The Design and Implementation of Partial Quiescence in a Concurrent Programming Language

By

BILLY YAN-KIT MAN
B.S. (University of California, Davis) 2003

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Ronald A. Olsson, Chair

Professor Aaron W. Keen

Professor Bertram Ludaescher

Committee in charge

2006

Contents

List of Figures

List of Tables

Acknowledgments

My educational journey to a Master's degree would not have been successful without the help and support of many people. I would like to take this opportunity to express my gratitude for them. First, I must thank my advisor, Professor Ronald Olsson, for his helpful guidance and advice in completing this thesis. The initial idea of partial quiescence that he came up with was interesting and motivated my interest in performing research in the area. The suggestions and idea he provided to me throughout my research career were always brilliant. I appreciate his patience in revising countless drafts of my thesis, his quick responses to my questions and requests, his tolerance of my unusual research hours, and his understanding of my busy grading schedule as a reader. I would also like to thank him for his efforts in writing and putting together a paper that we have submitted for publication based on this thesis.

Second, I would like to acknowledge my committee members, Professor Aaron Keen and Professor Bertram Ludaescher. I thank Professor Keen for his willingness to be part of my committee even though he had to remotely communicate with me from Cal Poly University, San Luis Obispo. I thank Professor Ludaescher for his immediate acceptance of my request that he be my committee member when I had only taken one of his classes. I am also grateful for their comments in making this thesis a better piece of writing and their promptness in meeting the thesis filing deadline despite the fact that they have very busy schedules.

Special thanks should also go to the members of the JR research group who have contributed to giving me valuable suggestions on the design of my work, performing testings on the early versions of the implementation, and porting it to the latest JR version. I would also like to especially thank Angela Chan for introducing me to the research opportunity with Professor Olsson and answering the numerous questions I have had on JR, which has helped to make my research experience a smoother one.

My family has given me invaluable encouragement for keeping up with my challenging educational career. They taught me the importance of educational values and urged me not to give up whenever I encountered any obstacles. I deeply appreciate their unconditional love and everlasting support, both emotionally and financially.

My life would be much more dull without the laughter I share with my friends. I thank them for sharing in all the ups and downs in my life. All the wonderful time we have spent together will surely remain in my memory.

Chapter 1

Introduction

In concurrent and distributed computing, multiple processes often cooperate to perform some task. With the development of multiprocessor architectures, program performance can be improved as processes can be distributed among different processors and executed in a parallel fashion. A major feature of distributed programs is that processes are allowed to communicate with each other either synchronously or asynchronously via messages to exchange information. This leads to an important, and well-studied, problem to determine when all processes and their interactions have terminated, or become *quiescent* in a system. Such a detection problem is challenging because each process has only local information, but to solve the problem requires the global state of the system to provide information about all processes. More formally, global quiescence is defined as the state in which each process has terminated or deadlocked, and there are no messages in the communication channels [15]. Quiescence detection is then the mechanism used to detect such a state in a distributed system.

Global quiescence detection exists as a built-in feature in some programming languages and systems. This feature is beneficial to programmers as it can save them additional efforts to write synchronization code to detect quiescence. As the detection becomes automatic, it can also provide programmers the convenience to perform various actions such as simply terminating the program, outputting a final result, gathering statistics from the overall computation, or initiating a new phase of the program, which might involve another new, corresponding phase of quiescence detection. Although useful, global quiescence detection is limited to detecting the state of *all* processes in a system. A more powerful detection mechanism would detect when a specified part of the program has become quiescent. In other words, since multiple processes exist to perform some computation, it is sometimes useful to know when certain components of a program have finished their tasks. This motivates us to define *partial quiescence* and to develop a detection mechanism for it.

The general idea of partial quiescence is to allow a user to put related processes of a program into a group. More than one group can be created for a single concurrent program. When all processes in a particular group have terminated or deadlocked, that group is considered to have become partially quiescent. Once partial quiescence is detected, the program has an opportunity to perform the various actions mentioned earlier that are available for global quiescence.

In this thesis, we propose possible ways of defining partial quiescence and explore the accuracy and implementation aspects of each. We then discuss the definition selected and describe the implementation of the algorithm used for an object-oriented concurrent programming language called JR, which extends Java. In JR, the feature of global quiescence detection already exists to provide the functionalities as mentioned earlier. We also discuss the performance when global quiescence detection is used to terminate the program instead of normal program termination. Moreover, we compare the performance of using partial quiescence detection and global quiescence detection.

The rest of this thesis is organized as follows. Chapter 2 provides the background on the general definition of termination detection and discusses previous work. It also gives a brief overview of the JR programming language and a discussion of

Java's thread and thread group. Chapter 3 describes the global quiescence detection mechanism presently implemented in JR and presents some examples of JR programs. Chapter 4 discusses the different ways of defining partial quiescence, introduces some expository partial quiescence examples in JR, presents some key aspects of partial quiescence, and describes the implementation of partial quiescence. Chapter 5 presents some performance results. Finally, Chapter 6 concludes this thesis. A summary of this work appears in [14].

Chapter 2

Background

2.1 Distributed Termination Detection

Termination detection is an important problem in the field of distributed and parallel computing. It was first introduced separately by Francez [6] and Dijkstra and Scholten [5] in 1980 [15]. Distributed termination detection (DTD) is identified as a difficult problem because it is concerned with the global state of the system. Because each process only contains a local view, but not an up-to-date view of the global state of the distributed system, it is nontrivial to efficiently determine whether the system has come to a complete termination [15][20][22]. The problem of termination detection has been well-studied and an extensive amount of work has been done on it. We shall present the definition of the problem in greater detail and briefly discuss some previous research performed.

2.1.1 Problem Description

More formally, DTD can be described as follows [15][16][19]. A distributed system consists of a collection of processes such that processes communicate with each other by sending *activation messages* via some communication channels. An activation message is used not only for communication purpose among processes, but also for creation of a new process. A process can be either in an active or a passive state. A process is said to be *active* if it is working on some computation or processing activation messages addressed to it. A *passive* process is one that is waiting for an activation message or termination. All processes in the system behave based on the following rules [15][16][19]:

1. Activation messages can be generated only by active processes.

2. A passive process may change its state to active only if it receives an activation message.

3. An active process may change its state to passive at any time.

The above rules ensure that no further activation messages can be created in a system where all processes are passive: messages cannot be generated spontaneously. When the system has reached a state such that all processes are passive and no activation messages are in transit, then the system is *quiescent*. Quiescence detection is defined as the mechanism used to detect a quiescent state. This definition generalizes that of DTD to both detecting termination as well as deadlock. The detections of termination and deadlock are similar in that both deal with sensing when the system is in a state that can no longer continue. Note that the quiescence detection algorithm is a separate activity that runs concurrently but without interfering with the underlying computation of the system. Thus, in addition to activation messages, *control messages* are used for system related activities such as detecting quiescence. All active and passive processes can participate in the detection activity via control messages while keeping their active or passive status unchanged.

2.1.2 The Dijkstra-Scholten Algorithm

One of the several categories of DTD algorithms, as classified in [15], is network topology. In network topology algorithms, it is necessary to make use of the system's topology for detecting quiescence. An example of this category is the termination detection algorithm for "diffusing computations" proposed by Dijkstra and Scholten [5]. Initially in the quiescent state, the system contains a source process (the root), which starts the underlying computation by sending (diffusing) activation messages to generate new processes. The new processes then keep track of which source activated them and continue with the protocol. When receiving a message, the activated process sends an acknowledgment message to the sending process. Each process also maintains a *deficit counter* to keep track of the number of its own sent messages that have not been acknowledged. When the deficit counter becomes zero, i.e., there are no more unacknowledged messages, and the process is locally quiescent such that it is no longer processing any other activation messages, then the process transforms back to the passive state and sends an acknowledgment to its activator.

A tree model is formed based on the processes in the system, where the parent is the process that sends an activation message to generate a child process. The tree grows as activation messages are sent to create new processes and is pruned as leaf processes becomes locally quiescent. At the point when only the root of the tree remains, the "diffusion" is done and the quiescence detection process terminates [5][15][22].

2.1.3 The Credit Distribution and Recovery Algorithm

Another interesting termination detection method for distributed systems is the credit distribution and recovery scheme presented by Mattern in [16]. The system begins with a credit of total value one, and the termination detection process distributes this credit amongst active processes and activation messages. When a process generates an activation message, it splits its credit equally between itself and the message. If the activation message is sent to a passive process, then the credit of the message is transferred to the process. Otherwise, if the activation message is sent to an active process, the credit goes to the quiescence detection process. The credit of a process is returned to the detection process only when it becomes passive. When the detection process regains the whole credit of one, the system reports quiescence [15][16].

2.2 Tools and Systems with Termination Detection Support

In some programming languages and systems, programs wait indefinitely when deadlock states are encountered. For instances, Java, Ada, MPI, and Pthreads programs are unable to detect deadlock situations automatically. However, there are some tools developed to assist such detection in these programs. There are also other programming languages such as SR, JR, and Charm that have deadlock detection algorithms implemented as a built-in feature so that the quiescence detection process is automated.

2.2.1 MPI-CHECK 2.0

Debugging MPI programs can be difficult and time-consuming. MPI-CHECK [13] is a tool designed to assist debugging MPI programs written in free or fixed format Fortran 90 and Fortran 77. MPI-CHECK version 2.0 [13] provides the additional feature that detects many actual and potential deadlocks that may occur when using collective, blocking, and non-blocking point-to-point routines. MPI-CHECK 2.0 uses a decentralized handshaking approach for deadlock detection.

2.2.2 Umpire

Umpire [21], developed at Lawrence Livermore National Laboratory, is another tool used to detect and analyze programming errors in MPI applications. Detecting deadlocks is one goal of the tool. Umpire monitors message passing operations of a program using a profiling layer that exists between the MPI application and runtime system. The central manager then runs a verification algorithm on MPI call information that is gathered from communicating with the shared memory buffer on the profiling layer and controls the execution of the MPI application.

2.2.3 GARLIC

GARLIC [11], which stands for "generic Ada reusable library for interpartition communication", is a high-level communication tool that provides Ada 95 with distributed programming features within the GNAT system. It supports features for deadlock and termination detection based on the algorithm proposed by J.M. Helary et al. [7]. GARLIC examines messages to monitor the stability and passivity of the distributed system in order to detect deadlock occurrence or the termination of an application [11].

2.2.4 SR

SR (Synchronizing Resource) [1] is a programming language that supports mechanisms for concurrency, communication, and synchronization. A resource in SR is similar to a module found in other programming languages in that it represents a template from which instances can be created dynamically. In a SR resource, programmers are allowed to optionally write final code that will be executed when a resource instance is destroyed. Before the resource instance disappears, final code can be used to clean up structures created dynamically. Moreover, when SR detects a deadlock state or termination of a program, the main SR resource will be destroyed and the final code that associates with it can be used to conduct activities such as outputting final result or collecting statistics from the overall computation. The program simply terminates if no final code is specified.

2.2.5 JR

JR [18] is a programming language that extends Java and provides a concurrency model based on that of SR. It supports an automatic termination detection mechanism (which is not available in Java) that enables programmers to optionally register an operation when the program is deadlocked or terminated. When the program is quiescent, if no operation is specified, the program simply terminates. Otherwise, the operation, known as a *quiescence operation*, is invoked and can perform various actions as those in the final code of SR (mentioned in Section 2.2.4). Additionally, JR allows a quiescent program to initiate a new phase of computation within a quiescence operation, for which the quiescence feature can be used again. Further details on JR and its global quiescence detection appear in Section 2.3 and Chapter 3 respectively.

2.2.6 Charm

Sinha, Kalé, and Ramkumar describe in [19] a dynamic and adaptive quiescence detection algorithm that is implemented for the machine independent parallel programming system called Charm. In Charm, when a deadlock situation or termination is detected, programmers can perform a variety of actions including collecting statistics of the user computation, initiating a new phase of activity, or simply terminating the computation [19].

2.3 Overview of the JR Programming Language

This section gives a brief overview of JR, the language in which we provide linguistic support for partial quiescence (see Chapter 4).

JR [18] is a concurrent programming language that extends Java and provides users the ability to use Java's features and program in an object-oriented manner. Furthermore, it possesses a flexible concurrent scheme to perform synchronization based on that of SR [1], and allows users to make use of its expressive power to program concurrent mechanisms. For example, some of the features that make JR a rich concurrent model are dynamic remote virtual machine creation, dynamic remote object creation, and process interaction mechanisms such as remote procedure call, rendezvous, dynamic process creation, and asynchronous message passing. More specifically, the mechanism to create dynamic remote virtual machines is what makes the JR model a truly distributed system, as a JR program is allowed to create remote objects and processes across different units of program distribution. JR also provides a means to deal with program quiescence. Section 3.1 discusses further details on global quiescence detection in JR.

2.4 Java Multithreaded Environment

Concurrent execution of threads is a common expression of parallelism. In JR, concurrent programming is performed using multiple processes, such that each process represents an independent thread of control and executes sequential code [18]. JR processes are created using Java threads, and JR process groups, used for partial quiescence, as we will discuss further in Chapter 4, are created using Java thread groups. This section provides an overview of Java threads and thread groups.

2.4.1 Java Thread

A Java thread represents a single flow of control within a program. Multiple threads are allowed to execute concurrently under a Java Virtual Machine. There are two ways to create a Java thread. One way is to create an object of a class that extends `java.lang.Thread` and define the sequential code of the thread in a `run()` method. When the thread is ready to begin execution, the object invokes the `start()` method of its superclass. Another way to create a Java thread is by creating an object of a class that implements the `java.lang.Runnable` interface and define the sequential code of the thread in a `run()` method. To begin the thread's execution, a `java.lang.Thread` object needs to be instantiated and the `Runnable` object needs to be passed as an argument to the constructor.

Java provides supports for thread scheduling and synchronization. Each thread has a priority level associated with it as to reflect the importance of the task it is performing. Generally, a thread with higher priority is given preference to execute before a lower-priority thread. However, at some other times, a lower-priority thread might execute first to avoid starvation, or when voluntary rescheduling mechanisms, such as the `sleep()` and `yield()` methods (defined in `java.lang.Thread`), are used [2].

To prevent simultaneous access to shared data, Java allows synchronizing access of threads to critical regions by using `synchronized` methods or `synchronized` statements. Either way involves acquiring a *lock* associated with an object before a thread can start the execution of the critical section. Once the lock is acquired, no other threads can acquire it until the current thread completes its execution and releases the lock. Another way to deal with thread synchronization is through the use of Java's monitor-like methods `wait()`, `notify()`, and `notifyAll()`. A `wait()` method causes one thread to wait until an optionally specified `timeout` period is reached, or until another thread invokes the `notify()` method (awakens one of waiting thread, picked arbitrarily) or the `notifyAll()` method (awakens all waiting threads).

2.4.2 Java Thread Group

Java allows threads to be organized into thread groups for management and security purposes. When putting related threads into a group, it becomes easier to manage them as a unit and place limitations on what threads in different groups can do [2]. For example, the threads in a group can all be interrupted at once. Usually, threads in the same group are related in some way, such as their creator and the type of functions they perform.

Each Java thread belongs to a thread group, represented by a `java.lang.ThreadGroup` object. When instantiating a `java.lang.Thread` object, Java programmers have an option to specify its `ThreadGroup` object as an argument to the `java.lang.Thread` constructor. If a thread group is not specified, by default, each new thread is created in the same thread group as that of the thread that created it. Once a thread is created, it belongs to its thread group permanently, i.e., the thread cannot join a new group after its creation. Usually, in a Java Virtual Machine, there is an initial non-daemon thread that begins execution and starts up other threads. (This thread typically executes the `main` method of a Java application.) As this thread is not created by the Java programmer, there is no option to choose which thread group this thread belongs to. By default, this initial thread belongs to the "main" group. A thread group can contain threads as well as other thread groups, providing a hierarchical tree structure, as illustrated in Figure 2.1 (from [8]). Specifically, every thread group in the tree structure has a parent thread group, except the root node that contains the default thread group "main" [8].

Some of the useful functionalities that Java thread groups provide include [9]:

- finding the number of active threads in the group

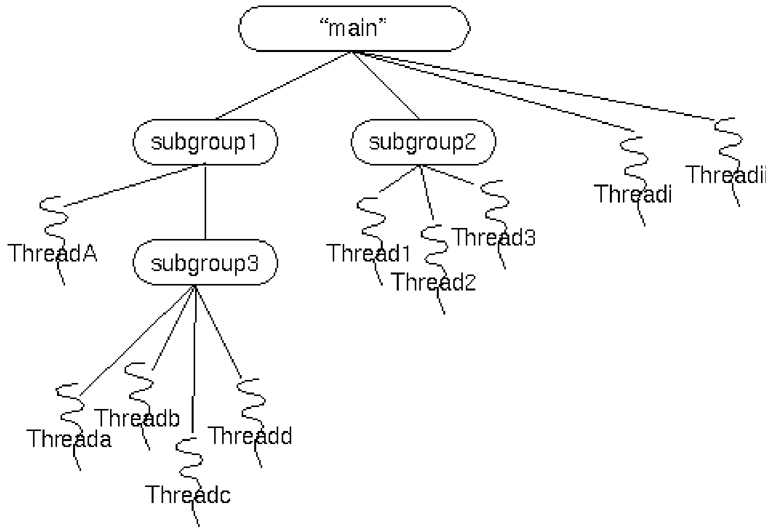

Figure 2.1: The thread group hierarchical structure.

- obtaining a list of references of the active threads or subgroups within the group
- retrieving and setting maximum priority for all threads in the group
- restricting the current thread to access or modify information in the group (through cooperating with the Java security manager)

Chapter 3

Global Quiescence (GQ)

3.1 Global Quiescence in Existing JR

Automatic termination detection [18] is a useful feature of JR in which programmers do not need to write special synchronization code to detect quiescence. Instead, they can focus on writing application code. A JR program is identified as *globally quiescent* when all of its processes have terminated or deadlocked. When a JR program reaches its quiescent state, normally, the execution of the JR program will be terminated. However, as noted in Section 2.2.5, JR programmers have an option to register an operation as the *quiescence operation*. This operation will be invoked[1] once the quiescent state has been detected; then the code associated with the quiescence operation will be executed. After its execution, the quiescence operation is set to the unregistered state and the JR program will terminate. However, note that as it is allowed to initiate new activities in the quiescence operation, a JR program will only terminate when these newly generated activities have also quiesced. If the programmer re-registers an operation (either the same or different one) as the quiescence operation within the current quiescence operation, the new quiescence operation will

[1] The operation is to be invoked implicitly via asynchronous send.

be invoked when these newly generated activities have quiesced. If no quiescence operation is ever registered, the JR program will simply terminate.

3.2 Examples of JR Programs using GQ

This section illustrates and discusses some examples of global quiescence. Note that JR example code and JR keywords are typeset in the `Typewriter` typeface in order to distinguish them from plain text used in this thesis.

3.2.1 Matrix Multiplication

As the inner products of a matrix are independent of each other, they can be computed in parallel. In the cases when the number of inner products is sufficiently large, parallel execution produces a more efficient program than running in a sequential mode. The program in Figure 3.1 (from [18]) performs matrix multiplication in parallel. It multiplies two $\mathtt{N} \times \mathtt{N}$ real matrices using processes and outputs the result when the program has become quiescent.

The `MMMain` class contains the `main` method, which simply reads in two $\mathtt{N} \times \mathtt{N}$ matrices, instantiates a `MMMultiplier` object, and registers the operation `done` as the quiescence operation. It also contains the declaration and definition of the operation `done`. The `MMMultiplier` class contains the processes that perform the actual computation: each process computes a corresponding inner product entry for the resulting matrix. These processes begin execution right after `MMMultiplier`'s constructor completes its execution. The inner product for each entry is stored in `C[r][c]`. The `print` method of the `MMMultiplier` class simply prints the resulting matrix.

Since parallel processes are used, global quiescence is needed to determine when they have all finished their tasks. Once it has been detected, the registered operation

```
public class MMMain {
  private static MMMultiplier m;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    m.print();
  }
}

public class MMMultiplier {
  int N; // A, B, and C are NxN
  double [][] A, B, C;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    // compute the inner product for C[r,c]
    C[r][c] = 0.0;
    for (int k = 0; k < N; k++) {
      C[r][c]+= A[r][k] * B[k][c];
    }
  }
  public void print() {
    // output C
    ...
  }
}
```

Figure 3.1: Matrix multiplication using global quiescence.

`done` will be invoked to output the resulting matrix. Hence, automatic termination detection assists programmer to avoid writing separate code to detect quiescence [18].

3.2.2 Matrix Multiplication with Re-registration of GQ Operation

Global quiescence is not only useful to detect when the whole program has completed execution. It can also be used to delay the initiation of a new phase of new activities until the current phase has completed. This can be done via re-registering the quiescence operation as mentioned in Section 3.1.

```
public class MMMain {
  private static MMMultiplier m;
  //number of times to re-register the quiescence operation
  private static int K;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    K = 10;
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
     if(K >= 1)  {
        K--;
        m.print();
        //read in new NxN arrays A and B
        ...
        m = new MMMultiplier(A, B, N);
        // re-register done as the quiescence operation
        try {
           JR.registerQuiescenceAction(done);
        } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
          e.printStackTrace();
        }
     }
  }
}
```

Figure 3.2: Multiple rounds of matrix multiplication using global quiescence – `MMMain` class.

Figure 3.2 revisits the matrix multiplication example in Section 3.2.1. It extends

Figure 3.1 to allow multiple rounds of matrix multiplication by re-registering the quiescence operation `done`. The `MMMultiplier` class remains the same as in Figure 3.1. The only changes appear in the `MMMain` class. Note that `done` is registered once in the main method and will be invoked when all `compute` processes for the object `m` complete execution, since there are no other processes in the program. When invoked, `done` prints the result, gets new input for arrays `A` and `B`, and re-registers itself for the next round of computation. This activity repeats until `K` becomes zero, at which point the quiescence operation is no longer re-registered, and thus the program terminates. By using global quiescence, we can synchronize the sequence of printing the result for each round, whereas if all rounds of computation are started at the same time and run concurrently, the outputs of their resulting matrices might be interleaved.

```
public class MMMain {
  private static MMMultiplier m;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    int K = 10; //number of for-loop iterations
    for(int i=0; i<K; i++)  {
       // read in NxN arrays A and B
       ...
       m = new MMMultiplier(A, B, N);
       // register done as the quiescence operation
       try {
         JR.registerQuiescenceAction(done);
       } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
         e.printStackTrace();
       }
    }
  }
  private static op void done() {
        m.print();
  }
}
```

Figure 3.3: Bad attempt at multiple rounds of matrix multiplication – `MMMain` class.

To obtain the effect of delaying the initiation of a new activity until the current one has completed, it is necessary to re-register a quiescence operation within a quiescence operation. Figure 3.3 shows the different effect that will result when the re-registration of the quiescence operation is done in the `main` method using a `for` loop instead. The `MMMultiplier` class remains the same as in Figure 3.1. Even though the quiescence operation is re-registered, it will only be invoked once during the entire execution of this program, because all re-registrations happen before the program is ever quiescent. Note that re-registering the quiescence operation in this manner just continuously updates the operation to be invoked when global quiescence is reached. Hence, when all instances of `m` have completed their computations, the `done` operation is invoked once and only the result from the last instance of `m` created is printed. As only the most recent quiescence operation registered in the `for` loop is taken into account, this program illustrates a bad attempt to use global quiescence as a barrier point to delay the initiation of new activities.

Figure 3.4 and 3.5 further revisit the matrix multiplication example in Figure 3.2. They also allow multiple rounds of matrix multiplication by re-registering the quiescence operation `done`. However, the only difference in this program is that the result of one round of computation depends on the previous round. When the quiescence operation `done` is invoked, in addition to printing the resulting matrix from the current round of computation, the program uses it as the first input matrix `A` for the next round by calling the newly added method `getResult()` from the `MMMultiplier` class in Figure 3.5. It then reads in the second input matrix `B` and re-registers the `done` operation so that after the next round of computation is completed, the `done` operation will be invoked again. This sequence of operations repeats for `K` times. By re-registering a quiescence operation, we can see the benefit of using global quiescence to begin an iteration to compute the product of two matrices after the result from the previous iteration is ready to be used.

```
public class MMMain {
  private static MMMultiplier m;
  //number of times to re-register the quiescence operation
  private static int K;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    K = 10;
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
     if(K >= 1) {
        K--;
        m.print();
        //get result to use as an input for the next round
        A = m.getResult();
        //read in NxN array B
        ...
        m = new MMMultiplier(A, B, N);
        // re-register done as the quiescence operation
        try {
           JR.registerQuiescenceAction(done);
        } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
          e.printStackTrace();
        }
     }
  }
}
```

Figure 3.4: Multiple rounds of matrix multiplication with dependent results between iterations – `MMMain` class.

3.2.3 Distributed Matrix Multiplication and Addition

JR also provides support for detecting global quiescence in a distributed application. The main program in Figure 3.1 is extended to perform a matrix addition, as shown in Figures 3.7 and 3.8. To illustrate the use of distributed programming in JR, the computations of matrix multiplication and addition are split into separate virtual machines, `vm1` and `vm2`, respectively. The main program, which runs on the

```
public class MMMultiplier {
  //remains the same as in the previous MMMultiplier class
  ...
  public double[][] getResult() {
    //return the product of A and B
    return C;
  }
}
```

Figure 3.5: Multiple rounds of matrix multiplication with dependent results between iterations – `MMMultiplier` class.

main virtual machine when the JR program starts up, creates two additional virtual machines using the `new` expression, as in a typical instantiation in JR. The value returned by `new` is a reference of type `vm`. Note that the virtual machines currently created belong to the same physical machine. However, JR provides a mechanism to allow them to be created on different physical machines by optionally specifying a physical machine's name or an existing virtual machine reference, in which case the new virtual machine will belong to that same physical machine. The code in Figure 3.6 shows the modification to virtual machines creation if they were to be created on different physical machines.

```
//machine names: garfield and sky
vm vm1 = new vm() on "garfield";
vm vm2 = new vm() on "sky";
```

Figure 3.6: Creation of virtual machines on different physical machines.

Then, the main program instantiates the `MMMultiplier` object `m` and the `MMAdder` object `n` to be placed on `vm1` and `vm2` respectively. To place an object on a different virtual machine requires the object be a remote one: such an object must be declared and created using the `remote` keyword. After the constructors of the remote objects execute, corresponding `compute` processes begin execution on different virtual machines. Lastly, the main program registers the quiescence operation `done`, which outputs the resulting product and sum of the input matrices.

```
public class MMMain {
  private static remote MMMultiplier m;
  private static remote MMAdder n;

  public static void main(String [] args) {
     int N;  // A and B are NxN
     double [][] A, B;
     // read in NxN arrays A and B
     ...

     //create virtual machines v1 and v2
     vm v1 = new vm();
     vm v2 = new vm();

     //create remote objects on v1 and v2 respectively
     m = new remote MMMultiplier(A, B, N) on vm1;
     n = new remote MMAdder(A, B, N) on vm2;

     // register done as the quiescence operation
     try {
       JR.registerQuiescenceAction(done);
     } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
        e.printStackTrace();
     }
  }
  private static op void done() {
     m.print();
     n.print();
  }
}
```

Figure 3.7: Distributed matrix multiplication and addition – `MMMain` class.

The approach to modify the matrix multiplication program into one that is distributed requires no change to the `MMMultiplier` class. The `MMAdder` class, as presented in Figure 3.8, has a similar structure as the `MMMultiplier` class but performs matrix addition.

When every process in *all* virtual machines terminates in the matrix multiplication and addition program, JR detects global quiescence. The quiescence operation is then invoked to output the final results of the matrices.

```
//The MMMultiplier class remains the same
public class MMMultiplier {
   ...
}
public class MMAdder {
  int N; // A, B, and C are NxN
  double [][] A, B, C;
  public MMAdder(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    //compute the sum for C[r,c]
    C[r][c] = A[r][c] + B[r][c];
  }
  public void print() {
    // output C
    ...
  }
}
```

Figure 3.8: Distributed matrix multiplication and addition – `MMAdder` class.

3.3 GQ Implementation in Existing JR

The implementation of the JR's quiescence feature uses an approach that differs from the general distributed termination detection algorithms described in Section 2.1 because of JR's particular model of computation. As seen in the example in Section 3.2.3, a distributed program in JR consists of a group of "virtual machines" (VMs). Each VM represents an address space, or unit of program distribution, and contains several processes, which can share variables within that address space or send message to other processes on that VM or to processes on other VMs. Such a model of computation is also used in SR and Java programs that use RMI (remote method invocation). Typically, the number of VMs is not very large, but it varies as the program executes.

3.3.1 JR Virtual Machines and Centralized Manager

JR virtual machines (JRVM) are created during run-time. As the information contained in a Java VM is not sufficient by itself to implement quiescence detection, JR creates its VM using a small Java program (`jrvm`) that provides a thin layer over the standard Java VM [18]. A JRVM, then, is able to record information, such as the number of active processes and messages that are in transit, that is useful in determining the quiescent state of the program.

As mentioned in Section 2.1, the distributed termination detection problem is a challenging one, because it requires the global state information of the system. JRX is the centralized manager of JR that possesses the global state information of the system. It is created when a JR program executes, and there is one JRX per each JR program's execution. It also provides communication to other physical hosts when creating JRVMs. When a new JRVM is to be created, the thread notifies JRX, which is responsible for contacting the specified physical host and initiating the `jrvm` program via `rsh` or a specified alternate program. When a JRVM is created, it registers itself with JRX and informs JRX that it is ready to receive requests through using RMI (remote method invocation). Thus, JRX contains a record of all JRVMs that is necessary for quiescence detection as well as other services, such as executing an explicit exit (stop) from the program code, which needs to shut down all JRVMs.

3.3.2 GQ Detection

JR's implementation of distributed termination detection involves the RTS (run-time system) on each JRVM and the centralized manager, JRX. Moreover, JRX creates a thread (the "idler" thread) that is used for the purpose of quiescence detection when a JR program starts up. As the current creation of a JRVM only adds a thin layer of mechanisms on top of the standard Java VM but does not actually modify it, quiescence detection is implemented within the `jrvm` code (by Java source code).

When a JR program creates a process or performs an asynchronous method invocation, the JR code is translated into Java code that creates a thread. Then, in the generated Java code, a "thread birth" is logged by calling a method in the `jrvm` program. Whenever this thread is blocked or terminates, a "thread death" is logged in the same manner. At a point when a JRVM can make no further progress (i.e., all of its processes (threads) have terminated or are waiting to receive a message), it sends an idle message to JRX. JRX then determines whether all JRVMs are idle, i.e., it has received an idle message from each JRVM. If so, it will awaken the "idler" thread to perform a final confirmation of idleness of all JRVMs and check that no messages are in transit, specifically: for each JRVM VM_a, the number of sends from VM_a to each other JRVM, VM_b, matches the number of receives from VM_a reported by VM_b. If JRX's "idler" determines that no messages are in transit, then the system is globally quiescent. Figure 3.9 depicts the interaction between JRX and all JRVMs.

Figure 3.10 shows the pseudo code of the "idler" thread. It has an infinite loop but executes a Java `wait` method that causes it to wait until JRX signals that all JRVMs are idle. Once awakened, it performs some final checks, as mentioned, and invokes the quiescence operation if there is one registered. Otherwise, the JR program simply terminates.

3.3.3 Costs of GQ Implementation

The implementation of GQ detection, however, has some extra overheads. The JR implementation allows a command-line argument to enable or disable GQ detection. The implementation internally tests a corresponding flag in places where it would take action to implement GQ. The additional overhead of this testing is not significant (i.e., the overall costs are nearly the same as they would be in an implementation that does

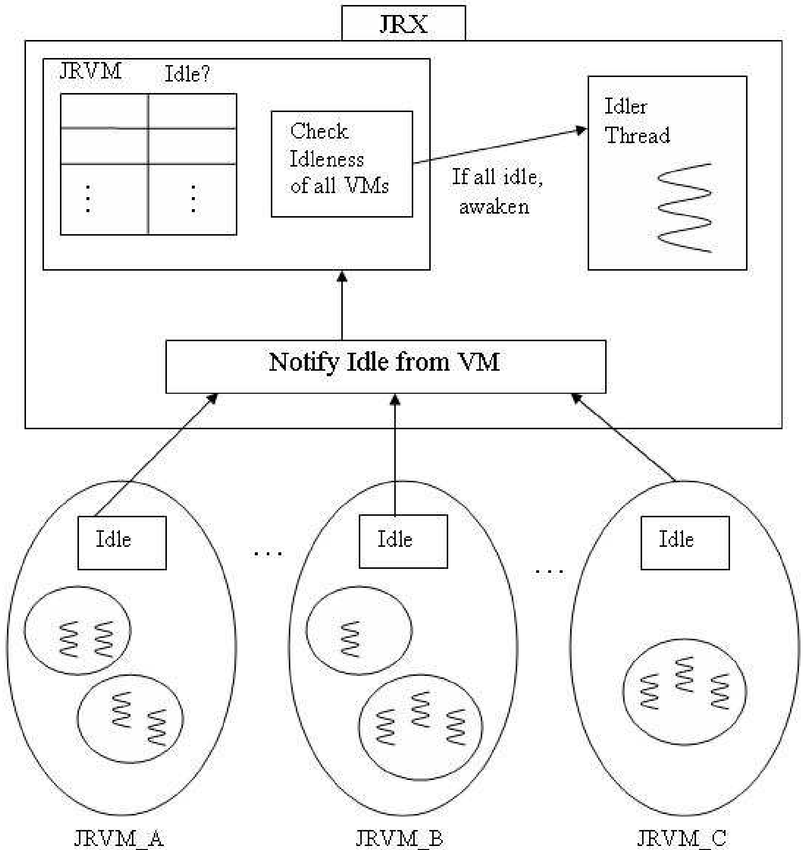

Figure 3.9: JRX and JRVMs interaction for global quiescence detection.

not implement GQ). Appendix A presents some performance results that compare the elapsed times of some simple non-distributed and distributed JR programs that have GQ detection enabled with those that do not. In general, the overall trend shows that the amount of overheads increases as the number of processes increases (e.g., up to 53% for 100 processes). Additionally, for the same number of processes, the amount of overhead also increases as the number of VMs increases (e.g., up to 26% for 8 VMs).

These overheads are due to building JRVM on top of Java VM (i.e., the additional bookkeeping required on each JRVM to determine its idleness). Since the

```
//The "idler" thread
while(true)
{
     //block until a signal is received from JRX
     //notifying that all JRVMs are idle
     wait();

     //Awakened:
     //contact every JRVM, then check its idle state
     //and get the message count
     for each JRVM_i
     {
         JRVM_i.checkIdle();
         JRVM_i.getMessageCount();
     }

     //loop until all JRVMs are idle and no messages are in transit
     if(not all idle and there are messages in transit)
        continue;

    //if a quiescence operation is not registered,
    //simply terminate the program
    if(quiescence operation Qop is null)
         exit();

    //else a quiescence operation Qop is registered:
    //asychronously invoke the quiescence operation,
    //and reset the Qop to null.
    //The "idler" thread then loops back
    //to wait for the quiescence of activities in the Qop
    else
    {
        Qop.send();
        Qop = null;
    }
}
```

Figure 3.10: Pseudo code of the “idler” thread.

GQ implementation logs information pertaining to each process, the increase in the number of processes causes an increase in bookkeeping, and thus adding extra overheads. Increasing the number of VMs, involves making more RMIs to notify JRX of a VM’s idleness, which adds additional costs to the GQ implementation. These factors, however, do not represent that the implementation of a GQ detection mechanism is always high. The performance results we obtained from running SR programs similar to the non-distributed and distributed JR programs shown in Appendix A indicate that SR's GQ detection requires at most 10% additional time. In SR, GQ detection is implemented directly as part of the run-time system, not as a separate layer as it is in the JR implementation.

Chapter 4

Partial Quiescence (PQ)

4.1 Definition of Partial Quiescence

Although global quiescence is useful, it restricts the detection to determine the quiescent state of *all* processes in a given program. Some notion of PQ, which addresses the quiescence of *part* of the program, would be useful. We want, for example, to combine two programs (i.e., two independent concurrent computations) that use global quiescence into a single program in which we want to perform different actions when each part of it becomes quiescent.

The first step is to define what partial quiescence means. A natural approach is to apply quiescence to a group of processes in a program. Modifying the definition of quiescence from Section 2.1.1 to apply to a specific group of processes yields:

> Quiescence of group A is defined as the state in which (1) there are no messages in the system in transit to group A and (2) all processes in group A have terminated or are waiting for a message.

This definition fits well if the process group is "closed" [12], i.e., only processes in group A send messages to processes in group A. However, this definition is not realistic if the process group is "open" [12], i.e., a message for a process in group A can be generated by a process outside of the group; such a message appears, from within group A, to have been generated "spontaneously". More concretely, a detection mechanism could detect that all processes in group A are passive and no message in transit is destined for group A, and so it would decide that group A is partially quiescent. However, that decision could be followed by a process outside group A sending a message to a process in group A. (In contrast, such spontaneous message generation is not possible for GQ (Section 2.1).)

A definition of PQ can deal with this spontaneous generation problem in various ways. One way would be to alter the above definition with a third clause, e.g., "and (3) no process outside of group A can possibly send to a process in group A". However, such a definition might not be useful: just because a process outside of group A can send a message to a process in group A does not guarantee that it ever actually will. Moreover, in general, keeping track of such information in a system where communication paths between processes is determined dynamically would be costly.

Therefore, we choose a weaker definition of partial quiescence, namely one that modifies (1) from the earlier definition:

> Quiescence of group A is defined as the state in which (1) there are no messages in the system from group A in transit to group A and (2) all processes in group A have terminated or are waiting for a message.

This definition fits well for closed process groups; the next sections illustrate that it is practical for open process groups.

4.2 Expository JR Examples of Partial Quiescence

We have extended JR to support partial quiescence. Now, JR programs can define groups of related processes and can register, for each process group, a *partial quiescence operation*. This section contains examples of JR programs to illustrate how partial quiescence in the extended JR works.

4.2.1 Matrix Multiplication and Addition

```
public class MMMain {
  private static MMMultiplier m;
  private static MMAdder n;

  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...

    //create process groups m_g and a_g
    //with their identifiers as arguments to the constructors
    ProcessGroup m_g = new ProcessGroup("Multiply Group");
    ProcessGroup a_g = new ProcessGroup("Add Group");

    //change the creation group appropriately
    //so that processes created in the instance
    //will belong to the specified process group
    JR.changeCreationGroup(m_g);
    m = new MMMultiplier(A, B, N);

    JR.changeCreationGroup(a_g);
    n = new MMAdder(A, B, N);

    // register partial quiescence operation for each process group
    try {
       JR.registerPartialQuiescenceAction(m_g, done_mult);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    try {
       JR.registerPartialQuiescenceAction(a_g, done_add);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
  }
  private static op void done_mult() {
    m.print();
  }
  private static op void done_add() {
    n.print();
  }
}
```

Figure 4.1: Matrix multiplication and addition using partial quiescence – `MMMain` class.

Figure 4.1 revisits the distributed matrix multiplication and addition program from Section 3.2.3, but with modifications to support partial quiescence detection.

As an introductory example, Figure 4.1 is also modified to work on a single VM, so we can focus our discussion on extending the program with partial quiescence.

This example shows how to use partial quiescence to perform matrix multiplication and addition simultaneously, then print the resulting matrix whenever either the multiplication or addition computation has been completed. In contrast, recall the main program in Figure 3.7 that performs the same computations but uses global quiescence. It would wait for *both* computations to complete before outputting the result from either.

A nice attribute of our partial quiescence approach is that it requires no change to the `MMMultiplier` class or the `MMAdder` class from Figure 3.8. The only changes take place in the main program. Figure 4.1 shows the main program that creates two process groups, `m_g` and `a_g`: processes performing multiplication are assigned to the `m_g` group and processes performing addition are assigned to the `a_g` group. Note that when instantiating a process group object, the process group's name is passed as a string argument to the constructor, which will be copied and stored in a global namespace to identify a process group in a multi-VM program. Before instantiating the `MMMultiplier` object, the main program uses `JR.changeCreationGroup()` to change the creation group from the default one to `m_g`, so that newly created processes in the `MMMultiplier` object `m` will be placed into the `m_g` process group. Likewise, before instantiating the `MMAdder` object, `JR.changeCreationGroup()` changes the creation group from `m_g` to `a_g` so that newly created processes in the `MMAdder` object `n` will be placed into the `a_g` process group. The main method then registers the partial quiescence operation for each process group. Note that to register a partial quiescence operation, the corresponding process group must be specified as well. When a partial quiescence is detected, i.e., either process group quiesces, its corresponding partial quiescence operation will be invoked, which will in turn call the `print` method to output the result.

When using partial quiescence, it is possible that the two process groups quiesce at about the same time, in which case the outputs from the quiescence operation might be interleaved. To serialize their outputs, we can delete the definitions for `done_mult` and `done_add` while keeping their `op` declarations. Then, we can add the code in Figure 4.2 to the end of the `main` method. This code uses JR's multi-way receive statement (`inni`) to wait for an invocation of either of the partial quiescence operations; it services one operation at a time, thus serializing their outputs.

```
for (int i = 0; i < 2; i++) {
  inni void done_mult() {m.print();}
  []   void done_add() {n.print();}
}
```

Figure 4.2: Code to serialize output from the multiple matrix multiplications.

As mentioned in Section 4.1, process groups can be categorized into two types: open group or closed group. The process groups `m_g`, for multiplication processes, and `a_g`, for addition processes, are closed process groups, since neither of them receives any message that is sent from outside the group. Hence, this example fits well in our partial quiescence definition, such that there is no spontaneous message generation that could happen even after the detection of partial quiescence.

4.2.2 Distributed Matrix Multiplication and Addition

Figures 4.3 and 4.4 present a distributed version of the matrix multiplication and addition program that uses partial quiescence. In order to use partial quiescence in a distributed program, our approach involves only slight changes to the program. These changes include creating remote objects and VMs in the main program of Figure 4.3, as mentioned in Section 3.2.3. The main program, which resides on the main VM, creates two additional VMs, `vm1` and `vm2`, on which the `MMMultiplier` instance `m` and the `MMAdder` instance `n` are created respectively. Process group creations, demands

```
public class MMMain {
  private static remote MMMultiplier m;
  private static remote MMAdder n;

  public static void main(String [] args) {
     int N;  // A and B are NxN
     double [][] A, B;
     // read in NxN arrays A and B
     ...

     //create process groups m_g and a_g
     //with their identifiers as arguments to the constructors
     ProcessGroup m_g = new ProcessGroup("Multiply Group");
     ProcessGroup a_g = new ProcessGroup("Add Group");

     //create virtual machines v1 and v2
     vm v1 = new vm();
     vm v2 = new vm();

     //create remote objects on v1 and v2 respectively
     //and change the creation group appropriately
     JR.changeCreationGroup(m_g);
     m = new remote MMMultiplier(A, B, N) on vm1;

     JR.changeCreationGroup(a_g);
     n = new remote MMAdder(A, B, N) on vm2;

     // register partial quiescence operation for each process group
     try {
          JR.registerPartialQuiescenceAction(m_g, done_mult);
     } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
     { e.printStackTrace(); }

     try {
          JR.registerPartialQuiescenceAction(a_g, done_add);
     } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
     { e.printStackTrace(); }
   }
   private static op void done_mult() {
      m.print();
   }
   private static op void done_add() {
      n.print();
   }
}
```

Figure 4.3: Distributed Matrix multiplication and addition using partial quiescence – `MMMain` class.

for change of creation group, and registrations of partial quiescence operations remain the same as in the non-distributed version of the program.

```
public class MMMultiplier {
  int N; // A, B, and C are NxN
  double [][] A, B, C;
  ProcessGroup m_g;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
    //instantiation of the "Multiply Group"
    //having the same identifier as in the main program
    m_g = new ProcessGroup("Multiply Group");
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    // compute the inner product for C[r,c]
    C[r][c] = 0.0;
    for (int k = 0; k < N; k++) {
      C[r][c]+= A[r][k] * B[k][c];
    }
  }
  public void print() {
    // output C
    ...
  }
}
public class MMAdder {
  int N; // A, B, and C are NxN
  double [][] A, B, C;
  ProcessGroup a_g;
  public MMAdder(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
    //instantiation of the "Add Group"
    //having the same identifier as in the main program
    a_g = new ProcessGroup("Add Group");
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    //compute the sum for C[r,c]
    C[r][c] = A[r][c] + B[r][c];
  }
  public void print() {
    // output C
    ...
  }
}
```

Figure 4.4: Distributed Matrix multiplication and addition using partial quiescence – `MMMultiplier` class and `MMAdder` class.

As the process groups are created and registered in the main program, but are used for processes located on different VMs, another change is to add the declaration and instantiation of the process group objects on the VMs where these processes are created. Specifically, the `MMMultiplier` and the `MMAdder` classes shown in Figure 4.4 contain such process group creations. Note that these additional process groups are instantiated using the *same* names as those in the main program to allow the proper reference to the same process group, e.g., the process groups named "Multiply Group" in the main program and the `MMMultiplier` class represent the same process group used to manage the processes computing matrix multiplication. (See Section 4.3.1 for further details.) Now, with these slight changes, the program functions in the same way as in the non-distributed version of the program.

4.2.3 Barrier Synchronization

Partial quiescence also allows JR programs to re-register a quiescence operation. As in global quiescence, it can be used as a barrier to delay the initiation of new activity until the current one is completed. A barrier is a common synchronization tool used in concurrent programming, and it is especially useful in iterative algorithms where some worker processes are required to finish certain tasks in one iteration before the new tasks can begin [18].

Figure 4.5 (from [18]) shows an example of a barrier synchronization program, in which a group of worker processes synchronize their iterations via a barrier, implemented with semaphores[1]. The program contains a coordinator process that controls when worker processes should begin their next iteration of task. In the program, worker processes solve parts of a program in parallel: each worker first performs some task, executes a `V` on the semaphore `done` to notify the coordinator of its completion for the current iteration, then executes a `P` on the semaphore `proceed[i]` to wait for the coordinator's message to continue a new iteration. Note that the program

[1]In JR, the semaphore primitives `P` and `V` are just special cases of the message passing primitives `receive` and `send`.

```
public class Barrier {
  private static final int N = 10; // number of workers
  private static sem done = 0;
  private static cap void () proceed[] = new cap void()[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      proceed[i] = new sem;
    }
  }
  private static process worker( (int i = 0; i < N; i++) ) {
    while (...) { //iterations remain
      // code to implement one iteration of task i
      ...
      // barrier
      V(done);       // tell coordinator "I did iteration i"
      P(proceed[i]); // wait for coordinator to say "continue"
    }
  }
  private static process coordinator {
    while (...) { //iterations remain
      for (int w = 0; w < N; w++) { P(done); }
      for (int w = 0; w < N; w++) { V(proceed[w]); }
    }
  }
  public static void main(String [] args) {
  }
}
```

Figure 4.5: Barrier synchronization using semaphores.

uses an array of semaphores `proceed` (one for each worker), rather than a single semaphore for all workers to prevent a fast worker from "stealing" the message intended for a slow worker. For example, as the coordinator executes the loop of `V` on a single semaphore `proceed`, if a fast worker wakes up from the `proceed` message (`P(proceed)`), continues its new iteration of task, then notifies the coordinator of its completion (`V(done)`), and grabs another `proceed` message (`P(proceed)`) from the coordinator that is intended for a slow worker, then the fast worker is able to begin yet another iteration of the task before all workers have completed the current iteration. Using an array of semaphores, `proceed`, eliminates such race condition problems. (See Reference [18] for details.)

This program can be rewritten using partial quiescence where fewer semaphores

```
public class Barrier {
  private static final int N = 10; // number of workers
  private static sem proceed;
  private static ProcessGroup WG;
  static {
    //instantiate process group WG
    WG = new ProcessGroup("WG");

    //change default process group to WG,
    //so that all worker processes will belong to process group WG.
    JR.changeCreationGroup(WG);
  }
  private static process worker( (int i = 0; i < N; i++) ) {
    while (...) { //iterations remain
      // code to implement one iteration of task i
      ...
      // barrier
      P(proceed); // wait for coordinator to say "continue"
    }
  }
  //coordinator is no longer a process,
  //it is now an operation invoked on partial quiescence
  private static op void coordinator()  {
    for (int w = 0; w < N; w++) { V(proceed); }
    if(...) { // iterations remain
      //re-register partial quiescence operation
      try {
        JR.registerPartialQuiescenceAction(WG, coordinator);
      }catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
      { e.printStackTrace(); }
    }
  }
  public static void main(String [] args) {
    //register partial quiescence operation
    try {
      JR.registerPartialQuiescenceAction(WG, coordinator);
    }catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
  }
}
```

Figure 4.6: Barrier synchronization using partial quiescence.

are needed to provide the necessary barrier synchronization, as shown in Figure 4.6. All worker processes are created in the process group `WG` in the beginning of the program by using `JR.changeCreationGroup()` in the static initializer. A quiescence operation is invoked when all worker processes in the group finish their iteration of the work tasks and are blocked on the barrier waiting for the coordinator's message to continue. Note that the role of the coordinator is now performed by a quiescence operation instead of a separate process. The semaphore `done`, which was previously used by the workers to notify the coordinator that they are done, is no longer needed, because partial quiescence would detect when all worker processes have done their iteration of the work task, and invoke the coordinator operation when they have become quiescent. Moreover, the array of `proceed` semaphores is now replaced with a single `proceed` semaphore, as it is now impossible for a fast worker to "steal" a message from a slow worker since all workers must quiesce before the coordinator operation begins to notify any worker to proceed.

The process group `WG` used in this example is open, because the `proceed` messages are sent from the `coordinator` operation, which is outside of group `WG`, to the worker processes in group `WG`. Yet, this example illustrates that the definition we chose is still practical for open process group. Although there are messages generated from outside the group, these messages are all guaranteed to have been generated before the next partial quiescence operation is registered; that is, before the program begins the next phase of partial quiescence detection.

In this example, global quiescence can be used instead of partial quiescence to obtain the same effect of barrier synchronization. However, if barrier synchronization is used only as a component of a larger application in which there are separate processes that do not require barrier synchronization, only partial quiescence is effective to achieve the desired synchronization, as global quiescence would cause the workers to delay their next iteration of tasks until the separate processes unrelated to barrier synchronization have also become quiescent.

```
public class Main {
  //number of workers in each VM
  private static final int N = 10;
  private static remote BW worker_one;
  private static remote BW worker_two;
  private static op void proceed();
  private static ProcessGroup WG;

  public static void main(String [] args) {
    //instantiate process group WG
    WG = new ProcessGroup("WG");

    //create virtual machines v1 and v2
    vm vm1 = new vm();
    vm vm2 = new vm();

    //change default process group to WG,
    //so that all worker processes will belong to process group WG.
    JR.changeCreationGroup(WG);

    //create remote workers on v1 and v2 respectively
    worker_one = new remote BW(N, proceed) on vm1;
    worker_two = new remote BW(N, proceed) on vm2;

    try {
      JR.registerPartialQuiescenceAction(WG, coordinator);
    }catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
  }

  public static op void coordinator() {
    for (int w = 0; w < 2*N; w++) { V(proceed); }
    if(...) { // iterations remain
      try {
        JR.registerPartialQuiescenceAction(WG, coordinator);
      }catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
      { e.printStackTrace(); }
    }
  }
}
```

Figure 4.7: Barrier synchronization with distributed workers using PQ – `Main` class.

4.2.4 Barrier Synchronization with Distributed Workers

The barrier synchronization example using partial quiescence also works in a distributed program. Figures 4.7 and 4.8 illustrate an example of the barrier synchronization program with distributed workers. It differs from the non-distributed version

```
public class BW {
  private final int N; // number of processes
  private cap void () proceed;
  private ProcessGroup WG;

  public BW(int N, cap void() proceed) {
    this.N = N;
    this.proceed = proceed;
    //instantiation of the "WG" group
    //having the same identifier as in the main program
    WG = new ProcessGroup("WG");
  }

  private process worker( (int i = 0; i < N; i++) ) {
    while (...) { //iterations remain
      //code to implement one iteration of task i
      ...
      //barrier: wait for coordinator
      //in the main program to say "continue"
      P(proceed);
    }
  }
}
```

Figure 4.8: Barrier synchronization with distributed workers using PQ – `BW` class.

in that instances of the `BW` objects are created so that worker processes are located on different VMs. As shown in Figure 4.8, after each worker process completes one iteration of its work task, it blocks on a capability, which references the `proceed` operation located remotely in the main program (Figure 4.7), instead of a semaphore. When the program detects partial quiescence, i.e., worker processes in all VMs belonging to the `WG` group are blocked, the coordinator in the main program performs the same actions as in Figure 4.6 to awaken all workers on different VMs.

4.3 Key Aspects of Partial Quiescence in Extended JR

As seen in the examples in the previous section, process groups allow the programmer to specify parts of the program for separate PQ detection. This section describes the key aspects of partial quiescence in our extended JR.

4.3.1 Process Group Identification

The names of process groups, specified by the string argument to the `ProcessGroup` constructor, are in a global namespace. In the case of a distributed program where processes from different VMs are in the same process group, each VM needs to create the process group with the same identifier. For example, in the multi-VM barrier synchronization program (Section 4.2.4), processes created in process group `"WG"` on two different VMs are in the same process group. The programmer can also create a process group specific to a VM by using a per-VM unique identifier in the name (e.g., by using parameterized VMs to include some notion of "VM id").

Note that other systems such as Horus, ISIS, MPE, and PVM provide support to organize groups of processes [12]. However, their process groups were not designed for the purpose of quiescence detection. For example, process groups in MPI are used for collective communication purposes, such as sending a message to a group of processes.

4.3.2 Registration of PQ Operation

PQ detection for a process group does not begin until the PQ operation has been registered. This avoids the following "startup problem". Suppose a process group has just been created, but no processes have yet been created within that group. Then, PQ detection would detect that the group has quiesced, which would not be

```
public class MMMain {
  private static MMMultiplier m;
  private static MMAdder n;

  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...

    //create process groups m_g and a_g
    //with their identifiers as arguments to the constructors
    ProcessGroup m_g = new ProcessGroup("Multiply Group");
    ProcessGroup a_g = new ProcessGroup("Add Group");

    // register partial quiescence operation for each process group
    try {
       JR.registerPartialQuiescenceAction(m_g, done_mult);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    try {
       JR.registerPartialQuiescenceAction(a_g, done_add);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    //change the creation group appropriately
    //so that processes created in the instance
    //will belong to the specified process group
    JR.changeCreationGroup(m_g);
    m = new MMMultiplier(A, B, N);

    JR.changeCreationGroup(a_g);
    n = new MMAdder(A, B, N);

  }
  private static op void done_mult() {
    m.print();
  }
  private static op void done_add() {
    n.print();
  }
}
```

Figure 4.9: Potential problem for partial quiescence in matrix multiplication and addition – `MMMain` class.

too useful for the programmer. For example, Figure 4.9 shows a variant of the main program from Figure 4.1 that registered its PQ operations before instantiating the `MMMultiplier` and the `MMAdder` objects. Similarly, the same problem exists if some processes of the group might have been created and already terminated before other processes in the group have even been created. Moreover, in a distributed program, this "startup problem" might involve the early PQ detection before the process group and its processes in another VMs have been created. For example, if the main program in Figure 4.7 registered its PQ operation after instantiating the first worker object, then PQ detection might detect that the group has quiesced when only the first worker in `vm1` completes its task. This is undesirable, as the second worker has not yet created its process group and processes.

Registration of a PQ operation applies globally in the distributed program rather than per-VM. Any subsequent PQ registration in any VM that is executed before PQ is detected will merely update the PQ op to be invoked. Just as in GQ, the PQ operation can start up new activity and can re-register another PQ operation.

4.3.3 Potential Nondeterministic Behaviors

The precise definition of PQ for JR differs slightly from that given in Section 4.1. The reason is that in JR a message is sent to an operation, which can be serviced by processes that might belong to different process groups. The PQ definition for JR, therefore, says "(1) there are no messages in the system *that are serviceable by a process in group* A from group A in transit to group A".

Because PQ's definition omits dealing with messages in transit from outside the process group, a program that uses PQ can be nondeterministic. For example, a message from outside a process group might be sent either before or after PQ is detected for that process group, thus affecting program behavior. However, such nondeterminism does not occur in the examples in this thesis (or other practical examples we have written so far). It remains to be seen whether such nondeterminism is a problem in further practice.

4.3.4 PQ vs. GQ

PQ is an extension to, not a replacement for, GQ. A program is globally quiescent when all parts of it have become partially quiescent and the remaining processes not associated with any group have terminated or deadlocked, and no messages are in transit.

4.3.5 Feature to Disable and Enable PQ Detection

The extended JR has an additional PQ feature that allows the program to disable or enable PQ detection features during execution. This feature allows the programmer to pause the detection of partial quiescence of a process group, by calling the `JR.disableDetection()` method, until (typically) some desired processes are added to the process group and a `JR.enableDetection()` method is called. The disable and enable methods take as a parameter a reference to the process group object.

4.3.6 Specification of Expected Number of Processes

An optional argument to the process group constructor can specify the number of processes expected in the group; quiescence of the group occurs when that number of processes has terminated or deadlocked. Note that in a distributed program, this number that is specified in the process group constructor of *each* VM applies locally to the number of terminated or deadlocked processes in that particular VM.

This feature can be used to avoid the “startup problem” discussed in Section 4.3.2 even if a PQ operation is registered before the desired processes have been created in the process group. Figure 4.10 shows an example that eliminates the potential “startup problem” seen in Figure 4.9 by specifying the number of expected processes in group `m_g` and group `a_g` to the `ProcessGroup` constructors. Now, PQ detection would not detect that the group has quiesced even though there are no processes

```
public class MMMain {
  private static MMMultiplier m;
  private static MMAdder n;

  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...

    //create process groups m_g and a_g
    //with their identifiers
    //AND the number of expected processes in each group
    //as arguments to the constructors
    ProcessGroup m_g = new ProcessGroup("Multiply Group", N*N);
    ProcessGroup a_g = new ProcessGroup("Add Group", N*N);

    // register partial quiescence operation for each process group
    try {
       JR.registerPartialQuiescenceAction(m_g, done_mult);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    try {
       JR.registerPartialQuiescenceAction(a_g, done_add);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    //change the creation group appropriately
    //so that processes created in the instance
    //will belong to the specified process group
    JR.changeCreationGroup(m_g);
    m = new MMMultiplier(A, B, N);

    JR.changeCreationGroup(a_g);
    n = new MMAdder(A, B, N);

  }
  private static op void done_mult() {
    m.print();
  }
  private static op void done_add() {
    n.print();
  }
}
```

Figure 4.10: A remedy to the potential problem for partial quiescence in matrix multiplication and addition – `MMMain` class.

created in the group yet, because it would wait for the N $\times$ N `compute` processes in each group to deadlock or terminate before detecting partial quiescence.

Specifying the number of expected processes, however, is sometimes not useful, because this feature requires some advance knowledge of the number of processes in the process group. For example, in the original matrix multiplication and addition program (Figure 4.1), the number of processes created (`N` × `N`) was known only within the `MMMultiplier` and the `MMAdder` classes. In the version in Figure 4.10, that information now needs to be known outside of those classes, which violates information hiding. If the MMMultiplier class chose to create `N` processes (instead of `N` × `N` processes) such that each process computes the result of a row of the matrix, then the `MMMain` class would need to be changed too.

4.3.7 Change of Process Group

A process can change its process group in the middle of execution by calling the `JR.changeCurrentGroup()` method with a `ProcessGroup` object as an argument. This feature is useful, for example, when processes of one object instance are to belong to different process groups. In this case, calling the method `JR.changeCreationGroup()` before instantiation of the object is impractical, because *all* processes created in that object instance would belong to the same process group.

Consider a different approach in writing the matrix multiplication and addition program as shown in Figures 4.11 and 4.12. Now, the `MMMultiplier` class and the `MMAdder` class have been merged into a single class: the `MMComputer` class that contains both the `multiply` and the `add` processes. In this programming approach, the only way to place these processes into their corresponding process groups is to change their groups in the beginning of their execution rather than at the creation stage. To do so, the main program in Figure 4.11 needs to pass in the `m_g` group and the `a_g` group in the `MMComputer`'s constructor, so that the processes within the `MMComputer`'s instance have references to these groups. As illustrated in Figure 4.12, we first call

```
public class MMMain {
  private static MMComputer m;

  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...

    //create process groups m_g and a_g
    //with their identifiers
    //AND the number of expected processes in each group
    //as arguments to the constructors
    ProcessGroup m_g = new ProcessGroup("Multiply Group", N*N);
    ProcessGroup a_g = new ProcessGroup("Add Group", N*N);

    //creation group is not changed here,
    //as all processes in m do not belong to the same group
    m = new MMComputer(A, B, N, m_g, a_g);

    // register partial quiescence operation for each process group
    try {
       JR.registerPartialQuiescenceAction(m_g, done_mult);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }

    try {
       JR.registerPartialQuiescenceAction(a_g, done_add);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
  }
  private static op void done_mult() {
    m.print_mult();
  }
  private static op void done_add() {
    m.print_add();
  }
}
```

Figure 4.11: Matrix multiplication and addition in the same class – `MMMain` class.

`JR.changeCurrentGroup()` in each of the `multiply` or `add` process in order to place the process into the appropriate process group. However, note that since the PQ operations are registered in the main program right after these processes are created, PQ detection might potentially detect quiescence before the `multiply` and `add` processes are placed into the `m_g` and `a_g` group. This "startup problem", again, can

```
public class MMComputer {
  int N; // A, B, C, and D are NxN
  double [][] A, B, C, D;
  ProcessGroup m_g, a_g;
  public MMComputer(double [][] A, double [][] B, int N,
                    ProcessGroup m_g, ProcessGroup a_g) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N]; D = new double [N][N];
    this.m_g = m_g; this.a_g = a_g;
  }
  process multiply ( (int r = 0; r < N; r++),
                     (int c = 0; c < N; c++) ) {
    //change process group to m_g
    //in the beginning of process execution
    JR.changeCurrentGroup(m_g);

    //compute the inner product for C[r,c]
    C[r][c] = 0.0;
    for (int k = 0; k < N; k++) {
      C[r][c]+= A[r][k] * B[k][c];
    }
  }
  process add ( (int r = 0; r < N; r++),
                (int c = 0; c < N; c++) ) {
    //change process group to a_g
    //in the beginning of process execution
    JR.changeCurrentGroup(a_g);

    //compute the sum for D[r, c]
    D[r][c] = A[r][c] + B[r][c];
  }
  public void print_mult() {
    // output C
    ...
  }
  public void print_add() {
    // output D
    ...
  }
}
```

Figure 4.12: Matrix multiplication and addition in the same class – `MMComputer` class.

be avoided by specifying the number of expected processes in each process group's constructors.

The change of process group feature is also useful when a process wants to move to another process group when some conditions are met. For example, consider a tennis tournament program that has several processes and process groups. Each

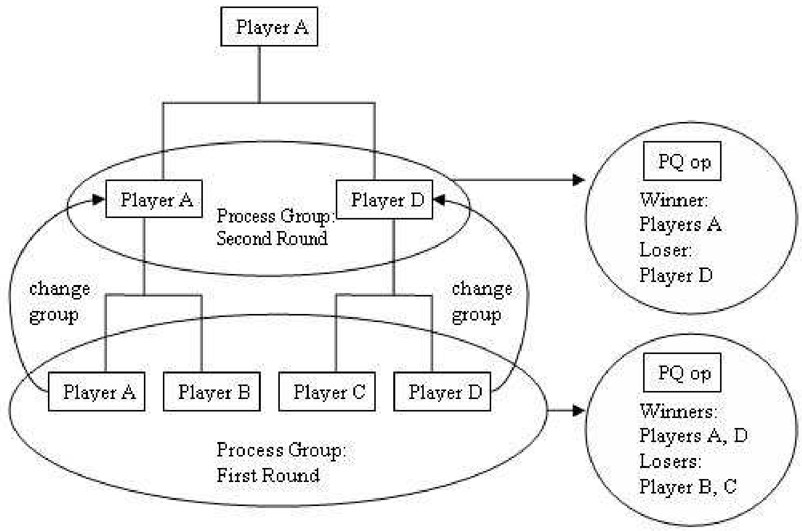

Figure 4.13: Process groups structure in a tennis tournament program.

process represents a tennis player, whereas each process group represents one round of competition. All tennis players first belong to a process group during the first round of competition. When all players have finished playing, the process group quiesces and outputs statistics of all the winners and the losers. The winning players then continue to the next round of competition and *change* their process group to the second round; as before, statistics are outputted when the process group of the second round quiesces. The program continues in the same manner to the final round until a player becomes the final champion. Figure 4.13 illustrates an overall structure of process groups in this tennis tournament example.

Figures 4.14–4.19 show the JR program for the tennis tournament example. This program simulates the tennis tournament's structure in Figure 4.13, and thus is written to work only with two rounds of competition (rather than any number of rounds in general, although it could be generalized fairly easily). Figure 4.14 contains the `TennisMatch` class with processes, operations, and the `main` method declarations.

```
public class TennisMatch {
   //some variables and ops declaration
   ...  //see a subsequent figure for the actual code
  //static initializer to initalize variables
  static {
    ... //see a subsequent figure for the actual code
  }
  private static process players( ( int i=0; i<N; i++) ) {
    ... //see a subsequent figure for the actual code
  }
  //coordinator to control barrier synchronization in players process
  private static process coordinator {
    ... //see a subsequent figure for the actual code
  }
  //called by player when a game begins
  private static op void play(int myID, int oppID) {
    ... //see a subsequent figure for the actual code
  }
  //PQ op for process group "Round 1"
  public static op void r1Stats() {
    ...  //see a subsequent figure for the actual code
  }
  //PQ op for process group "Round 2"
  public static op void r2Stats() {
    ...  //see a subsequent figure for the actual code
  }
  //GQ op to output final champion
  public static op void finalStats() {
    ...  //see a subsequent figure for the actual code
  }
  public static void main(String [] args) {
    ...  //see a subsequent figure for the actual code
}
```

Figure 4.14: Tennis tournament program – `TennisMatch` class.

(Due to space constraint, their definitions are presented in the figures that follow.) Figure 4.15 shows the declarations of some bookkeeping variables related to each player's status as well as the process group for each round of competition. The static initializer initializes the variable values and instantiates the process group objects.

Figure 4.16 contains the quantified `players` processes such that each represents one tennis player. Each tennis player process is first assigned an opponent before invoking the `play` operation to begin the game. When the game is done, a player waits on a barrier until all players in the same round have finished playing; after that,

```
public class TennisMatch {
  private static final int N = 4;   //number of tennis players

  //current game status
  private static boolean status[] = new boolean [N];

  //round 1 and round 2 player status
  private static boolean r1Status[] = new boolean [N];
  private static boolean r2Status[] = new boolean [N];

  //opponent ID for each player
  private static int paired[] = new int [N];

  //number of games each player has won
  private static int winCount[] = new int [N];

  //used for barrier
  private static cap void () proceed[] = new cap void () [N];
  private static sem done  = 0;

  //process group for round 1 and round 2
  private static ProcessGroup r1, r2;
  private static sem mutex = 1;

  //static initializer to initalize variables
  static {
    for(int i=0; i<N; i++) {
      status[i] = true; r1Status[i] = false; r2Status[i] = false;
      paired[i] = -1; winCount[i] = 0;  proceed[i] = new sem;
    }
    r1 = new ProcessGroup("Round 1"); r2 = new ProcessGroup("Round 2");
  }
  ...
}
```

Figure 4.15: Tennis tournament program – variables declaration and static initializer.

they can then proceed to the next round. Also, the winner needs to change its process group to the next round, while the loser process simply terminates. When all players have quiesced in the current round's process group (each process has either changed to the next round's process group or has terminated), a PQ operation is invoked to output the statistics of the current round and to register the PQ operation for the next round's process group. Each winner process then proceeds to the next round of competition until a final champion is generated.

Figure 4.17 shows the details of the `play` operation that is invoked when players

```
private static process players( ( int i=0; i<N; i++) ) {
  //each player join round 1 process group
  JR.changeCurrentGroup(r1);
  //round number
  int counter = 1;
  //proceed to the next round only if the player has won
  while(status[i] == true) {
    //opponent assignment, "opp_id" and "pair[i]" are assigned
    ...
    //player begins playing
    P(mutex);
    play(i, opp_id);
    V(mutex);
    //barrier to wait until all players have finished the current round
    //before proceeding
    if(counter==1) {
      V(done);
      P(proceed[i]);
    }
    //assign winning or losing status for each round
    if(counter == 1)
      r1Status[i] = status[i];
    else if (counter == 2)
      r2Status[i] = status[i];
    //if player has won, update variables appropriately
    //and the player to the next round process group
    if(status[i]) {
      winCount[i]++;
      paired[i] = -1;
      if(winCount[i] == 1) {
        JR.changeCurrentGroup(r2);
      }
    }
    counter++;
    //no more rounds
    if(counter>2) break;
  }
}
```

Figure 4.16: Tennis tournament program – the `players` process.

begin a game. It basically uses the random generator to decide who the winner and the loser are in order to simulate an actual tennis game. This figure also shows the `coordinator` process that provides the necessary barrier synchronization for each player. Figure 4.18 contains the definition of the PQ operations and a GQ operation. Each PQ operation outputs the statistics pertaining to the round of competition it

```
//To simulate the tennis game,
//use random generator to decide the winner
private static op void play(int myID, int oppID) {
  Random r = new Random();
  //if both players status is true, players havent' played yet
  if(status[myID] && status[oppID])  {
    if(r.nextDouble() > 0.5)  {
      status[myID] = true;  status[oppID] = false;
    }
    else  {
      status[oppID] = true;  status[myID] = false;
    }
  }
}
//coordinator to control barrier synchronization in players process
private static process coordinator {
  while(true)  {
    for(int w=0; w<N; w++) {P(done);}
    for(int w=0; w<N; w++) {V(proceed[w]);}
  }
}
```

Figure 4.17: Tennis tournament program – the `play` op and the `coordinator` process.

represents. Moreover, the PQ operation of the first round's process group registers the PQ operation for the second round. The GQ operation simply outputs the final champion at the end. Figure 4.19 shows the main program that registers a PQ operation for the first round of competition and the GQ operation.

Note that when a process changes its group, the active state of this process is transferred to the new group. As this process no longer belongs to its previous process group, the previous process group would consider this process as being terminated in the group. With the change of process group feature, the precise definition of PQ for JR differs slightly from that given in Section 4.3.3. The definition now becomes:

> Quiescence of group A is defined as the state in which (1) there are no messages in the system that are serviceable by a process in group A from group A in transit to group A and (2) all processes in group A have terminated, are waiting for a message, *or have left the group.*

```
//PQ op for process group "Round 1"
public static op void r1Stats() {
  System.out.println("Round 1 statistics: ");
  for(int i=0; i<N; i++) {
    if(r1Status[i]) System.out.println("player " + i + " wins");
    else  System.out.println("player " + i + " loses");
  }
  //register PQ op for the next round
  try {
    JR.registerPartialQuiescenceAction(r2, r2Stats);
  } catch(edu.ucdavis.jr.QuiescenceRegistrationException e)
  { e.printStackTrace(); }
}
//PQ op for process group "Round 2"
public static op void r2Stats() {
  System.out.println("Round 2 statistics: ");
  for(int i=0; i<N; i++) {
    if(r2Status[i])  System.out.println("player " + i + " wins");
    else  System.out.println("player " + i + " loses");
  }
}
//GQ op to output final champion
public static op void finalStats() {
  for(int i=0; i<N; i++)  {
    if(status[i]) {
      System.out.println("*****The champion is player " + i + "*****");
      break;
    }
  }
}
```

Figure 4.18: Tennis tournament program – PQ ops and GQ op.

4.3.8 Process Group Hierarchy

Process groups can be hierarchical. A parent group is defined to have become partially quiescent only when all of its child process groups and its own processes have become quiescent. To organize process groups into a hierarchy, JR users can specify an array of child process groups as an optional argument in the `ProcessGroup` constructor.

Figures 4.20–4.22 illustrate a bus stop example that makes use of the process group hierarchy to determine quiescence of child process groups. The program simulates a scenario in which there are some buses waiting in line to pick up some passengers who

```
    //the main program with PQ and GQ registration
    public static void main(String [] args) {
      //register PQ op for the first round
      try {
        JR.registerPartialQuiescenceAction(r1, r1Stats);
      } catch(edu.ucdavis.jr.QuiescenceRegistrationException e)
      { e.printStackTrace(); }

      //register GQ op to out the final champion
      try {
        JR.registerQuiescenceAction(finalStats);
      } catch(edu.ucdavis.jr.QuiescenceRegistrationException e)
      { e.printStackTrace(); }
    }
```

Figure 4.19: Tennis tournament program – the `main` method.

```
public class Main {
  public static void main (String [] args) {
    //size[0] denotes the number of buses
    //size[1] denotes the number of passengers
    //set default sizes
    int [] size = {10, 10};

    //get inputs: sizes of buses and passengers
    for (int i = 0; (i < args.length) && (i < 2); i++) {
      size[i] = Integer.parseInt(args[i]);
    }

      //create a bus stop with the specified sizes
      new BusStop(size[0], size[1]);
  }
}
```

Figure 4.20: A bus stop program with child process groups – `Main` class.

are also waiting in line. We want to know when *each* bus has finished picking up its passengers (partial quiescence of each bus process group), and when *all* buses have left the bus stop (partial quiescence of the bus stop process group, which contains each bus process group as a child). The main program given in Figure 4.20 creates a `BusStop` instance with the specified number of passengers to be picked up by each bus and the total number of buses in the bus stop.

In the `BusStop`'s constructor (Figure 4.21), an array of child process groups

```
public class BusStop {
  private int B;   //number of buses
  private int PB;  //number of passengers
  // a list of buses
  private List busesObj = Collections.synchronizedList(new ArrayList());
  //an array of child process groups, each represents a bus
  private ProcessGroup [] pg_buses;
  //a bus stop process group, parent group of all "busses" groups
  private ProcessGroup pg_busStop;
  private op void notFull(); //tell the passenger to take a bus
  public BusStop (int B, int PB) {
    this.B = B; this.PB = PB;
    pg_buses = new ProcessGroup[B];
    //instantiate the array of "buses" process group
    for(int i=0; i<B; i++) {
      pg_buses[i] = new ProcessGroup(Integer.toString(i), PB);
    }
    //instantiate the busStop process group, child groups used as an argument
    pg_busStop = new ProcessGroup(pg_buses, "Bus Stop");
    //register PQ operation process group pg_busStop
    try {
      JR.registerPartialQuiescenceAction(pg_busStop,done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
    for(int i = 0; i < B; i++) {
      new Bus(i, PB, busesObj, pg_buses[i], notFull);
    }
    //notify the first passenger to take a bus
    send notFull();
  }
  private process passengersLine {  //a line of passengers waiting for buses
    for(int i = 0; i < B*PB; i++) {
      //wait for an available bus and add a passenger to it
      receive notFull();
      synchronized (busesObj) {
        send ((Bus)(busesObj.get(0))).add_passenger();
      }
    }
  }
  private op void done () {
    System.out.println("***All buses have left***");
  }
}
```

Figure 4.21: A bus stop program with child process groups – `BusStop` class.

named `pg_buses` is created and passed in as an argument to its parent process group `pg_busStop`. Each process group in the `pg_buses` array represents one bus. The parent process group `pg_busStop` is registered with a PQ operation; it quiesces when

```
public class Bus {
  private int id;
  private int size;
  private List busesList;
  private ProcessGroup pg_bus;
  private cap void () notFull;
  private int passAdded;  //number of passengers added so far
  public Bus (int id, int size, List busesList,
              ProcessGroup pg_bus, cap void () notFull) {
    this.id = id;  this.busesList = busesList;  this.pg_bus = pg_bus;
    this.size = size; this.notFull = notFull;
    //add this bus object to the busObj list in main program
    synchronized (busesList) {
      busesList.add(this);
    }
    System.out.println("Bus " + id + " has arrived");
    //register PQ operation for each bus process group
    try {
      JR.registerPartialQuiescenceAction(pg_bus, done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
    { e.printStackTrace(); }
  }
  public op void add_passenger () {
    //a passenger has joined this bus's process group
    JR.changeCurrentGroup(pg_bus);
    System.out.println("Passenger added to bus " + id);
    //notify the next passenger that this bus has space
    if(size > ++passAdded) send notFull();
  }
  //PQ op: removes itself from the line of buses
  //when this bus finishes picking up its passengers,
  private op void done () {
    synchronized (busesList) {
      for (int i = 0; i < busesList.size(); i++) {
        if (((Bus) (busesList.get(i))).id == id) {
          busesList.remove(i);
          break;
        }
      }
    }
    System.out.println("Bus " + id + " full and leaving");
    //notify the next passenger to take the next bus
    send notFull();
  }
}
```

Figure 4.22: A bus stop program with child process groups – `Bus` class.

all of its child process groups `pg_buses` have quiesced, i.e., they have finished picking up their specified number of passengers and left the bus stop. The constructor also creates the specified number of `Bus` instances. When a `Bus` instance is created, its constructor adds itself into the list of waiting buses (`busesObj` created in the `busStop` class), and indicates its arrival to the bus stop. It then registers a partial quiescence operation to detect when it has finished picking up the specified number of passengers.

The `BusStop` class has a `passengersLine` process that represents the line of passengers waiting to take a bus. In the process, each passenger in line is added to a bus, denoted as Bi of type `Bus`, that has arrived and has an available space (i.e., when a `notFull()` message is received) by asynchronously invoking the op method[2] `add_passenger()` of bus Bi.

The `add_passenger()` operation changes its process group in the `Bus` class to the `pg_bus` process group, which is a reference to one of the child process groups in the `BusStop` class's array `pg_buses`. It then checks whether the bus is full yet. If not, it sends a `notFull()` message to the next passenger in line to take this bus. If the bus is full, a `notFull()` message will not be sent until the PQ operation is invoked. A child process group quiesces when the specified number of passengers have finished their executions of the `add_passenger()` op method (i.e., the process terminates). When the bus quiesces, it invokes the PQ operation `done` in the `Bus` class to remove itself from the list of waiting buses, and notifies the waiting passenger to take the next available bus by sending a `notFull()` message. At the point when all buses quiesce, the parent process group `pg_busStop` detects quiescence and outputs a message indicating all buses in the bus stop have left.

With the hierarchical partial quiescence detection feature, JR users can then perform their desired actions when groups of process groups have become quiescence without too much additional effort.

[2] Recall that in JR, an asynchronous invocation serviced by an op method is equivalent to creating a JR process.

4.4 Implementation of Partial Quiescence

Our current implementation of PQ is built on JR version 1.00061, which is based on Java 1.4. The implementation will be ported to JR version 2.00001, which is based on Java 1.5. The implementation of PQ adapts the centralized manager implementation of GQ described in Section 3.3. The implementation of PQ for closed process groups (Section 4.1) could follow the GQ implementation rather directly, but with message counts specific to process groups. This section discusses the details of how PQ is implemented in JR.

4.4.1 JR Thread and Process Group

In the existing JR, the generated code for creating a JR process creates a Java thread. To provide the partial quiescence detection feature in our extended JR, we introduce a new class, `JRThread`, which extends Java `Thread`. `JRThread` is used instead of Java `Thread` for the following reasons. A JR process is allowed to change its process group in the middle of the execution; while in Java, once a thread is associated with a thread group (which occurs when the thread is created), its thread group attribute can never be changed. `JRThread` then contains an attribute to represent the current group it belongs to, which can be modified. `JRThread` also contains a creation group attribute, which can be modified as well when the `JR.changeCreationGroup()` method is called. The creation group represents the process group for the JR processes that a `JRThread` creates.

As mentioned in Section 2.4, Java also provides means to organize a group of threads as a thread group for management and security purposes. To do so, Java programmers pass in a `ThreadGroup` object as an argument to the Java `Thread` constructor. However, Java `ThreadGroup` does not provide the necessary support for partial quiescence detection in JR. As such, the implementation of PQ uses the `ProcessGroup` class, which extends Java `ThreadGroup`. By extending Java `ThreadGroup`, we can organize JR processes into process groups by simply including an extra parameter for the `JRThread`'s constructor during JR code translation (which invokes the constructor of its superclass, Java `Thread`, at runtime), so that a `ProcessGroup` object can be passed in to the constructor at runtime. The `ProcessGroup` class contains features that are useful to PQ detection, such as keeping a counter for the number of active processes, notifying JRX of its idle state, and logging "thread birth" and "thread death" (similar to the features provided in `jrvm`, as mentioned in Section 3.3.1, that are used in GQ detection). The `ProcessGroup` class is part of the `JR` package.

4.4.2 The Centralized Manager and PQ Detection

As in the implementation of GQ, the centralized manager (JRX) plays a significant role in PQ detection. In the GQ implementation, JRX contains the global state information such as the references to all VMs that are created in the application, and their idle states. In the PQ implementation, JRX contains some additional information, which includes a record of process group identifiers, the lists of VMs on which each process group has been created and a capability for the PQ operation.

More concretely, when a process group is created on a VM, the RTS (run-time system) on the VM sends a message to the manager. The manager uses the process group name as the key into a hashtable; the hashtable entry contains the list of VMs on which the process group has been created and a capability for the PQ operation. When the PQ operation is registered, it is sent by the RTS to the manager. The manager then creates a thread (the "PGIdler" thread, which works in a similar way as the "idler" thread in global quiescence detection) to handle quiescent messages for this group (if such a thread has not already been created). The thread executes until the group becomes quiescent (as described in the following paragraph), at which point the thread invokes the PQ operation and terminates. If the PQ operation is re-registered, a new "PGIdler" thread is created. Exactly when the thread is created is important so that the thread does not detect quiescence before the operation has been (re-)registered, i.e., to avoid the "startup problem" mentioned in Section 4.3.2.

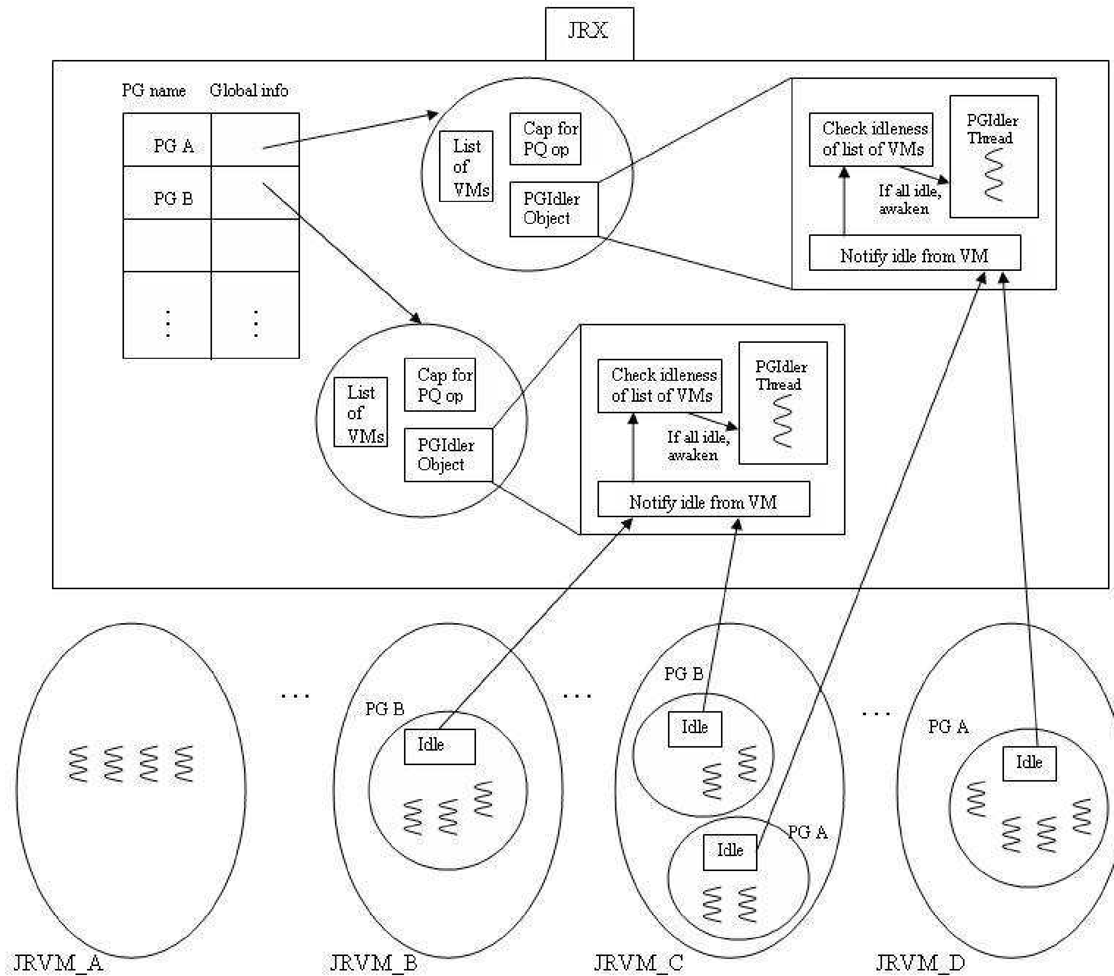

Figure 4.23: JRX and JRVMs interaction for partial quiescence detection.

Each time a process of a particular group is created, a "thread birth" is logged by calling a method in the process group instance specific to the VM in which the process is located. Likewise, when a process is blocked or terminates, a "thread death" is logged by calling a method in the same process group instance. Whenever a process changes its group in the middle of the execution, a "thread death" is logged in its old process group and a "thread birth" is logged in its new process group. When the RTS on a VM detects that a process group on that VM becomes quiescent (i.e., all of its processes have terminated or are waiting to receive a message), it sends an idle message to the manager. The manager then determines whether all process groups (with the same name) in different VMs are idle, i.e., it has received an idle message from each VM that contains this process group. If so, it will awaken the "PGIdler" thread that is managing the process group. The thread then sends a message to *all* VMs to confirm that each VM is indeed idle. If the manager receives such confirmation, then the process group is quiescent. Otherwise, it waits for idle messages from those VMs who reported they were not idle before it attempts confirmation again. Note that in the re-confirmation phase, the thread sends messages not only to the VMs that have previously reported non-idle, but to *all* VMs, including those who have previously reported idle. This second, confirmation phase to all VMs is necessary to account for one VM reporting that the process group is idle just after it sends a message to another process within the same process group on another VM that already reported that it was idle. Now, as the confirmation phase also checks for the idleness of a VM that has previously reported idle, the confirmation phase ensures the proper detection of a VM becoming non-idle again, i.e., to implement the modified PQ definition in Section 4.3. The diagram in Figure 4.23 illustrates the interaction between the process groups in each VM and the centralized manager, JRX, when detecting partial quiescence.

An alternate implementation approach similar to the one just discussed can also work for PQ detection. In this alternate implementation approach, instead of having a VM notify JRX of its idleness, JRX would periodically probe each VM to obtain its current state of the process group and whether that state has changed since the last probe. If JRX detects that the state of any VM has changed from non-idle to idle since the last probe, then a message could have possibly been sent to a process group on a VM that JRX has previously found idle. Therefore, a subsequent probe is needed to confirm the idleness of all VMs. Once JRX is able to obtain all idle states from each VM and that no VM has changed its state since the last probe, PQ is detected. However, this implementation approach leads to a concern of how often the probing is necessary. The more often probing is performed, the higher the overhead; whereas the less often probing is performed, the higher the chance to delay PQ detection.

Chapter 5

Performance of Partial Quiescence

Because PQ is a new language feature, we have no direct basis of comparison to assess the performance of our implementation. However, we have compared the performance of PQ in several programs with the performance of GQ in roughly comparable programs.

The results gathered were measured in elapsed execution time (in seconds), based on Linux's `time` command. For each test, we measured the result five times (the variance was insignificant), and calculated an average elapsed time, which is presented in the subsequent tables. We also ran the entire group of tests multiple times and observed that the results were nearly identical. We ran our tests on various PCs (1.4GHz, 2.0GHz, and 3.0GHz uniprocessors; 2.4GHz and 2.8GHz dual-processors) running Linux. Specific results, of course, varied according to platform, but the overall trends were the same.

5.1 Matrix Multiplication and Addition

We compared the performance of both the non-distributed version and the distributed version of the PQ matrix multiplication and addition program against those that use GQ.

1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
Size	**GQ**	**PQ**	**Overhead**	**Size**	**GQ**	**PQ**	**Overhead**
5×5	3.266	3.314	1.5%	5×5	2.010	2.010	0.0%
10×10	3.330	3.366	1.1%	10×10	2.010	2.034	1.2%
20×20	3.552	3.602	1.4%	20×20	2.070	2.094	1.2%
50×50	4.854	4.916	1.3%	50×50	2.514	2.550	1.4%

Table 5.1: Average elapsed execution time (in seconds) for the non-distributed matrix multiplication and addition program.

Specifically, we compared the non-distributed version of PQ matrix multiplication and addition program in Section 4.2.1 (Figure 4.1) with a variant of the original main program in Section 3.2.3 (Figures 3.7 and 3.8) that is executed on a single VM. Table 5.1 shows that over a range of different sized matrices (5×5, 10×10, 20×20, and 50×50) PQ required only from 0% to 1.5% additional time. These overheads of the PQ programs were the results of the process group creations that required RMI interactions with the centralized manager (JRX), the PQ detection activity that took place in JRX, and the additional logging for each thread in the process groups.

1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
Size	**GQ**	**PQ**	**Overhead**	**Size**	**GQ**	**PQ**	**Overhead**
5×5	5.822	5.902	1.4%	5×5	3.326	3.342	0.5%
10×10	5.884	5.978	1.6%	10×10	3.308	3.344	1.1%
20×20	6.088	6.196	1.8%	20×20	3.392	3.426	1.0%
50×50	7.514	7.638	1.7%	50×50	3.860	3.918	1.5%

Table 5.2: Average elapsed execution time (in seconds) for the distributed matrix multiplication and addition program using the same physical machine.

We also compared the elapsed time of the distributed version of the program that uses PQ in Section 4.2.2 (Figures 4.3 and 4.4) with the one that uses GQ in Section 3.2.3 (Figures 3.7 and 3.8). Recall that these programs create a `MMMultiplier` object and a `MMAdder` object on two different VMs that are located on the same physical machine. Table 5.2 shows that over a range of different sized matrices (5×5, 10×10, 20×20, and 50×50) PQ required only from 0.5% to 1.8% additional time.

To observe the performance of PQ vs. GQ further in the distributed environment, we compared the elapsed time of a similar program that is executed on different numbers of VMs such that each VM is created on a different physical host. Table 5.3 presents the different average elapsed times for computing a range of different sized matrices (5×5, 10×10, 20×20, and 50×50) over a range of different numbers of VMs (1, 2, 4, and 8). The results show that PQ required from 0.4% to 4.1% additional time.

3.0GHz uniprocessor							
# of VMs = 1				**# of VMs = 2**			
	Average Elapsed Time				**Average Elapsed Time**		
Size	**GQ**	**PQ**	**Overhead**	**Size**	**GQ**	**PQ**	**Overhead**
5×5	2.966	2.996	1.0%	5×5	3.838	3.878	1.0%
10×10	3.100	3.166	2.1%	10×10	3.994	4.043	1.2%
20×20	3.768	3.836	1.8%	20×20	4.564	4.600	0.8%
50×50	7.294	7.548	3.5%	50×50	7.826	8.150	4.1%
# of VMs = 4				**# of VMs = 8**			
	Average Elapsed Time				**Average Elapsed Time**		
Size	**GQ**	**PQ**	**Overhead**	**Size**	**GQ**	**PQ**	**Overhead**
5×5	5.897	5.922	0.4%	5×5	9.780	9.928	1.5%
10×10	6.098	6.122	0.4%	10×10	9.906	9.976	0.7%
20×20	6.942	7.008	0.9%	20×20	10.722	10.815	0.9%
50×50	11.438	11.750	2.7%	50×50	15.694	16.065	2.4%

Table 5.3: Average elapsed execution time (in seconds) for the distributed matrix multiplication and addition program using different physical machines.

The additional time of the distributed PQ programs over the distributed GQ programs are due to the same factors as mentioned for the non-distributed version. However, the overheads imposed on the distributed programs are slightly higher than those imposed on the non-distributed ones, because, in addition to creating the process groups in the main program (which resides in the main JRVM), process groups with the same names are also created in the JRVMs where the remote objects are created. These additional process group creations require extra RMI interactions with JRX. Furthermore, there are also extra RMI interactions for a process group in each VM to notify JRX about its idleness and, in turn, for JRX to confirm that the process group in each VM is in fact idle. Although there are overheads in our PQ implementation, we can see from Table 5.3 that the percentages of overhead did not increase as the number of VMs increases, which shows the impact of using more VMs in a PQ program is insignificant.

5.2 Barrier Synchronization

This section compares and evaluates the performance of the PQ and GQ versions of the non-distributed and the distributed barrier programs.

For the non-distributed version, the barrier program that uses PQ in Section 4.2.3 (Figure 4.6) is compared with a barrier program that uses GQ. We measured the average elapsed times over a range of different numbers of workers (10, 20, 50, and 100) as well as different numbers of rounds for PQ or GQ registration. Table 5.4 shows that the times for the two versions were always within 3% of each other.

For the distributed version, the barrier program that uses PQ in Section 4.2.4 (Figures 4.7 and 4.8) is compared with a barrier program that uses GQ. Note that this program uses two VMs that are located in the same physical machine. Again, the elapsed times are measured for various numbers of workers (10, 20, 50, and 100) as well as various numbers of rounds for PQ or GQ registration. Table 5.5 shows that the average elapsed times for the two versions were always within 7% of each other.

The performance evaluation for the distributed barrier synchronization program

Number of rounds registered = 3							
1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
10	4.160	4.044	-2.8%	10	2.286	2.298	0.5%
20	4.436	4.394	-0.9%	20	2.610	2.622	0.5%
50	5.102	5.098	-0.1%	50	3.346	3.414	2.0%
100	5.660	5.754	1.7%	100	4.014	4.086	1.8%
Number of rounds registered = 6							
1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
10	4.600	4.584	-0.3%	10	2.516	2.526	0.4%
20	4.614	4.574	-0.9%	20	3.042	3.054	0.4%
50	5.636	5.804	3.0%	50	3.954	4.002	1.2%
100	8.342	8.440	1.2%	100	5.144	5.250	2.1%

Table 5.4: Average elapsed execution time (in seconds) for the non-distributed barrier synchronization program.

is extended further to include different numbers of worker processes that are located in different VMs and different physical machines. The total number of workers varies from 16, 40, and 80 and are distributed equally among different number of VMs (1, 2, 4, and 8). Table 5.6 shows that the average elapsed times for the two programs were within 5% of each other, and they do not change drastically when the number of VMs increases.

Tables 5.4, 5.5, and 5.6 show that the costs of PQ in the barrier programs are not significant. Yet, in some cases, PQ programs have average elapsed times that are lower than those for GQ programs. Table 5.7 shows the average number of RMI calls the distributed barrier synchronization program makes. As we can see, the PQ barrier programs often make fewer RMI calls than the GQ ones. This shows that the GQ program usually requires more interactions between each VM and the centralized

Number of rounds registered = 3							
1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
10	8.102	7.678	-5.2%	10	4.170	3.990	-4.3%
20	8.960	8.702	-2.9%	20	4.752	4.470	-5.9%
50	10.526	10.072	-4.3%	50	6.648	6.380	-4.0%
100	13.080	13.904	6.2%	100	8.882	8.394	-5.5%
Number of rounds registered = 6							
1.4GHz uniprocessor				**2.8GHz dual-processor**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
10	9.226	8.664	-6.1%	10	4.626	4.352	-5.9%
20	10.378	9.734	-6.2%	20	5.698	5.294	-7.1%
50	12.266	12.300	0.3%	50	8.028	7.464	-7.0%
100	16.126	15.766	-2.2%	100	12.088	11.492	-4.9%

Table 5.5: Average elapsed execution time (in seconds) for the distributed barrier synchronization program using the same physical machine.

manager (JRX) when detecting quiescence. In the GQ implementation, these RMI calls are generated when a process is blocked on a `receive` or `P` waiting for a message that is located on another VM. When the process attempts to acquire a "lock" for this message queue, it registers itself as a "thread death" on the VM on which it is located and registers itself as a"thread birth" on the VM on which the message queue for the operation is located. The loggings of these "thread deaths" in one VM might, at times, causes it to notify JRX about its idleness. At this point, the VM on which the message queue is located might have already reported its idleness (before the new "thread birth" is logged there). This causes JRX to sends messages (via RMI) to confirm the idleness of all VMs. However, in the PQ implementation, even though all VMs might have reported to JRX that they are idle, JRX would not perform RMIs to confirm the idleness of all VMs yet, as there is still a "PGIdler" thread for each

3.0GHz uniprocessor							
Number of rounds registered = 3							
# of VMs = 1				**# of VMs = 2**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
16	3.416	3.534	3.5%	16	4.456	4.342	-2.6%
40	4.166	4.188	0.5%	40	5.488	5.334	-2.8%
80	5.282	5.133	-2.8%	80	7.238	7.208	-0.4%
# of VMs = 4				**# of VMs = 8**			
	Average Elapsed Time				**Average Elapsed Time**		
# of workers	**GQ**	**PQ**	**% diff.**	**# of workers**	**GQ**	**PQ**	**% diff.**
16	7.158	6.856	-4.2%	16	25.724	25.826	0.4%
40	9.238	8.778	-5.0%	40	33.674	32.230	-4.3%
80	13.098	12.612	-3.7%	80	41.530	40.453	-2.6%

Table 5.6: Average elapsed execution time (in seconds) for the distributed barrier synchronization program using different physical machines.

3.0GHz uniprocessor					
Number of rounds registered = 3					
# of VMs = 1			**# of VMs = 2**		
	# of RMIs			**# of RMIs**	
# of workers	**GQ**	**PQ**	**# of workers**	**GQ**	**PQ**
16	1172	1023	16	1755	1377
40	3256	3134	40	4447	3760
80	7923	5788	80	9076	8721
# of VMs = 4			**# of VMs = 8**		
	# of RMIs			**# of RMIs**	
# of workers	**GQ**	**PQ**	**# of workers**	**GQ**	**PQ**
16	2350	1614	16	3580	2280
40	6701	5357	40	10930	7633
80	14343	10558	80	24492	17725

Table 5.7: Average number of RMI calls in the distributed barrier synchronization program using different physical machines.

process group executing in JRX to detect partial quiescence. As the VM on which JRX is located has not reported idle yet, JRX would not perform the confirmation phase to verify the idle states of all other VMs, which reduces the number of RMIs as compared to the GQ programs. Since RMIs are costly, the additional RMIs in the GQ programs cover the costs of process group creations and the additional bookkeeping for partial quiescence in the PQ programs.

In the matrix multiplication and addition program, since processes never block or wait for a message, it does not make these additional RMI calls and thus, we did not observe a case when a PQ matrix multiplication and addition program would perform better than a GQ one.

Chapter 6

Conclusion

Termination detection is an important problem in distributed computing. In this thesis, we first discussed the concept and the definition of termination detection in a global sense. Specifically, we presented some useful examples of global quiescence in JR and explained how the detection mechanism is implemented in the language. From the various examples, we can observe that global quiescence can assist programmers in determining the quiescent state of a program automatically. It also enables the programmers to perform different actions such as outputting final results, gathering statistics, and initiating a new phase of activity.

The useful features of an automatic global quiescence detection led us to explore it in a more general scope such that the detection is not limited to detecting the state of *all* processes in a program. The core of this thesis introduced the notion of partial quiescence: detecting when a specified *part* of the program has become quiescent. We based the definition of partial quiescence on the definition of global quiescence and provided linguistic support for it by incorporating it into the JR concurrent programming language. Based on some expository examples, we can see that having such a PQ mechanism can lead to a different style of programming, which in some cases is simpler. The useful features of global quiescence mentioned earlier can also be used when a program has become partially quiescent. Furthermore, we discussed the keys aspects of partial quiescence in JR. The potential nondeterministic behaviors caused by the spontaneous message generation outside of a process group is one problem that could be explored in future work.

This thesis also discussed the implementation of partial quiescence and its performance. We compared the performance of some programs that use partial quiescence with roughly comparable programs that use global quiescence. Although the extra interactions between each VM and the centralized manager when detecting partial quiescence have additional overheads, these overheads are often insignificant, as, typically, the number of VMs and process groups created is not large. Our early results are promising, however, further experience using partial quiescence mechanisms is needed.

Appendix A

Performance of Global Quiescence

This appendix presents the performance results of enabling and disabling GQ implementation in some simple JR programs. An overview of these results is presented in Section 3.3.3. We ran our tests multiple times on various PCs (1.4GH and 2.0GHz uniprocessors; 2.4GHz and 2.8GHz dual-processors) running Linux; specific results varied according to platform, but the overall trends were the same.

A.1 Benchmarks

```
public class main {
  public static void main(String [] args) {
    int n = Integer.parseInt(args[0]);
    int K = Integer.parseInt(args[1]);
    co ((int k = 0; k < K; k++)) p(n);
    //explicit call to terminate the program
    JR.exit(0);
  }
    static op void p(int n) {
      int sum = 0;
      for (int k = 0; k < n; k++) sum += k;
    }
}
```

Figure A.1: A non-distributed JR program – GQ disabled.

```
public class main {
  public static void main(String [] args) {
    int n = Integer.parseInt(args[0]);
    int K = Integer.parseInt(args[1]);
    co ((int k = 0; k < K; k++)) p(n);
  }
  static op void p(int n) {
    int sum = 0;
    for (int k = 0; k < n; k++) sum += k;
  }
}
```

Figure A.2: A non-distributed JR program – GQ enabled.

Figures A.1 and A.2 present JR programs that create their processes using the concurrent invocation statement. Each program generates `K` processes such that each process computes the sum of `n` consecutive numbers. The program in Figure A.1 disables GQ detection and uses an explicit `JR.exit()` method to terminate the program, whereas Figure A.2 terminates through the GQ detection.

Figures A.3 and A.4 are simply distributed versions of the programs in Figures A.1 and A.2. Processes in a remote object `r` are created using the concurrent invocation statement and are placed on different virtual machines and physical machines. The program in Figure A.3 disables GQ detection and uses an explicit `JR.exit()` method to terminate the program, whereas Figure A.4 terminates through the GQ detection. See Section 3.3.3 for a brief description of the GQ enabling and disabling feature.

A.2 Results

Table A.1 compares the average elapsed times of the non-distributed programs that have GQ disabled (Figure A.1) and GQ enabled (Figure A.2) over a range of different numbers of processes (2, 20, 50, and 100). It shows that enabling the GQ implementation requires up to 52.5% more time than disabling it. Table A.2 compares the average elapsed times of the distributed programs that have GQ disabled (Figure

```
public class main {
  public static void main(String [] args) {
    int n = Integer.parseInt(args[0]);
    int K = Integer.parseInt(args[1]);
    int M = Integer.parseInt(args[2]);
    int C = 3;
    vm [] vms = new vm[M];
    remote r [] rs = new remote r[M];
    for (int v = 0; v < M; v++) {
      vms[v] = new vm() on args[C+v];
      rs[v] = new remote r() on vms[v];
    }
    co ((int k = 0; k < K; k++)) rs[k%M].p(n);
    //explicit call to terminate the program
    JR.exit(0);
  }
}

public class r {
  op void p(int n) {
  int sum = 0;
  for (int k = 0; k < n; k++) sum += k;
  }
}
```

Figure A.3: A distributed JR program – GQ disabled.

```
public class main {
  public static void main(String [] args) {
    int n = Integer.parseInt(args[0]);
    int K = Integer.parseInt(args[1]);
    int M = Integer.parseInt(args[2]);
    int C = 3;
    vm [] vms = new vm[M];
    remote r [] rs = new remote r[M];
    for (int v = 0; v < M; v++) {
      vms[v] = new vm() on args[C+v];
      rs[v] = new remote r() on vms[v];
    }
    co ((int k = 0; k < K; k++)) rs[k%M].p(n);
  }
}

public class r {
  ... //same as in the previous figure
}
```

Figure A.4: A distributed JR program – GQ enabled.

2.0GHz uniprocessor			
	Average Elapsed Time		
# of processes	GQ disabled	GQ enabled	Cost
2	2.982	3.020	1.3%
20	3.012	3.516	16.7%
50	3.088	4.203	36.1%
100	3.198	4.878	52.5%

Table A.1: Average elapsed execution time (in seconds) for the simple non-distributed JR program (Figures A.1 and A.2).

2.0GHz uniprocessor							
# of VMs = 1				# of VMs = 2			
	Average Elapsed Time				Average Elapsed Time		
# of proc	GQ disabled	GQ enabled	Cost	# of proc	GQ disabled	GQ enabled	Cost
2	4.355	4.443	2.1%	2	5.443	5.570	2.3%
20	5.540	6.163	11.3%	20	6.673	7.340	10.0%
50	7.423	8.510	14.6%	50	9.297	10.470	12.6%
100	10.573	12.557	18.8%	100	13.710	15.910	16.0%
# of VMs = 4				# of VMs = 8			
	Average Elapsed Time				Average Elapsed Time		
# of proc	GQ disabled	GQ enabled	Cost	# of proc	GQ disabled	GQ enabled	Cost
2	7.380	7.547	2.3%	2	11.423	12.147	6.3%
20	9.103	10.253	12.6%	20	13.500	15.423	14.2%
50	11.937	14.350	20.2%	50	16.650	20.397	22.5%
100	17.780	21.153	19.0%	100	23.923	30.067	25.7%

Table A.2: Average elapsed execution time (in seconds) for the simple distributed JR program (Figures A.3 and A.4).

A.3) and GQ enabled (Figure A.4) over a range of different numbers of processes (2, 20, 50, and 100) as well as different numbers of VMs (1, 2, 4, and 8). It shows that the GQ implementation requires up to 25.7% more time. Tables A.1 and A.2 show that the cost of GQ implementation in JR is quite high as the numbers of processes and VMs increase. (See Section 3.3.3 for an explanation of the overheads.)

Bibliography

[1] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Pub. Co., 1993.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programmming Language*. Addison-Wesley Professional, 2000.

[3] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *PODC '82: Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 157–164, New York, NY, USA, 1982. ACM Press.

[4] Mani Chandy and Jayadev Misra. An example of stepwise refinement of distributed programs: quiescence detection. *ACM Transactions on Programming Languages and Systems*, 8(3):326–343, 1986.

[5] E. W. Dijkstra and C. S. Scholten. Termination detection for disffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[6] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, 1980.

[7] Jean-Michael Helary, Claude Jard, Noël Plouzeau, and Michel Raynal. Detection of stable properties in distributed applications. In *PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 125–136, New York, NY, USA, 1987. ACM Press.

[8] *The Java ThreadGroup Class*. `http://stat.wvu.edu/~dchilko/tut/java/threads/threadgroup.html`.

[9] *JDK 1.4.2 Documentation*, 2003. `http://java.sun.com/j2se/1.4.2/docs/index.html`.

[10] Aaron W. Keen, Tingjian Ge, Justin T. Maris, and Ronald A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, 26(3):578–608, 2004.

[11] Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC: generic Ada reusable library for interpartition communication. In *TRI-Ada '95: Proceedings*

of the Conference on TRI-Ada '95, pages 263–269, New York, NY, USA, 1995. ACM Press.

[12] L. Liang, S. T. Chanson, and G. W. Neufeld. Process groups and group communications: classifications and requirements. *IEEE Computer*, 23(2):56–66, 1990.

[13] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.

[14] Billy Yan-Kit Man, Hiu Ning (Angela) Chan, Andrew J. Gallagher, Appu S. Goundan, Aaron W. Keen, and Ronald A. Olsson. Toward a definition and linguistic support for partial quiescence. submitted for publication.

[15] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *The Journal of Systems and Software*, 43(3):pp 207–221, 1998.

[16] Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(4):195–200, 1989.

[17] Jayadev Misra. Detecting termination of distributed computations using markers. In *PODC '83: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 290–294, New York, NY, USA, 1983. ACM Press.

[18] Ronald A. Olsson and Aaron W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java.* Kluwer Academic Publishers, Boston, Massachusetts, 2004.

[19] Amitabh B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, 1993.

[20] Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, 1993.

[21] Jeffrey S. Vetter and Bronis D. de Supinski. Dynamic software testing of MPI applications with Umpire. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California.

[22] Michael P. Wellman and William E. Walsh. Distributed quiescence detection in multiagent negotiation. In *ICMAS '00: Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, page 317, Washington, DC, USA, 2000. IEEE Computer Society.