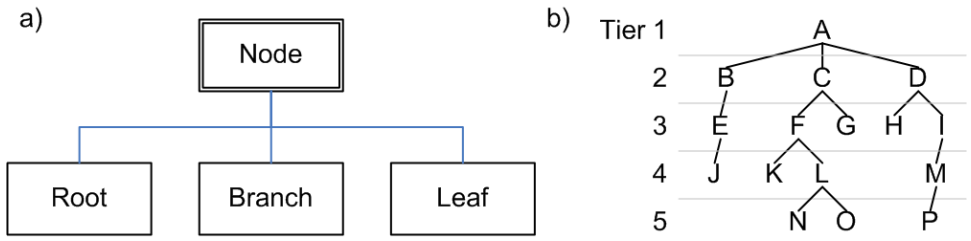


Figure 3. Tree abstraction. (a) *Root*, *Branch*, and *Leaf* are subclasses of *Node*. (b) Example configuration: A represents the *Root* node; B, C, D, E, F, I, L, and M represent *Branch* nodes; and J, G, H, K, N, O, and P represent *Leaf* nodes.

tier. The disadvantage is that multiple requests may need to be made in order to obtain the same response, increasing network latency time.

2.3. Tree Model

In the current architecture, the multi-tier client-server paradigm is implemented through a tree-based abstraction. As illustrated in figure 3, each node in the system can be represented as either a *Root*, *Branch*, or *Leaf* object. The standard tree terminology is also adopted in the following descriptions, including parent, child, sibling, etc.

2.3.1. Root

The *Root* node is the information focal point of the tree data structure. As such, for every network tree there can exist a single root node. It functions as a global master by synchronizing its child nodes. It is also the access point for the client to the entire framework. The root node can have as children any combination of branches or leaves. The root node has no parents but listens for data requests from the client for such tasks as visualization. The Root node is used as a central location for configuration files as well as mapping information about which domains belong to which servers.

2.3.2. Leaf

Terminal nodes in the network tree are called leaves and cannot have child Nodes. The primary function of a leaf is to host computational domains which handle the computational workload.

2.3.3. Branch

A branch assumes the same responsibilities as those of both root and leaf nodes. The branch must wait until signaled by its parent node (either a *Root* or *Branch* node) and in turn must

signal its child nodes (either *Leaf* or other *Branch* nodes) and finally wait for its children to finish their tasks. While it is possible, it is not necessary for a branch node to have computational domains. If it does, however, it must also signal and wait for the domains.

3. Communication

With computation domains distributed over a multi-tiered, heterogeneous network, communication becomes of critical importance. In the current framework, there are three primary types of information which must be communicated between different nodes in the network. These include

(i) boundary information shared between domains
(ii) global information
(iii) signals for thread communication and cooperation

In all three cases, Java RMI is the protocol used to establish a connection between the nodes, allowing objects to interact and share data. Here we provide details on the exchange process of boundary and global data. Thread communication is an integral part of the synchronization process and is discussed in the next section.

3.1. Boundary Communication

Continuity of the solution domain is left fragmented after the domain decomposition process. To restore continuity, each computation domain must exchange its boundary information with its neighbors during program execution. For time-accurate simulations, this translates into one or more boundary communications per time step.

In the current framework, boundary information is passed from one domain to another within buffer objects. Source buffers are used to store boundary information which originates on the local domain and is to be sent to the remote domain. The domain on the receiving end of the source buffer stores it locally as a "target" buffer. The information from the target buffer is then used to fill the ghost cells of that domain. This process is illustrated in figure 4.

A partial listing of the Buffer class implemented in the current framework is given below. Along with fields for storing the boundary information of the domain, the Buffer class also contains all necessary information to ensure that data is transferred and received in the proper order. This includes information about the source and target pairs for the domain, cell, and face indices. Encapsulating this information within the Buffer class greatly simplifies implementation of dynamic load balancing. Sending this information each exchange may seem extraneous and unnecessary, but the overhead is negligible since the performance limitation is based on the time it takes to make a remote call overshadows the time it takes to transfer the extra data. This "connectivity" information tells the local domain how to fill the buffer and how the remote domain extracts data from the buffer is determined a priori by the preprocessor. To be transferrable via the RMI protocol, the *Buffer* class implements *java.io.Serializable*.

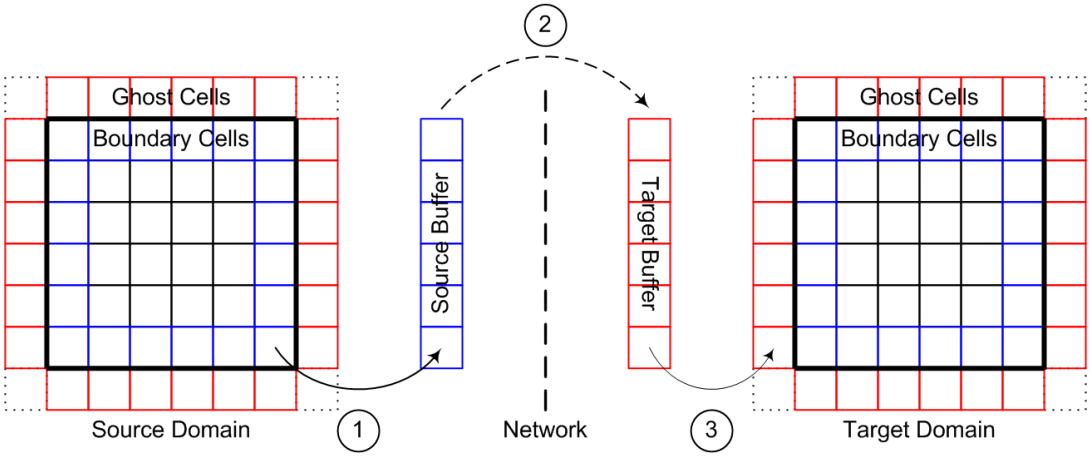


Figure 4. Domain Buffers are used to transfer data from the boundary layers of the source domain over the network and into the ghost layers of the target domain.

Before a method call is made to transfer the data, a check is in place to determine if the source and target domains exist within the same JVM. If so, a remote method call is initiated, otherwise a local call is made.

Listing 1. Buffer

```
public class DomainBuffer implements java.io.Serializable
{
    public int sourceDomain;
    public int targetDomain;

    public int sourceFace[];
    public int targetFace[];

    public int sourceCell[][];
    public int targetCell[][];

    public double[][][][] Q;

    ...
}
```

Buffer objects can either be pushed from the source domain to the target domain by a *set()* method or pulled from the source domain by the target domain by a *get()* method. This may seem like a trivial choice, but proper implementation of the two methods can be quite different, with one resulting in performance gains over the other. To analyze the differences between a push and a pull, we present two scenarios. The first case represents a push by the source domain while the second is a pull by the target domain. Both cases represent the steps taken by each domain during a complete time step with all domains assumed to be synchronized at the start the iteration.

Cycle I:

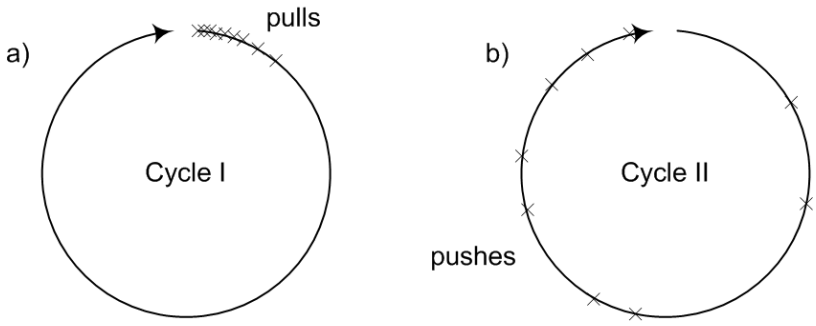


Figure 5. Push vs. Pull. Domain buffers can be transferred by either (a) a *pull* from the source domain by the target domain or (b) a *push* from the source domain to the target domain.

(i) Begin time step n.
(ii) Fill ghost cells with target buffer data.
(iii) Advance solution to time n+1
(iv) Fill source buffer with boundary cell data corresponding to time level n+1.
(v) Send source buffer to target domains.
(vi) End Iteration.

Cycle II:

(i) Begin time step n.
(ii) Get buffer from source domain.
(iii) Fill ghost cells with target buffer data.
(iv) Advance solution to time n+1
(v) Fill source buffer with boundary cell data corresponding to time level n+1.
(vi) End Iteration.

Note that the *push* at the end of the iteration by the source domain in Cycle I becomes a *pull* by the target domain at the beginning of the iteration in Cycle II. For the pull taking place at the beginning of the time step, it is more likely that all domains will attempt to get their boundary information at the same time, leading to a possible I/O bottleneck. This is less of an issue for a push, however, since it is less probable that all domains will finish their cycle at the exact same moment. This means that the domains will likely perform the push during a time when they would otherwise be idle, waiting for the final domain to finish its cycle. The chances for the remote calls being staggered in time is much greater, thus maintaining a higher level of asynchronicity. The degree of this effect is dependent upon how many domain threads are executing on each server and the thread scheduling characteristics of the operating system.

It is important to note that there exists the potential for synchronization issues with both of the above schemes. If either of the cases are not implemented properly, a critical race condition

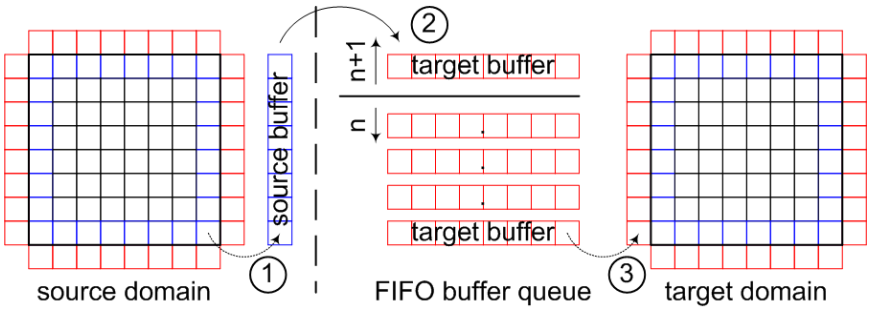


Figure 6. FIFO buffer queue. FIFO buffer queues on the receiving end of a push from the source domain are used to ensure correct synchronization of data between iterations.

may ensue. In both scenarios, it is possible for the source domain to have completed its iteration before the target domain has even begun. In this situation, the target domain will attempt to fill its ghost cells with buffer data from an incorrect time step—the next one (n+1).

This race condition can be remedied by storing the buffers on each domain in a First-In-First-Out (FIFO) queue as illustrated in figure 6. For Cycle I, as 6 illustrates, as source buffers are pushed to the target domain, they are added to the top of the buffer queue. The target domain then extracts these buffers from the bottom of the queue, ensuring that buffers of different time steps are effectively separated. The fix for Cycle II is similar, necessitating a FIFO buffer queue on the source domain from which the target domain *pulls*.

In the current framework, the buffer queue is implemented with the *java.util.LinkedList* class that provides such FIFO access. Since the object can be accessed by more than one thread, we must ensure that its methods are synchronized. This is accomplished by creating the *LinkedList* object in conjunction with the *java.util.Collections.synchronizedList()* static method.

4. Synchronization

The end of Moore's Law has ushered in a new era of symmetric multiprocessor (SMP) and multi-core platforms. As a result, successful applications must be able to exploit inherent parallelism to take advantage of the processing power available from all cores/processors. As a thread-based language with support for mutual exclusion and thread cooperation built in, Java is ideal for developing parallel applications on shared-memory machines. Any object (and corresponding class) can be locked by a thread that enters either a synchronized method or synchronized statement of that object. Once inside a synchronized region, a thread blocks all others from accessing any field or method. While a thread maintains a lock on an object, inter-thread communication and coordination is possible through the wait/notify mechanism. A thread can either invoke the *wait()* method, in which case the thread releases its hold on

the object and allows for the possibility that another thread may enter and obtain a lock, or it can invoke the *notify()* method as it leaves the synchronized region, to signal to awaiting threads that the object has become available.

The *wait/notify* mechanism provides the essentials for thread coordination, provided that critical sections are encapsulated within a synchronized region. For a domain decomposition-based parallelization scheme, a particular example of a critical section is code that accesses the buffer data objects. Since multiple buffers are accessed by multiple threads through *get/set* methods, explicitly encapsulating this critical code in synchronized regions can quickly lead to a complicated implementation that is difficult to understand. Furthermore, the *wait/notify* mechanism is a low level concurrency construct which does not guarantee against deadlock conditions.

As an alternative, the *wait/notify* mechanism itself can be encapsulated within a higher-level lock construct. Critical code can then interact with this construct, thereby invoking the *wait/notify* mechanism implicitly. The lock construct implemented in the current work is based on the BooleanLock utility of Paul Hyde [3]. BooleanLock is a shared class that encapsulates a boolean variable that can be accessed only through several synchronized methods. The boolean variable is a thread-safe condition variable upon which threads can wait for other thread to change its state to either true or false. Encapsulation of the *wait/notify* mechanism is provided in such a way that it is also thread safe, eliminating the possibility of deadlock conditions.

Listing 2. BooleanLock Utility

```
public class BooleanLock
{
    private boolean value;

    public synchronized boolean get()
    {   return value;
    }

    public synchronized void set(boolean value)
    {   this.value = value;
        notifyAll();
    }

    public synchronized void waitUntil(boolean value) throws InterruptedException
    {   while( this.value != value )
            wait();
    }

}
```

Within the framework, *BooleanLock* is used to synchronize several processes including initialization of the simulation, computation cycles, and visualization just to name a few. A unique lock is associated with each process for each thread. *BooleanLock* is further encapsulated within the *NodeSync* class listed below which is used directly within the framework. It provides methods to set, query, and wait on the state of individual threads as well as groups of threads. The *NodeSync* class also allows locks to be stored in *java.util.TreeMaps* so that they can easily be accessed through unique IDs.

Upon the completion of an iteration, each domain must wait for the final domain to complete its computations before the cycle can continue. This form of barrier synchronization is implemented by looping over all domain locks and waiting until each lock has been released.

This is accomplished in the code by invoking *waitUntil(DOMAINS, RUNNING, false)* at any point in the code where all domains must be synchronized together.

Listing 3. NodeSync

```
public class NodeSync implements NodeSyncInterface
{
    private int Nchildren;
    private int Ndomains;
    private int Nproxies;

    private Lock[] clientLock;
    private Lock[] localLock;

    private Map<Integer, Lock[]>  childLock = new TreeMap<Integer, Lock[]>();
    private Map<Integer, Lock[]> domainLock = new TreeMap<Integer, Lock[]>();
    private Map<Integer, Lock[]>  proxyLock = new TreeMap<Integer, Lock[]>();

    ...

    public void set(Identifier identifier, int state, boolean value) throws RemoteException
    {
        switch(identifier)
        {   case CLIENT:
                clientLock[state].set(value);
                break;
            case LOCAL:
                localLock[state].set(value);
                break;
            case CHILDREN:
                for(Lock[] lock : childLock.values())
                    lock[state].set(value);
                break;
            case DOMAINS:
                for(Lock[] lock : domainLock.values())
                    lock[state].set(value);
                break;
            case PROXIES:
                for(Lock[] lock : proxyLock.values())
                    lock[state].set(value);
        }
    }

    public void set(Identifier identifier, int index, int state, boolean value) throws RemoteException
    {
        switch(identifier)
        {   case CHILD:
                childLock.get(index)[state].set(value);
                break;
            case DOMAIN:
                domainLock.get(index)[state].set(value);
                break;
            case PROXY:
                proxyLock.get(index)[state].set(value);
                break;
        }
    }

    public boolean is(Identifier identifier, int state) throws RemoteException
    {
        switch(identifier)
        {   case CLIENT:
                return clientLock[state].get();
            case PARENT:
                return parentLock[state].get();
            case LOCAL:
                return localLock[state].get();
        }
    }

    public boolean is(Identifier identifier, int index, int state) throws RemoteException
    {
        switch(identifier)
```

```
        {   case CHILD:
                return childLock.get(index)[state].get();
            case DOMAIN:
                return domainLock.get(index)[state].get();
            case PROXY:
                return proxyLock.get(index)[state].get();
        }
    }

    public boolean are(Identifier identifier, int state) throws RemoteException
    {
        switch(identifier)
        {   case CHILDREN:
                for(Lock[] lock : childLock.values())
                    if( !lock[state].get() )
                        return false;
                return true;
            case DOMAINS:
                for(Lock[] lock : domainLock.values())
                    if( !lock[state].get() )
                        return false;
                return true;
            case PROXIES:
                for(Lock[] lock : proxyLock.values())
                    if( !lock[state].get() )
                        return false;
                return true;
        }
    }

    public void waitUntil(Identifier identifier, int state, boolean value) throws RemoteException, InterruptedExceptio
    {
        switch(identifier)
        {   case CLIENT:
                clientLock[state].waitUntil(value);
                break;
            case LOCAL:
                localLock[state].waitUntil(value);
                break;
            case CHILDREN:
                for(Lock[] lock : childLock.values())
                    lock[state].waitUntil(value);
                break;
            case DOMAINS:
                for(Lock[] lock : domainLock.values())
                    lock[state].waitUntil(value);
                break;
            case PROXIES:
                for(Lock[] lock : proxyLock.values())
                    lock[state].waitUntil(value);
        }
    }

    public void waitUntil(Identifier identifier, int index, int state, boolean value) throws RemoteException, Interrup
    {
        switch(identifier)
        {   case CHILD:
                childLock.get(index)[state].waitUntil(value);
                break;
            case DOMAIN:
                domainLock.get(index)[state].waitUntil(value);
                break;
            case PROXY:
                proxyLock.get(index)[state].waitUntil(value);
        }
    }

}
```

The *NodeSync* class is an implementation of the *NodeSyncInterface* class.

Listing 4. NodeSyncInterface

```
public interface NodeSyncInterface extends Remote
{
    public static enum Identifier
    {   CLIENT,
        LOCAL,
        CHILDREN,
        CHILD,
        DOMAINS,
        DOMAIN,
        PROXIES,
        PROXY,
    }

    public final static int ONLINE = 0;
    public final static int CONNECTED = 1;
    public final static int INITIALIZED = 2;
    public final static int READY = 3;
    public final static int IDLE = 4;
    public final static int BUSY = 5;
    public final static int ON = 6;
    public final static int ACTIVE = 7;
    public final static int STARTED = 8;
    public final static int RUNNING = 9;
    public final static int Nstates = 10;

    public void set(Identifier identifier, int state, boolean value) throws RemoteException;
    public void set(Identifier identifier, int index, int state, boolean value) throws RemoteException;
    public boolean is(Identifier identifier, int state) throws RemoteException;
    public boolean is(Identifier identifier, int index, int state) throws RemoteException;
    public boolean are(Identifier identifier, int state) throws RemoteException;
    public void waitUntil(Identifier identifier, int state, boolean value) throws RemoteException, InterruptedExceptio
    public void waitUntil(Identifier identifier, int index, int state, boolean value) throws RemoteException, Interrup
}
```

The elegance of the *BooleanLock*/*NodeSync* combination is that it not only simplifies local thread coordination (reducing the chances of error-prone code), but it is easily extended to coordinate communication between remote threads via RMI. As can be seen from the NodeSync listings above, *NodeSyncInterface* extends java.rmi.Remote, making its implemented methods invokable from remote JVMs. Threads on one JVM can transparently set, query, and wait for thread states on an entirely different JVM. The *NodeSync* class is the basis for distributed-shared memory (DSM) computing in the current architecture.

Each Node in the network has its own local instance of *NodeSync* as well as a *NodeSyncInterface* proxy corresponding to the *NodeSync* implementation of its parent Node. As illustrated in the figure 7, the convention adopted in the framework is for a Node to communicate with its parent through a *NodeSyncInterface* proxy. A Node then communicates with its child through its local *NodeSync* instance.

BooleanLock is just one example of a concurrency construct for barrier synchronization. The following *IntergerLock* could also be used to synchronize multiple domains by incrementing an integer value as each domain obtains a lock and then decrementing the value upon release of the lock. This could result in a more efficient implementation.

Listing 5. IntegerLock

```
public class IntegerLock
{
    private int value;

    public synchronized int get()
    {   return value;
    }

    public synchronized void set(int value)
    {   this.value = value;
```

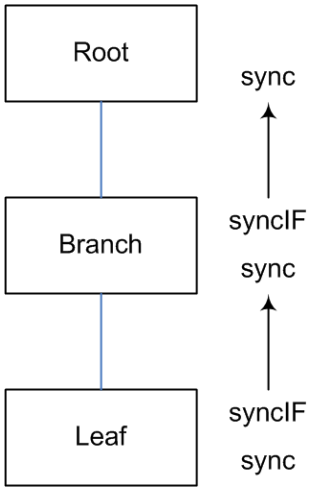


Figure 7. Synchronization convention. Synchronization calls between a child node and its parent are accomplished through the *Sync* implementation on the parent and the *Sync* interface of the child.

```
        notifyAll();
    }

    public synchronized void increment()
    {   value++;
        notifyAll();
    }

    public synchronized void decrement()
    {   value--;
        notifyAll();
    }

    public synchronized void waitUntil(int value) throws InterruptedException
    {   while( this.value != value )
            wait();
    }

}
```

To demonstrate the usage of the *NodeSync* class, we detail a computation cycle in the current framework for a given network configuration. The example considered is a two-tier network structure, consisting of a *Root* node, a *Leaf* node, and multiple *Domains* as illustrated in figure 8. The stages along with the corresponding methods invoked in *NodeSync*.

ROOT THREAD:

(i) wait until proxies have released their locks → *sync.waitUntil(PROXIES, RUNNING, false)*
(ii) get time step from proxies and compute global time step
(iii) lock each proxy → *sync.set(PROXIES, RUNNING, true)*

PROXY THREAD:

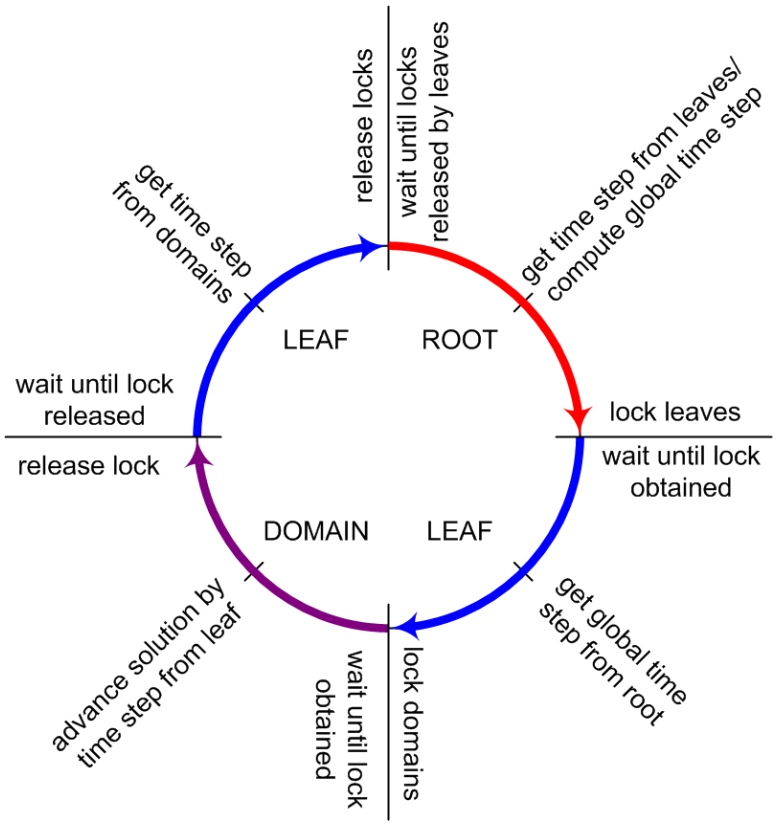


Figure 8. Global cylce. Interaction between *Root* and *Leaf* nodes and associated *Domains* during a global cycle.

(i) wait until lock is obtained → *sync.waitUntil(PROXY, 0, RUNNING, true)*
(ii) get global time step
(iii) lock child Node (Leaf) → *sync.set(CHILD, 0, RUNNING, true)*
(iv) wait until child has released its lock → *sync.waitUntil(CHILD, 0, RUNNING, true)*
(v) get time step from child
(vi) release lock → *sync.set(PROXY, 0, RUNNING, false)*

LEAF THREAD:

(i) wait until lock is obtained → *syncIF.waitUntil(CHILD, 0, RUNNING, true)*
(ii) get global time step
(iii) lock domains → *sync.set(DOMAINS, RUNNING, true)*
(iv) wait until domains release their locks → *waitUntil(DOMAINS, RUNNING, false)*

(v) get time step from domains
(vi) release lock → *syncIF.set(CHILD, 0, RUNNING, false)*

DOMAIN THREADS:

(i) wait until lock is obtained → *sync.waitUntil(DOMAIN, n, RUNNING, true)*
(ii) get global time step
(iii) fill ghost cells from target buffers
(iv) advance solution by time step and calculate new time step
(v) fill source buffers with boundary data and push to neighbor domains
(vi) release lock → *sync.set(DOMAIN, n, RUNNING, false)*

5. Results

The parallel efficiency of the architecture is now presented, measured in terms of its scalability. As previously mentioned, the software architecture is independent of the hardware architecture on which it is executed. More specifically, the software is optimized for both distributed-memory and shared-memory computer architectures. It is worth noting here that while there are many parallel software packages, most are either designed for distributed-memory or shared-memory architectures, while very few have been programmed for both. Furthermore, the software is not limited to one case or the other, but can execute as a combination of the two, while maintaining a maximum level of optimization.

To test the scalability, we solve the Euler equations of gas dynamics for a Rayleigh-Taylor instability problem with the finite-volume formulation,

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{1}{V}\sum_{s=1}^{N} \mathbf{F}_s A_s = \mathbf{S} \tag{1}$$

with

$$\mathbf{Q} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} \quad \mathbf{F}_n = \begin{pmatrix} \rho v_n \\ \rho u v_n + n_x P \\ \rho v v_n + n_y P \\ v_n (E + P) \end{pmatrix} \quad \mathbf{S} = \begin{pmatrix} 0 \\ 0 \\ \rho \\ \rho v \end{pmatrix} \tag{2}$$

where $\hat{n}$ is the outward unit normal of surface s and $v_n = \hat{n} \cdot \vec{v}$ is the normal velocity component.

Time integration is accomplished via the 2^{nd}-order accurate Adams-Bashforth (AB2) scheme

$$\mathbf{Q}^{n+1} = \mathbf{Q}^n + \tfrac{\Delta t}{2}[3L(\mathbf{Q}^n) - L(\mathbf{Q}^{n-1})] \tag{3}$$

where $L(\mathbf{Q})$ denotes the finite volume approximation and the source term. With AB2, the solution can be advanced in time with a single evaluation of $L(\mathbf{Q})$, and hence a single boundary exchange, per cycle. The fluxes are evaluated using 3rd-order accurate spatial interpolation of the conserved variables and is based on the monotonicity-perserving (MP) scheme of Suresh and Huynh [5]. The scheme is high-order and requires two layers of ghost cells per block for

communication. This increases the computational redundancy, as the information in the ghost cells are computed twice, thus reducing scalability performance.

Rayleigh-Taylor instability occurs at the interface between a heavy fluid which is accelerated into a lighter one. This problem has been simulated extensively in the literature, e.g. [1] and [6]. The problem is set up as follows: the solution domain spans the region $[0, \frac{1}{4}] \times [0, 1]$. Initially, the interface is at $y = \frac{1}{2}$, the heavy fluid with $\rho = 2$ is below the interface, and the light fluid with density $\rho = 1$ is above the interface with gravity acceleration in the positive y-direction. The pressure p is continuous across the interface, making the discontinuity a pure contact surface. The fluid speed is slightly perturbed in the y-direction. Thus, for $0 \leq y < \frac{1}{2}$, $\rho = 2$, $u = 0$, $p = 2y + 1$, $v = -0.025\cos(8\pi x)$; and for $\frac{1}{2} < y \leq 1$, $\rho = 1$, $u = 0$, $p = y + \frac{3}{2}$, $v = -0.025c\cos(8\pi x)$, where $c = \sqrt{\gamma p/\rho}$ is the sound speed, with the adiabatic exponent, $\gamma = 5/3$. Periodic boundary conditions are imposed for the left and right boundaries, while reflective boundary conditions are imposed for the top and bottom. The solution for the Rayleigh-Taylor problem as computed in the following sections is given in figure 9.

5.1. Shared-Memory Scalability

Of critical importance in evaluating the scalability of the code is determining how effective shared-memory hardware can handle multiple threads as well as associated overheads. To test this, several different domain decompositions are performed for a solution domain consisting of 1M cells, resulting in varying threadpools from 1 to 64 threads of equal workload. The code was executed on a single workstation with these various workloads and timed. Figure 10 highlights the results for the code executed on a workstation with dual AMD Opteron 265 processors running Open Solaris b77. With two processors, the code should execute most efficiently utilizing a domain decomposition of two domains. However, optimum performance is achieved with four threads, suggesting that multi-tasking of I/O, communication, and computational workload can be enhanced with a ratio of threads to processors/cores ¿ 1. Furthermore, when this ratio is increased beyond 2, performance drops off slowly, representing the overhead penalty incurred with too many threads.

5.2. Distributed-Memory Scalability

Distributed scalability of the code is evaluated using the same solution domain as in the shared-memory case, but is decomposed and distributed among the nodes of a cluster. The problem is solved with several different server configurations, each corresponding to a different number of server nodes, with two processors per node. Results for the various configurations are given in figure 11 for two different soultion domains consisting of 230,400 and 518,400 cells. The speed-up is almost linear up until 32 nodes (64 processors), signifying nearly ideal scalability. However, a drop off in speed-up to 80% efficiency beyond 32 nodes signifies bottlenecks in the framework, stemming from network latency for RMI calls and computational redundancy for ghost cells.

REFERENCES

Figure 9. Solution of Rayleigh-Taylor instability at $t = 2.5$ as computed with the current framework.

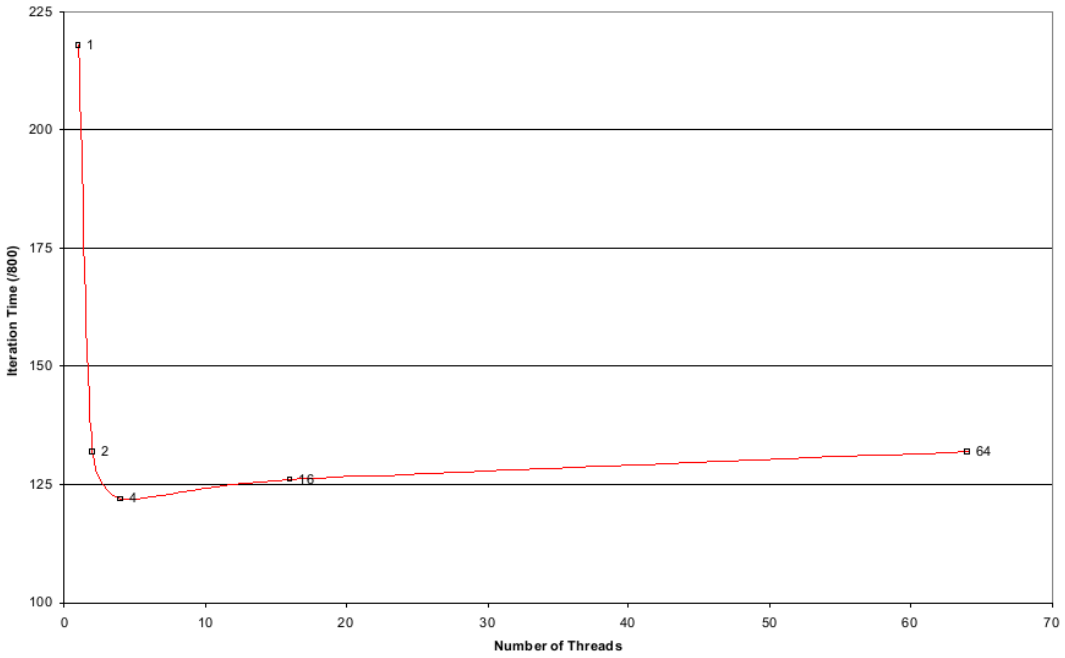


Figure 10. Shared-memory scalability. Computation time per 800 iterations versus number of threads for a Rayleigh-Taylor simulation with 1M cells executed on a single workstation. The computing platform consisted of a dual processor AMD Opteron 265 workstation running Open Solaris b77 with Java 6.

1. Glimm J, Grove J, Li X, Oh W, Tan DC. The dynamics of bubble growth for Rayleigh-Taylor unstable interfaces. *Physics of Fluids* **31**(1-2):447–465.
2. Grosso W. *Java RMI*. O'Reilly, 2002.
3. Hyde P. *Java Thread Programming*. Sams Publishing, 1999.
4. Markidis S, Lapenta G, VanderHeyden WB, Budimlic Z. Implementation and performance of a particle-in-cell code written in Java. *Concurrency—Practice and Experience* 2003; **17**(7-8):821–837.
5. Suresh A, Huynh HT. Accurate montonicity-preserving schemes with Runge-Kutta time stepping. *J. Comput. Phys.* 1997; **136**(1-2):83–99.
6. Young YN, Tufo H, Dubey A, Rosner R. On the miscible Rayleigh-Taylor instability: two and three dimensions. *Journal of Fluid Mechanics* **447**(3-7):377–408.

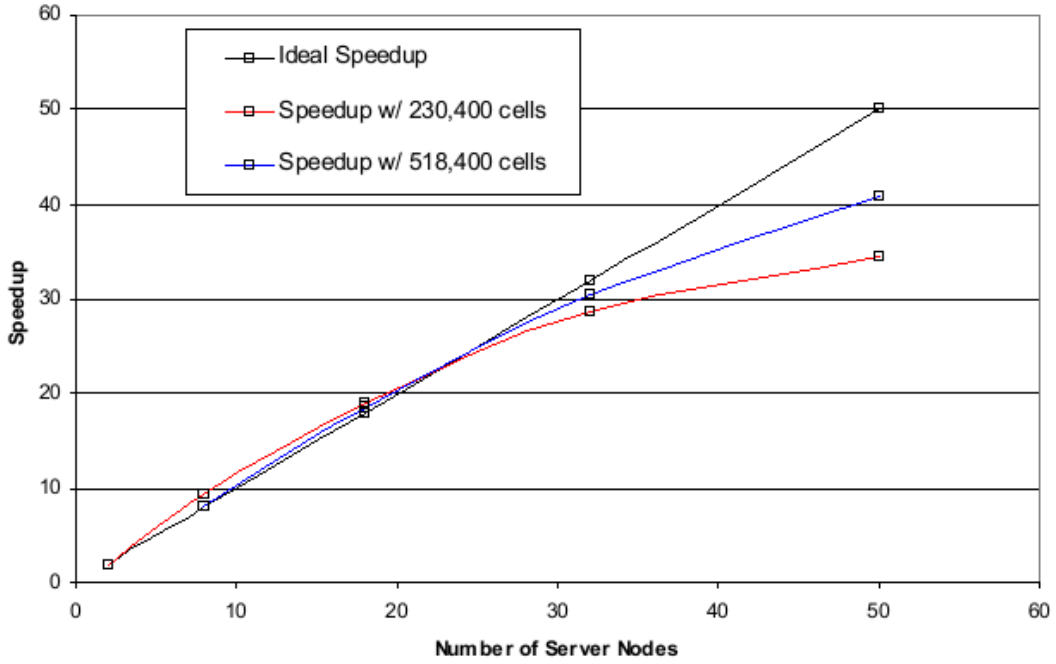


Figure 11. Distributed-memory scalability. Computation speedup as a function of number of server nodes for two different solution sizes. Speedup is nearly ideal up to 32 nodes but efficiency drops to just above 80% for the case with 518,400 cells and just below 75% for 230,400 cells.