Applying Idioms for Synchronization Mechanisms

Synchronizing communication components for an N Body Simulation

Jorge L. Ortega-Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM
jloa@ciencias.unam.mx

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '18, July 4–8, 2018, Irsee, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6387-7/18/07.
https://doi.org/10.1145/3282308.3282334

ABSTRACT

The Idioms for Synchronization Mechanisms is a collection of patterns related with the implementation of synchronization mechanisms for the communication components of parallel software systems. The selection of these idioms take as input information *(a)* the design pattern of the communication components to synchronize, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of communication required.

In this paper, it is presented the application of the Idioms for Synchronization Mechanisms to synchronize the communication components for an N Body Simulation, within the Detailed Design stage of the Pattern-based Parallel Software Design Method. In two previous papers, this method has been used in two previous stages: *(a)* in the Coordination Design stage, selecting the Manager-Workers architectural patterns as the coordination, which depends on the N- Body problem; and *(b)* in the Communication Design stage, selecting the Remote Rendezvous design pattern as communication, which depends on the memory organization of the parallel hardware platform, and on the architectural pattern previously selected.

CCS CONCEPTS

• **Software and its engineering** → **Cooperating communicating processes**; **Cooperating communicating processes**; **Object oriented architectures**;

KEYWORDS

Idioma, Synchronization Components, N-Body Simulation

ACM Reference Format:
Jorge L. Ortega-Arjona. 2018. Applying Idioms for Synchronization Mechanisms: Synchronizing communication components for an N Body Simulation. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3282308.3282334

1 INTRODUCTION

For the last fifty years, a lot of work and experience has been gathered in concurrent, parallel, and distributed programming around the synchronization mechanisms originally proposed during the late 1960s and 1970s by E.W. Dijkstra [4], C.A.R. Hoare [6–8], and P. Brinch-Hansen [1–3]. Further work and experience has been gathered today, such as the formalization of concepts and their representation in different programming languages.

In this paper, the objective is to show how the idioms that provide a pattern description of well-known synchronization mechanisms can be applied for the N-Body Simulation as a particular programming problem under development, applying the Pattern-based Parallel Software Design Method [11]. In two previous papers, the method has been used:

- in the Specification of the System stage, selecting the **Manager-Workers architectural patterns** as the coordination, which depends mostly on the N-Body problem [12], and;
- in the Specification of the Communication Components stage, selecting the **Remote Rendezvous design pattern** as the communication, which depends on the memory organization of the parallel hardware platform, and on the architectural pattern previously selected [13].

The description of synchronization mechanisms as idioms should aid software designers and engineers with a description of common programming structures used for synchronizing communication activities within a specific programming language, as well as providing guidelines on their use and application during the design and implementation stages of a parallel software system. This development of implementation structures constitutes the main objective of the Detailed Design step within the Pattern-based Parallel Software Design method [11].

RIGHTSLINK

2 BACKGROUND

2.1 Specification of the Systems: the Manager-Workers pattern

In the paper, *Applying Architectural Patterns for Parallel Programming. An N Body Simulation* [12], the Manager-Workers (MW) Architectural Pattern has been selected as a viable solution for the coordination within the parallel program that solves an N Body Simulation. In order to apply the Idioms for Synchronization Mechanisms (ISM), some information is required related to the MW Pattern, such as the parallel platform and programming language.

For this implementation, the parallel platform available for this parallel program is a cluster of computers, specifically, a dual-core server (Intel dual Xeon processors, 1 Gigabyte RAM, 80 Gigabytes HDD) 16 nodes (each with Intel Pentium IV processors, 512 Megabytes RAM, 40 Gigabytes HDD), which communicate through an Ethernet network. The parallel application for this platform is programmed using the Java programming language.

2.2 Specification of the Communication Components: the Remote Rendezvous pattern

In the paper *Applying Design Patterns for Communication Components. Communication between Manager and Worker components for an N-Body Simulation* [13], the Remote Rendezvous Design Pattern has been selected as a viable solution for the communication components of the MW pattern for solving the N Body Simulation. In order to apply the ISM, some information related with the Remote Rendezvous Pattern is required as well. This information is summarized as follows.

2.2.1 *The Remote Rendezvous pattern.*

The communication components are defined so they enable the exchange of `Body` type in a bidirectional, point-to-point, remote communication subsystem [13]. Hence, the Remote Rendezvous pattern has already been previously chosen as an adequate solution for such communications [9, 12, 13].

- **Description of the communication.** The Remote Rendezvous (RR) pattern provides a bidirectional, point-to-point, remote communication subsystem for the N Body simulation, which is based on the MW pattern. As an array of RR components, it is described as a set of communication components that disseminate requests and data to multiple Worker components executing on different processors or computer systems. Hence, a single RR pattern is replicated to distribute the whole set of bodies to be processed to the Worker components, executing on other memory systems. All Worker components execute simultaneously. However, they must communicate synchronously over the network of the distributed memory parallel system.
- **Structure and dynamics.** This section takes information of the RR design pattern, expressing the interaction between the software components that carry out the communication between parallel software components for the actual example.

(1) *Structure.* The structure of the RR pattern applied for designing and implementing communication components of the MW pattern is shown in Figure 1, using a UML Collaboration Diagram [5]. Notice that the communication component structure allows a synchronous, bidirectional communication between Manager and Worker components [10, 13].

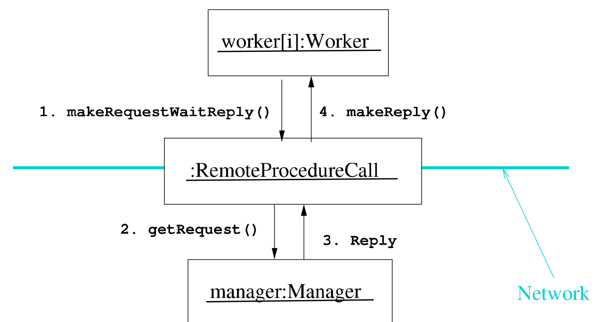

Figure 1: UML Collaboration Diagram of the Remote Rendezvous pattern used for synchronously exchange `Body` objects between Manager and Worker components of the MW solution to the N Body Simulation.

(2) *Dynamics.* This pattern actually performs a single remote calls within the available distributed memory parallel platform. Figure 2 shows the behavior of the participants of this pattern for the actual example [11, 13].
In this scenario, a single bi-directional, synchronous remote calls is carried out, as follows [13]:
- The worker requests data from the manager, so it issues a request operation to its remote procedure call component. This redirects the call to the manager through a socket, synchronizing the call so the worker remains blocked until it receives a response. If it made a read request for data, it waits until the data is made available: if it made a write request, the worker blocks until it receives an acknowledgement from the manager.
- The manager receives the request. If it is a request for data, it makes the data available by issuing a reply to the remote procedure call component (normally via a socket). On the other hand, if the request was for a write operation, the manager writes the partial result at the relevant place within the data structure and issues an acknowledgement message to the worker, enabling the it to request more work, if needed.

RIGHTSLINK Copyright Clearance Center

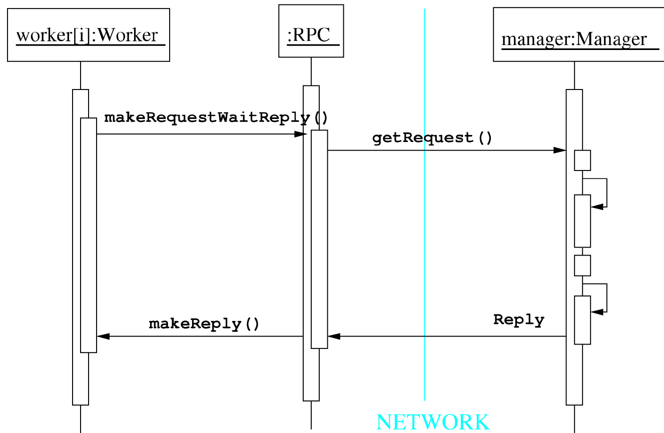

Figure 2: UML Sequence Diagram for the Remote Rendezvous pattern applied for exchanging `Body` objects between Manager and Wroker components of the MW solution for the N-Body Simulation.

(3) *Functional description of software components.* This section describes each software component of the Remote Rendezvous pattern as the participant of the communication sub-system, establishing its responsibilities, input, and output [13].

(a) **Worker.** The worker component has responsibility for requesting read operations of the set of bodies that act on a particular body, processing these information, and requesting write operations of the new position of such a body.

(b) **Manager.** The manager component has responsibility for maintaining the integrity and order of the local data structure of bodies, and serving read and write requests from the workers.

(c) **Remote procedure call.** The remote procedure call components in this pattern have two main responsibilities: (a) to serve as a remote communication and synchronization mechanism, allowing bidirectional synchronous communication between any two components on different computers that it connects, and (b) to serve as a remote communication stage for the distributed memory organization between the components, decoupling them so that communication between them is synchronous. Remote procedure calls are normally used for distributed memory environments.

3 DETAILED DESIGN

In the Detailed Design step [11], the software designer applies one or more idioms as the basis for synchronization mechanisms. From the decisions taken in the previous steps (Specification of the Problem [12], Specification of the System [12], and Specification of Communication Components [13]), the main objective now is to decide which synchronization mechanisms are to be used as part of the communication substructures.

3.1 Specification of the Synchronization Mechanism

– **The scope.** This section takes into consideration the basic previous information for solving the N Body Simluation. The objective is to look for the relevant information for applying a particular idiom as a synchronization mechanism.
For the N Body Simulation, the factors that now affect selection of synchronization mechanisms are as follows:
  * The available hardware platform is a cluster, this is, a distributed memory parallel platform, programmed using Java as the programming language.
  * The Manager Workers pattern is used as an architectural pattern, requiring to communicate layer software components [12].
  * The Remote Rendezvous design pattern is selected for the design and implementation of communication components to support synchronous communication between layers [13].

  Based on this information, the procedure for selecting an ISM for the N Body Simulation is as follows [11]:

(a) *Select the type of synchronization mechanism.* The RR pattern requires a synchronization mechanism that controls the access and exchange of `Body` values between a manager and a worker as software components that cooperate. These `Body` values are communicated using basically remote procedure calls. Hence, the idioms that describe this type of synchronization mechanism are the Message Passing idiom and the Remote Procedure Call idiom [11].

(b) *Confirm the type of synchronization mechanism.* The use of a distributed memory platform, given the previous design decisions, confirms that the synchronization mechanisms for communication components in this example may be message passing or remote procedure calls.

(c) *Select idioms for synchronization mechanisms.* Communication between layer components needs to be performed synchronously, that is, the manager component should wait for a response from its worker components. This is normally achieved using the RR pattern. Nevertheless, this design pattern requires synchronization mechanisms. In Java, the Remote Procedure Call idiom allows to develop a synchronization mechanism used here to show how implementation of the RR pattern can be achieved using this idiom.

RIGHTSLINK

(d) *Verify the selected idioms.* Checking the Context and Problem sections of the Remote Procedure Call idiom [11]:
   * Context: *'A parallel or distributed application is to be developed in which two or more software components execute simultaneously on a distributed memory platform. Specifically, two software components must communicate, synchronize and exchange data. Each software component must be able to recognize the procedures or functions in the remote address space of the other software component, which is accessed only through I/O operations.'*.
   * Problem: *'To allow communications between two parallel software components executing on different computers on a distributed memory parallel platform, it is necessary to provide synchronous access to calls between their address spaces for an arbitrary number of call and reply operations.'*.

   Comparing these sections with the synchronization requirements of the actual example, it seems clear that the Remoter Procedure Call idiom can be used as the synchronization mechanism for the communication. The use of a distributed memory platform implies the use of message passing or remote procedure calls, whereas the need for synchronous communication between manager and worker components points to the use of remote procedure calls.

The design of the parallel software system can now continue using the Solution section of the Remote Procedure Call idiom, directly implementing it in Java.

– *Structure and Dynamics.*

(a) **Structure.** The Remote Procedure Call Idiom is used for implementing the synchronization mechanisms of the communication components for the PL pattern. The Remote Procedure Call idiom in Java is presented as an interface, declaring some basic methods on which synchronization is achieved. Notice that the remote procedure call allows a synchronization over the two basic distributed components: a server and a client [11].

```
interface RemoteProcedureCallInterface{
    public abstract Object makeRequestWaitReply(Object m);
    public abstract Object getRequest();
    public abstract void makeReply();
}
```

(b) **Dynamics.** Remote Procedure Calls are used in several ways as synchronization mechanisms. Here, they are used for synchronous communication. The Remote Procedure Call idiom actually synchronizes the operation of the layer components over distributed memory. Figure 3 shows a UML Sequence diagram of the possible execution of the two participants of this idiom as the synchronization mechanism within the MRC pattern. Two parallel software components: a `client c` and a `server s`, which synchronize to exchange `int` values. Since they execute on different nodes of the distributed memory platform, they can only communicate using the remote procedure call methods.

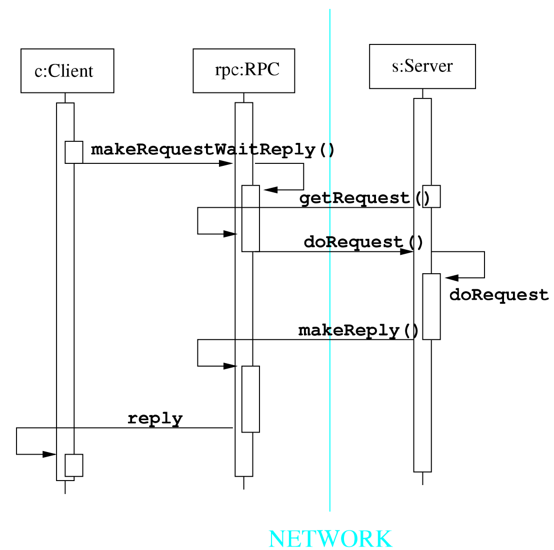

Figure 3: UML Sequence Diagram for the Remote Procedure Call idiom.

In this scenario, the synchronization over the remote procedure call is performed as follows:

* The communication between software components starts when the client `c` invokes `makeRequestWaitReply()`. Assuming that the `remote procedure call` component is free, it receives the call along with its arguments. The client `c` blocks, waiting until the `remote procedure call` component issues a `reply`.
* At the remote end, the server `s` invokes `getRequest()` to retrieve any requests issued to the `remote procedure call` component. This triggers the execution of a procedure within the server `s`, here `doRequest()`, which serves the call issued by the client `c`, operating on the actual parameters of the call.
* Once this procedure finishes, the server `s` invokes `makeReply()`, which encapsulates the `reply` and sends it to the `remote procedure` call component.
* Once the `remote procedure call` has the `reply`, it makes it available to the client `c`, which unblocks and continues. Note how the `remote`

RIGHTSLINK

`procedure call` acts as a synchronization mechanism between client and server.

– *Synchronization Analysis.* This section describes the advantages and disadvantages of the Remote Procedure Call idiom as a base for the synchronization code proposed [11].

(a) **Advantages**

∗ Multiple parallel worker components can be created in different address spaces of the computers that make up the cluster, as a distributed memory parallel platform. They are able to execute simultaneously, non-deterministically and at different relative speeds. All can execute independently, synchronizing to communicate.

∗ Synchronization is achieved by blocking the `client` until it receives a reply from the `server`. When implementing `remote procedure calls`, blocking is more manageable than non-blocking: `remote procedure call` implementations map well onto a blocking communication paradigm.

∗ Each worker component works its own address space. No other component interferes during communication.

∗ Data to be sorted is passed as arguments of the remote procedure calls. The integrity of arguments and results is maintained during all communication.

(b) **Liabilities**

∗ An implementation issue for remote procedure calls in this application example is the number of calls that can be in progress at any time from different threads within the manager component. It is important that a number of worker components on a computer within a distributed system should be able to execute simultaneously. Thus, it may be the case that the manager component is composed as a multi-threaded component, in order to receive responses and request work from a larger number of workers.

∗ It is commonly argued that the simple and efficient remote procedure call can be used as a basis for all distributed communication requirements of the present N Body Simulation. However, there are variations that can be applied here. Such variations include (a) a simple send for event notification, with no requirement for reply, (b) an asynchronous version of a remote procedure call that requests the server to perform the operation and keep the result so the client can picks it up later, (c) a stream protocol for different sources and destinations, such as terminals, I/O and so on.

4 IMPLEMENTATION

In this section, the communication components and their respective remote procedure call components are implemented as described in the Detailed Design step, using the Java programming language [12, 13]. So, the implementation is presented here for developing the RR as communication and synchronization components. Nevertheless, this design and implementation of the whole parallel software system goes beyond the actual purposes of the present paper.

4.1 Communication components – Remote Rendezvous

A class `RemoteProcedureCall` is used as the synchronization mechanism component of several components of the RR pattern. For example, let us consider the synchronization within the communication between a `Worker` and a `MultithreadManager`, using remote procedure calls [13].

```
class Worker implements Runnable {
    ...
    private RemoteProcedureCall rpc; // reference to rpc
    private Object data; // Data to be processed
    private Object result; // Result from the call
    ...
    public void run(){
       ...
       rpc = new RemoteProcedureCall(socket s);
       ...
       while(true){
            ...
            result = rpc.getRequest(data);
            ...
       }
    }
}
```

The `MultithreadManager` receives this remote call as follows:

```
class MultithreadManager implements Runnable {
   ...
   private RemoteProcedureCall rpc; // reference to rpc
   private Body data[]; // Data to be processed
   private Body subData[]; Data to be distributed
   private Body reply[]; // Results from worker threads
   private Body result[]; // Overall result
   private WorkerThread workerThread[];
   private int numWorkers;
   private Boolean request = false; // is there a request?
   ...

   //Function called by the rpc
   private void performRequest(Body d[]){
       data = d;
       synchronized(this){
          request = true;
          this.notify();
       }
   }
   ...
   public void run(){
```

RIGHTSLINK

```
        //Wait until someone make a request
        while(true){
           synchronized(this){
              while(!request){
                  try{wait();}
                  catch(InterruptedException e){}
              }
           }
         //Create worker threads
         for(int i=0;i<numWorkers;i++){
            subdata = getNextSubData(data,i);
            workerThread[i] = new workerThread(subData);
         }
         //Wait for all workers termination
         for(int i=0;i<numWorkers;i++){
            reply[i] = workerThread[i].returnResult();
            try{
               workerThread[i].join();
            }
            catch(InterruptedException e){}
         }
         result = gatherReplies();
         rpc.makeReply(result);
       }
   }
   ...
}
```

Notice the way both components rely on a remote procedure call component to exchange and distribute `Body` values as data and results of the computation. Hence, the successful operation of the communication structure relies on how the remote procedure call component implements the methods of the interface `RemoteProcedureCallInterface`: `makeRequestWaitReply()`, `getRequest()`, and `makeReply()`. This is shown in the following section.

4.2 Synchronization Mechanism – Remote Procedure Calls in Java

Based on the Remote Procedure Call idiom and their implementation in the Java programming language, the basic synchronization mechanism that controls the communication between the `Worker` components and the `MultithreadedManager` is presented as follows:

```
import java.net.*;
...

class RemoteProcedureCall extends UnicastRemoteObject
           implements RemoteProcedureCallInterface {

   protected Object data;
   protected Object reply;
   private MultithreadedManager mm;
   ...
   private MessagePassing in = null;
   private MessagePassing out = null;
   ...
   public RemoteProcedureCall(Socket socket) {
      ...
      this.in = new ObjPipedMessagePassing(socket);
      this.out = this.in;
   }
   public Object clientMakeRequestAwaitReply(Object m) {
      send(in, m);
      return receive(out);
   }
   public Object serverGetRequest() {
      return receive(in);
   }
   public void serverMakeReply(Object m) {
      send(out, m);
   }
   ...
}
```

The class `RemoteProcedureCall` implements a two-way flow of information based on sockets, as a one-way flow of information between message passing sender and receiver. The `MultithreadedManager` component sends an object to a `Worker` that represents a request, creates a thread to manage it, which blocks waiting for the `Worker`'s reply. On its side, each `Worker` blocks waiting for a request. When it gets such a request, computes the reply, and sends it to the `MultithreadedManager` component, unblocking its associated thread. As described in the RR pattern, the `MultithreadedManager` spawns off threads to handle the requests, thus being able to concurrently manage several requests.

Notice that the `MultithreadedManager` acts as a call distributor: it waits for requests from the `Worker` components that are able to do some work. The `MultithreadedManager` sends a work command to each `Worker` components, which sends a result back later in another call. Notice that this part of the functionality of the RR pattern is not shown in this code.

The Remote Procedure Calls here are based on synchronous message passing rather than asynchronous because buffering is unnecessary and would waste space; any client blocks on the send in synchronous case and, on the receive in asynchronous case. Hence, there is no need of synchronized methods, because synchronization is handled inside the send and receive methods. This method should be synchronized if there are multiple client threads sharing this object.

Finally, it is important to notice that a deadlock possibility exists: if the server makes another call to `serverGetRequest()` before calling `serverMakeReply()` then this `RemoteProcedureCall` object is deadlocked (assuming just one client is using this object, the intended situation) in the sense that the client is blocked on `receive(out)` and the server is blocked on `receive(in)`. This still needs to be fixed for the present implementation.

5 SUMMARY

The ISM are applied here along with a method. In order to show how to apply an idiom that copes with the requirements of the communication components present in the Manager Workers solution to the N Body Simulation. The main objective of this paper is to demonstrate, with a particular

RIGHTSLINK

example, the detailed design and implementation that may be guided by a selected idiom. Moreover, the application of the ISM and the method for selecting them is proposed to be used during the Detailed Design and Implementation for other similar problems that involve synchronous distribution of data, executing on a distributed memory parallel platform.

6 ACKOWLEDGEMENTS

The author would like to thank Victor Sauermann, my shepherd for EuroPLoP 2018, for his valuable comments, as well as the attendants to the Writers' Workshop group, for their greatly helpful suggestions.

REFERENCES

[1] P. Brinch-Hansen, *Structured Multiprogramming.* Communications of the ACM, Vol. 15, No. 17. July, 1972.
[2] P. Brinch-Hansen, *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering, Vol. 1, No. 2. June, 1975.
[3] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.
[4] E.W. Dijkstra *Co-operating Sequential Processes*, In Programming Languages (ed. Genuys), pp.43-112, Academic Press, 1968.
[5] M. Fowler, *UML Distilled.* Addison-Wesley Longman Inc., 1997.
[6] C.A.R. Hoare, *Towards a theory of parallel programming.* Operating System Techniques, Academic Press, 1972.
[7] C.A.R Hoare, *Monitors: An Operating System Structuring Concept.* Communications of the ACM, Vol. 17, No. 10. October, 1974.
[8] C.A.R. Hoare *Communicating Sequential Processes.* Communications of the ACM, Vol.21, No. 8, August 1978.
[9] J.L. Ortega-Arjona *The Manager-Workers Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming.*, Proceedings of the 9th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2004), Kloster Irsee, Germany, 2004.
[10] J.L. Ortega-Arjona *Design Patterns for Communication Components*, Proceedings of the 12th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2007), Kloster Irsee, Germany, 2007.
[11] J.L. Ortega-Arjona *Patterns for Parallel Software Design.* John Wiley & Sons, 2010.
[12] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. An N Body Simulation*, Proceedings of the 2nd Asian Conference on Pattern Languages of Programs (AsianPLoP2011), Tokyo, Japan, 2011.
[13] J.L. Ortega-Arjona *Applying Design Patterns for Communication Components. Communication between Manager and Worker components for an N-Body Simulation.*, Proceedings of the Workshop on Parallel Programming Patterns (ParaPLoP2011), Carefree, Arizona, USA, 2011.

RIGHTSLINK
Copyright Clearance Center