CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

A Definition of and Linguistic Support for Partial Quiescence

Billy Yan-Kit Man[1], Hiu Ning (Angela) Chan[1], Andrew J. Gallagher[1], Appu S. Goundan[1], Aaron W. Keen[2], and Ronald A. Olsson[1,*,†]

[1] *Department of Computer Science, University of California, Davis, CA 95616 USA*
[2] *Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407 USA*

Note to editor and reviewers:

This paper is based on our earlier paper (Reference [1]), which appeared in the Euro-Par 2006 conference. The present paper extends that earlier paper with additional details about our overall approach, more examples, further experimental results, and additional discussion.

*Correspondence to: Prof. Ronald A. Olsson, Department of Computer Science, University of California, Davis, CA 95616 USA
†E-mail: {many,chanhn,gallagha,goundan,olsson}@cs.ucdavis.edu; akeen@csc.calpoly.edu
Contract/grant sponsor: National Science Foundation, equipment grant; contract/grant number: EIA-0224469

Received Sometime 2007

SUMMARY

The global quiescence of a distributed computation (or distributed termination detection) is an important problem. Some concurrent programming languages and systems provide global quiescence detection as a built-in feature so that programmers do not need to write special synchronization code to detect quiescence. This paper introduces *partial quiescence* (PQ), which generalizes quiescence detection to a specified part of a distributed computation. Partial quiescence is useful, for example, when two independent concurrent computations that both rely on global quiescence need to be combined into a single program. The paper describes how we have designed and implemented a PQ mechanism within an experimental version of the JR concurrent programming language, and have gained experience with several representative applications. Our early results are promising qualitatively and quantitatively.

1. Introduction

In distributed programs, multiple processes cooperate to perform some task and communicate via messages to exchange information. One important, and well-studied, problem for such programs is to determine when the program's computation has completed, i.e., it has terminated normally or deadlocked. This *quiescence* problem is challenging because each process has only local information, but to solve the problem requires information about all processes (i.e., global state information). More formally, global quiescence (GQ) is defined as the state in which each process has terminated or deadlocked and there are no messages in the communication channels [2]. *Quiescence detection*, then, is the mechanism used to detect such a state in a distributed system. Many solutions to this problem have been proposed (e.g., [3, 4, 5, 6, 7, 8]).

Some programming languages and systems provide GQ detection as a built-in feature. That is, programmers do not need to write special synchronization code to detect quiescence. Instead, they can focus on writing application code. When quiescence is reached, the program can perform various actions such as simply terminating the program, outputting final results, gathering statistics from the overall computation, or initiating a new phase of the program, which might involve a new, corresponding phase of quiescence detection.

Although useful, GQ is limited to dealing with the state of *all* processes in a program. A more general, but more difficult to detect, property would define when a specified *part* of the program has become quiescent. For example, suppose we have two programs that use GQ and we want to combine them into a single program in which we want to perform different actions when each part of it becomes quiescent. This motivation led us to explore *partial quiescence* (PQ).

This paper proposes possible ways of defining PQ. It then discusses the particular definition selected and implemented in an experimental version of the JR concurrent programming language. (JR extends Java [9] with a richer concurrency model [10, 11, 12].) The definition of PQ and its implementation in JR apply to truly distributed programs. The paper also shows how PQ leads to a different programming style for some problems. We compare the performance of using PQ detection and GQ detection. PQ might be a useful feature for other languages and libraries that define process or thread groups, as many do, and especially useful for those languages and libraries that already provide GQ detection.

Our work involves detecting quiescence (GQ or PQ) *dynamically*, i.e., during the actual execution of a concurrent program. An alternative approach involves *statically* determining various properties of concurrent programs, e.g., determining whether a concurrent program is deadlock-free. For example, [13] describes how to statically determine whether a given Ada program is deadlock-free and [14, 15, 16] describe how to statically verify that a program in notations such as process calculus with communication channels or timed automata with shared variables is *partial-deadlock* free. Partial deadlock means that a specified part of the program deadlocks. For example, [14] places restrictions on how processes communicate over channels and shows that the part of the program that uses "reliable" communication channels does not deadlock. The key difference between our approach and the static approaches, besides when the checking is performed, is that our approach treats quiescence as a normal part of program execution; the program itself is aware of its own quiescence and can react to quiescence as it desires.

The rest of this paper is organized as follows. Section 2 provides background on the general definition of GQ and how it has been incorporated as a built-in feature in some programming languages and systems; it describes how GQ is defined and implemented in JR and presents examples of programs that use GQ. Section 3 discusses different ways of defining PQ. Section 4 discusses the definition of PQ we chose to provide in JR and gives examples of programs that use PQ. Section 5 describes our implementation and Section 6 discusses its performance. Section 7 reflects on our work. Finally, Section 8 concludes. Further details appear in [17].

2. Background

This section presents background on the general definition of global quiescence in context of previous work on distributed termination detection. It then describes how GQ has been incorporated as a built-in feature in some programming languages and systems; it focuses on how GQ is defined and implemented in JR and presents example JR programs that use GQ.

2.1. Distributed Termination Detection (DTD)

As noted in Section 1, detecting the termination of a distributed computation is an important and challenging problem, which has been well-studied over the years. A nice survey [2] describes DTD as follows. 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 purposes 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 *active* if it is working on some computation or processing activation messages addressed to it. A process is *passive* if it is waiting for an activation message or termination. All processes in the system behave based on the following rules [2][7][8]:

1. Activation messages can be generated only by active processes.
2. An active process may change its state to passive at any time.
3. A passive process may change its state to active only if it receives an activation message.

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 such a state, i.e., all processes are passive and no activation messages are in transit, then the system is *quiescent*. Or equivalently:

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

Quiescence detection is defined as the mechanism used to detect the state in which there are no messages in transit and all processes are waiting [2]. This definition generalizes that of DTD to both detecting termination as well as deadlock: i.e., sensing when the system is in a state from which it can no longer continue.

The quiescence detection algorithm is conceptually (and often in practice) a separate activity that runs concurrently with, 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.

Two main categories of DTD algorithms, as classified in [2], are wave (or network topology) algorithms (e.g., [4, 3, 5, 8, 6]) and credit distribution and recovery algorithms (e.g., [7]).

A wave algorithm for DTD makes use of the system's topology for detecting quiescence. An example of this category is the termination detection algorithm for "diffusing computations" [4], which imposes a spanning tree upon the computation (A similar view is taken by the probe-echo algorithm for network topology [18].) 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. A process also needs to maintain 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. Thus, 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 [4, 2, 6, 19].

A credit distribution and recovery algorithm for DTD begins with the the system having a credit of total value one. The termination detection distributes this credit amongst active processes and activation messages [7]. 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.

2.2. Tools and Systems with Support for Termination Detection

In some languages and systems (e.g., Ada [20], Java [9], MPI [21], and Pthreads [22]), programs that reach a deadlock state wait indefinitely for the user to terminate them manually. However, in some cases, tools can assist in such detection. For example, Umpire [23] and MPI-CHECK 2.0 [24] detect deadlocks for MPI programs.

Some other programming languages or systems provide GQ detection as a built-in feature. GARLIC [25] extends Ada 95 with distributed programming features; it detects termination based on the algorithm proposed in [26]. JR [10, 11], SR [27, 28], and Charm [8] allow a quiescent program to output final results, gather statistics from the overall computation, or simply terminate the program. In this regard, JR and Charm are similar: when a program quiesces, it can initiate new computation, for which the quiescence feature can be used again. SR's quiescence feature is not as powerful: it is intended only for the program to clean up and terminate, and programs cannot use quiescence repeatedly.

2.2.1. Brief Overview of JR

Since this paper deals with adding partial quiescence for JR, this section gives a very brief overview of the JR concurrent programming language. JR extends Java [9] with a richer concurrency model [10, 11, 12], based on that of SR [27, 28]. Of specific interest here is that JR extends Java with synchronous and asynchronous message passing (of which semaphores are a special case).

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.

JR, as mentioned previously, provides a global quiescence feature. The next section illustrates this feature with several examples.

2.2.2. Example JR Programs Using GQ

The program in Figures 1 and 2 (from [11]) performs matrix multiplication. Its `MMMain` class reads in two N × N matrices, instantiates a `MMMultiplier` object, and registers the operation

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

Figure 1. Matrix multiplication using GQ – `MMMain` class.

```
public class MMMultiplier {
  double [][] A, B, C;  int N; // A, B, and C are NxN
  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 2. Matrix multiplication using GQ – `MMMultiplier` class.

`done` as the quiescence operation. (Technically, the registration needs to be within a `try/catch` block.) Its `MMMultiplier` class contains the processes that perform the actual computation. These processes begin execution after `MMMultiplier`'s constructor completes its execution. GQ is used to determine when these `compute` processes have finished their tasks. Once GQ has been detected, the registered operation `done` is invoked and its code outputs the resulting matrix. Without GQ detection, the programmer would need to write additional code to determine when the computation has terminated.

If no GQ operation is registered, then the program simply terminates when it quiesces. The quiescence operation can initiate new activity and can re-register the GQ operation (either the same or different operation), which will be invoked when the newly initiated activity quiesces.

```
public class AQ {
  private static final int N = 20; // number of workers
  private static op void bag(double, double, double, double);
  private static op void result(double);
  private static final double Epsilon = 0.001; // convergence test

  private static double f(double x) {
    return Math.pow(x,3.0); // some function to integrate
  }

  private static double area = 0.0;

  // administrator and worker processes appear in next figure

  public static void main(String [] args) {
    // register done as the quiescence operation
    JR.registerQuiescenceAction(done);
  }
  private static op void done() {
    System.out.println("computed area = "+area);
  }
}
```

Figure 3. Adaptive quadrature using bag-of-tasks using GQ.

Of course, GQ is not required to achieve the overall outcome from the matrix multiplication program. Instead, as each `compute` process completes its computation, it can atomically increment a shared counter and test whether that counter equals the number of processes created. If so, that process (the last one to finish) can print out the entire result matrix. Similarly, other synchronization (barrier-like) strategies can be used. For example, an additional coordinator process can delay until it has received a notification of completion from each `compute` process. The coordinator can then print out the entire result matrix. However, these approaches, require additional programming effort.

A more interesting example is adaptive quadrature using the bag-of-tasks paradigm. The program in Figures 3 and 4 (from [11]) computes the approximate area under a curve over some interval. Each task in the bag represents a subinterval over which to compute the area. The bag is represented as a shared operation, whose invocations correspond to the tasks and on which workers compete to receive invocations. Each worker takes a task and approximates the area under the curve for the subinterval in two ways: using a single rectangle and using the sum of two rectangles. If the two areas differ by less than some predefined value `Epsilon`, then the worker returns the second area as its result. Otherwise, more work is required to compute

```
private static process administrator {
  double part;
  double l = 0.0, r = 4.0;
  send bag(l,r,f(l),f(r)); // put initial task (whole interval) in bag
  while (true) {
    receive result(part);  // receive back area for subinterval
    area += part;          // and add it into overall area
  }
}

private static process worker( (int i = 1; i <= N; i++) ) {
  double a, b, m, fofa, fofb, fofm;
  double larea, rarea, tarea, tarea2, diff;
  while (true) {
    receive bag(a,b,fofa,fofb);
    m = (a+b)/2;  fofm = f(m);
    // compute tarea, larea, rarea, and tarea2 using trapezoidal rule
    tarea = (b - a) * (fofa + fofb) / 2.0;
    larea = (m - a) * (fofa + fofm) / 2.0;
    rarea = (b - m) * (fofm + fofb) / 2.0;
    tarea2 = larea + rarea;
    diff  = Math.abs(tarea - tarea2);
    if (diff <= Epsilon) { // diff small enough -- send back area
      send result(tarea2);
    }
    else { // diff > Epsilon // diff too large -- split into 2 tasks
      send bag(a, m, fofa, fofm);
      send bag(m, b, fofm, fofb);
    }
  }
}
```

Figure 4. Adaptive quadrature using bag-of-tasks using GQ – `administrator` and `worker` processes.

the area in this subinterval, so the worker puts back into the bag two subproblems. In either case, the worker continues by taking another task out of the bag.

GQ is used to determine when the computation has finished. Once GQ has been detected, the registered operation `done` is invoked and its code outputs the computed area. Without GQ detection, the programmer would need to write additional code to determine when the computation has terminated: the bag is empty *and* all workers have blocked (waiting to take a task from the bag). That termination condition is not so simple to express directly in a program.

One approach is to modify the code to use a credit distribution and recovery algorithm (Section 2.1). The administrator keeps track of total part of the interval for which it needs a response, $WidthLeft$. When $WidthLeft$ becomes zero, the administrator can terminate the workers with a special message. When workers send the area back to the administrator, they also include the interval width in the message, which the administrator subtracts from $WidthLeft$. (This approach ignores potential problems with roundoff errors in the computations involving $WidthLeft$.)

A second approach is to have the administrator count the number of outstanding tasks remaining to be computed, $TasksLeft$. When $TasksLeft$ becomes zero, the administrator can terminate the workers. This approach requires extra messages. A worker sends a new message to the administrator just before the worker splits an interval. When the administrator receives such a message, it increments $TasksLeft$. When the administrator receives a result message, it decrements $TasksLeft$. In a sense, this solution can be viewed as a diffusing computation (Section 2.1). However, the style of computation is different. Each worker may perform any part of the overall computation instead of a specific part within some hierarchical decomposition. Moreover, the administrator acts as a central site of "parental responsibility" [4], instead of having that role distributed.

2.2.3. Implementation of Quiescence in JR

The implementation of quiescence in JR† 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 (Section 2.2.1).

2.2.3.1. JR Virtual Machines and Centralized Manager As the information contained in a standard Java VM is not sufficient by itself to implement quiescence detection, JR creates its own virtual machine (JRVM), which provides a thin layer over the standard Java VM. JR virtual machines (JRVMs) are created during run-time. A JRVM, then, is able to record information, such as the number of active processes and messages that are in transit, that is essential in determining the quiescent state of the program. Having JRVMs allows the JR implementation to run on any standard Java VM.

As mentioned in Section 2.1, the distributed termination detection problem is a challenging one, because it requires global state information for the system. JRX is the centralized manager of JR. It possesses the global state information of the system. JRX is created when a JR program executes; there is one JRX per each JR program's execution. JRX 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 a JRVM on that host. 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

†JR's implementation of quiescence extends SR's implementation of quiescence to provide the additional functionality described in Section 2.2.

detection as well as other services, such as executing an explicit exit from the program code, which needs to shut down all JRVMs.

2.2.3.2. 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 an "idler" thread to assist with quiescence detection.

Figure 5 depicts the interaction between JRX and all JRVMs, which is as follows. 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 JRVM. Whenever this thread is blocked or terminates, a "thread death" is logged in the same manner. When a JRVM can make no further progress (i.e., all of its threads have terminated or are waiting to receive a message), it sends an idle message to JRX. This message contains the number of messages this JRVM has sent to each other JRVM and the number of messages this JRVM has received from each other JRVM. 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 VM $JRVM_a$, the number of sends from $JRVM_a$ to each other JRVM, $JRVM_b$, matches the number of receives from $JRVM_a$ reported by $JRVM_b$, If so, then the system is globally quiescent. Additional information about the GQ implementation appears in [17].

3. Definition of Partial Quiescence (PQ)

Although GQ is useful, it requires 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 GQ 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 PQ 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 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.

Because PQ deals with the interactions of groups of processes, it is, in general, more difficult to detect. This definition fits well if the process group is "closed" [29], 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" [29], 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

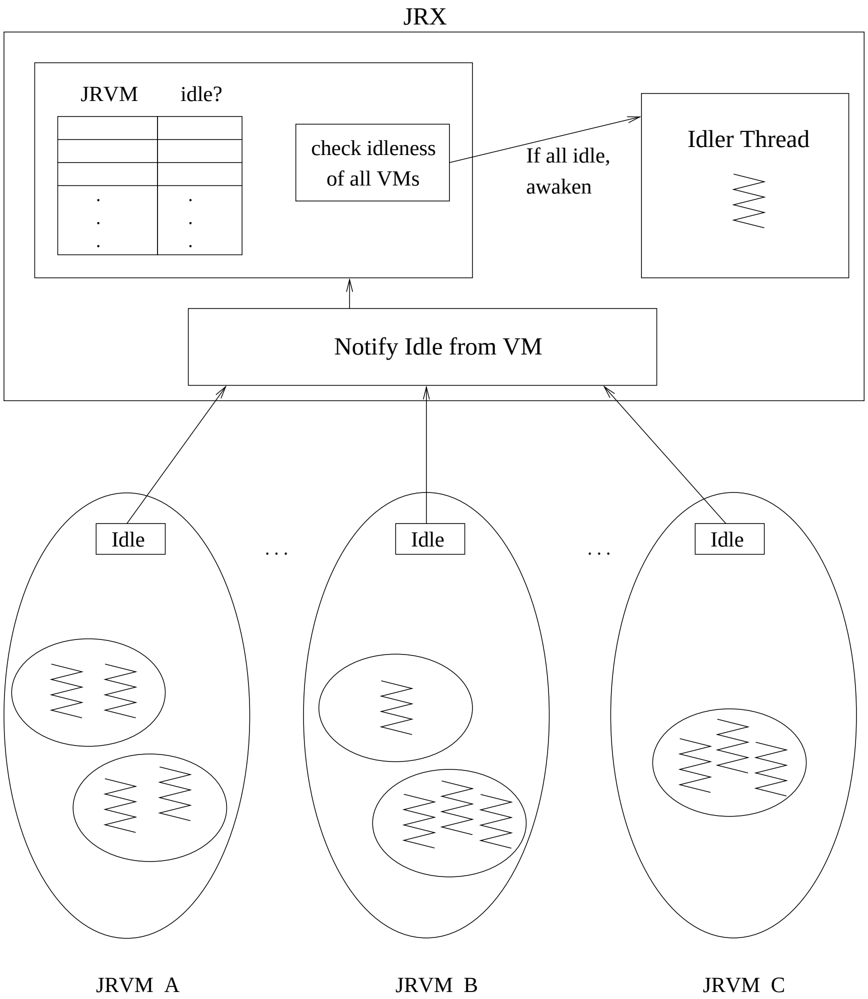

Figure 5. JRX and JRVMs interaction for global quiescence detection.

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 more restrictive definition of partial quiescence, namely one that modifies clause (1) from the previous 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. JR Extended for Partial Quiescence

We have extended JR to support PQ. Now, JR programs can define groups of related processes and can register, for each process group, a *partial quiescence operation*. This section begins with examples to illustrate how PQ in the extended JR works, discusses key aspects of the various mechanisms, and presents additional mechanisms to facilitate programming with PQ.

4.1. Expository Examples of PQ in JR

4.1.1. Multiple Matrix Multiplications

The main program in Figure 6 shows how to use PQ to perform two simultaneous matrix multiplications. A nice attribute of our PQ approach is that the same `MMMultiplier` class from Figure 2 works here. The main program creates two process groups, one for each matrix multiplication. It uses `JR.changeCreationGroup` to specify the group in which newly created processes will be placed for each new matrix computation. (There is one default process group.) The main program then registers the PQ operation for each process group. When either group quiesces, its PQ operation will be invoked and that code outputs the results.

In contrast, consider a variant of the original main program in Figure 1 that starts two matrix multiplications and that uses GQ. It would wait for *both* computations to finish before outputting the result from either.

In Figure 6, two process groups might quiesce at about the same time, in which case the outputs from their quiescence operation might be interleaved. Their outputs can be serialized by deleting the present code for `done1` and `done2` (but keeping their `op` declarations) and adding the code in Figure 7 to the end of the `main` method. This code uses JR's multi-way

```
public class MMMain {
  private static MMMultiplier m1, m2;
  public static void main(String [] args) {
    double [][] A1, B1, A2, B2;  int N;  // A1, B1, A2, B2 are NxN
    // read in NxN arrays A1, B1, A2, B2
    ...
    ProcessGroup m_g1 = new ProcessGroup("Multiply Group1");
    ProcessGroup m_g2 = new ProcessGroup("Multiply Group2");
    JR.changeCreationGroup(m_g1); // processes within m1 will be in m_g1
    m1 = new MMMultiplier(A1, B1, N);
    JR.changeCreationGroup(m_g2); // processes within m2 will be in m_g2
    m2 = new MMMultiplier(A2, B2, N);
    // register partial quiescence operation for each process group
    JR.registerPartialQuiescenceAction(m_g1, done1);
    JR.registerPartialQuiescenceAction(m_g2, done2);
  }
  private static op void done1() { m1.print(); }
  private static op void done2() { m2.print(); }
}
```

Figure 6. Multiple matrix multiplications using PQ – `MMMain` class.

```
for (int i = 0; i < 2; i++) {
  inni void done1() {m1.print();}
  []   void done2() {m2.print();}
}
```

Figure 7. Code to serialize output from the multiple matrix multiplications.

receive statement (`inni`) to wait for an invocation of either of the PQ operations; it services one at a time, thus serializing their outputs.

4.1.2. Barrier Synchronization

PQ, as noted earlier for GQ, allows JR programs to re-register a quiescence operation. Consider the program in Figure 8 (from [11]). It shows a group of worker processes synchronizing their iterations via a barrier, implemented with semaphores.[‡] The program also contains a

[‡]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 8. Barrier synchronization using semaphores.

coordinator process that controls when workers begin their next iteration. The program uses an array of semaphores, `proceed` (one for each worker), rather than a single semaphore, to prevent a fast worker from "stealing" the message intended for a slow worker. With a single semaphore, a slow worker might be context switched after `V(done)` and before the `P(proceed)`, which would allow a fast worker to finish its iteration and get past the `P(proceed)`. (See [11] for details.)

This program can be rewritten using PQ and fewer semaphores, as shown in Figure 9. Worker processes no longer need to tell the coordinator that they are done (via the `done` semaphore); instead PQ will detect that. The role of the coordinator is no longer performed by a separate process. It is now the PQ operation that is invoked when all worker processes quiesce. Also, the `proceed` array of semaphores is now replaced with a single `proceed` semaphore: a fast worker cannot overtake a slow worker since all workers must quiesce before the coordinator operation is invoked and tells any worker it may proceed.

```
public class Barrier {
  private static final int N = 10; // number of workers
  private static sem proceed;
  private static ProcessGroup WG;
  static { // all worker processes will belong to process group WG.
    WG = new ProcessGroup("Worker Group");
    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's now invoked on PQ.
  private static op void coordinator() {
    for (int w = 0; w < N; w++) { V(proceed); }
    if (...) // iterations remain
      JR.registerPartialQuiescenceAction(WG, coordinator);// re-reg PQ op.
  }
  public static void main(String [] args) {
    JR.registerPartialQuiescenceAction(WG, coordinator);// register PQ op.
  }
}
```

Figure 9. Barrier synchronization using partial quiescence.

4.1.3. Distributed Barrier Synchronization

The barrier synchronization example (Section 4.1.2) can be extended to be a distributed program. Figures 10 and 11 present such a program, in which instances of `BW` ("barrier worker") objects are created so that worker processes are located on different VMs. As shown in Figure 11, after each worker process completes one iteration of its work task, it blocks on a capability, which references the `proceed` semaphore located remotely in the main program (Figure 10). The program uses partial quiescence to detect that worker processes in all VMs belonging to the `WG` group are blocked. When the program detects partial quiescence, the coordinator in the main program performs the same actions as in Figure 9 to awaken all workers on different VMs.

```
public class Main {
  // number of worker processes in each BW object
  private static final int N = 10;
  // two BW objects:
  private static remote BW w1, w2;
  private static sem proceed;
  private static ProcessGroup WG;

  public static void main(String [] args) {
    // 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);

    // create virtual machines vm1 and vm2,
    // and create a remote worker on each
    vm vm1 = new vm();  vm vm2 = new vm();
    w1 = new remote BW(N, proceed) on vm1;
    w2 = new remote BW(N, proceed) on vm2;

    JR.registerPartialQuiescenceAction(WG, coordinator);
  }

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

Figure 10. Barrier synchronization with distributed workers using PQ – `Main` class.

4.2. Key Aspects of Partial Quiescence

As seen in the examples in the previous section, process groups allow the programmer to specify parts of the program for separate PQ detection. The names of process groups, specified by the string argument to the `ProcessGroup` constructor, are in a global namespace. For example, in a multi-VM program (Section 4.1.3), processes created in process group `"WG"` on two different VMs are in the same process group. Similarly, there is one PQ operation for a process group,

```
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 11. Barrier synchronization with distributed workers using PQ – `BW` class.

even if the process group is distributed across different VMs. The programmer can create a process group specific to a VM by using a per-VM unique identifier in the name.

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. For example, suppose that the main program in Figure 6 registered its PQ operations before instantiating the `MMMultiplier` objects. Then, PQ detection would detect that the group has quiesced, which would not be likely what the programmer wanted.

Just as for a GQ operation, a PQ operation can start up new activity and can re-register another (or the same) PQ operation. As seen in Figure 9, for example, the re-registration of the PQ operation occurs after the worker processes are awakened to avoid the startup problem. Note that even if all the workers execute and quiesce before the PQ operation is re-registered, quiescence detection will detect that (because detection for the group began anew when the PQ operation was invoked) and invoke the PQ operation when it is re-registered.

The precise definition of PQ for JR differs slightly from that given in Section 3. The reason is that in JR a message is sent to an operation. Operations can be shared and serviced by

Prepared using cpeauth.cls

processes that might belong to different process groups. Exactly which processes services which operations is determined as the program executes (not statically) [30]. The PQ definition for JR, therefore, modifies clause (1) of the earlier definition to be more specific with respect to serviceablility: "(1) there are no messages in the system from group A in transit to *an operation serviceable by a process in* group A".

How the definition of PQ deals with messages in transit can introduce additional nondeterminism into a program. 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 paper (or other practical examples we have written so far). It remains to be seen whether such nondeterminism is a problem in further practice.

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. Additional PQ Programming Features

The extended JR provides four additional PQ features for more complicated programming situations.

First, the program can disable or enable PQ detection features during execution. This feature allows the programmer to pause the detection of partial quiescence of a specific process group. It is useful in some programs when waiting for some desired processes to be added to the process group. (Section 7 mentions another example.)

Second, an optional argument to the `ProcessGroup` constructor can specify the number of processes expected in the group. Quiescence of the group occurs when that number of processes have terminated or deadlocked. This feature can be used to avoid the "startup problem" discussed in Section 4.2 even if a PQ operation is registered before the desired processes have been created in the process group. Of course, this feature is not always useful because it requires advance knowledge of the number of processes in the process group. It also might reduce information hiding. For example, suppose the `MMMultiplier` class in Section 4.1.1 chose to create, say, `N/2` processes, each computing a strip of results (instead of `N` $\times$ `N` processes). That information would now need to be known outside of that class.

Third, a process can change its process group in the middle of execution. 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. Another use of this feature is when a process wants to move to another process group when some conditions are met. As a concrete example, consider a program modeling a tennis tournament. The program represents each tennis player as a process and the players (processes) in each round of competition form a distinct process group. Each tennis player initially belongs to a process group representing the first round of competition. When all players have finished playing the first round, 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 process group for the second round; as

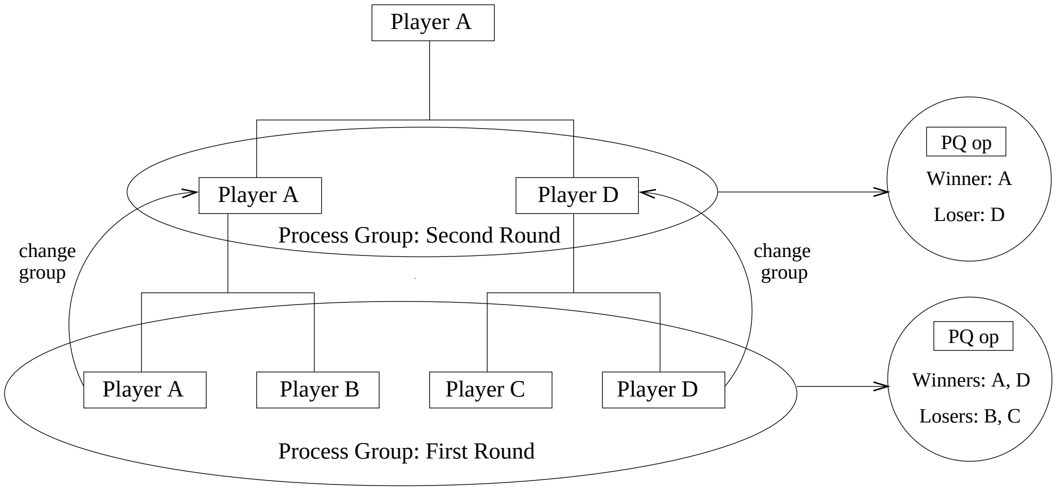

Figure 12. Process groups structure in a tennis tournament program.

before, statistics are output 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 12 illustrates the overall structure of process groups in this tennis tournament example. The code for this program appears in [17].

Fourth, 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, the `ProcessGroup` constructor takes as an optional argument an array of child process groups. As a concrete example, consider a program modeling a bus stop. The program represents buses and passengers as processes. The program simulates a scenario in which there are some buses waiting in line to pick up some passengers who are also waiting in line. The program uses a process group for each bus to represent the passengers who board it and a hierarchical process group for the bus stop to represent all the buses. PQ provides a relatively straightforward solution for this simulation problem to determine critical activities:

- when *each* bus has finished picking up its passengers, i.e., partial quiescence of each bus process group, and
- when *all* buses have left the bus stop, i.e., partial quiescence of the bus stop process group, which contains each bus process group as a child.

The code for this program appears in [17].

5. Implementation of Partial Quiescence

Our current implementation of PQ is built on JR version 1.00061, which is based on Java 1.4. It has also been ported to JR version 2.00002, which is based on Java 1.5. The implementation of PQ adapts the centralized manager implementation of GQ described in Section 2.2.3. The implementation of PQ for closed process groups (Section 3) could follow the GQ implementation rather directly, but with message counts specific to process groups. However, our general implementation of PQ is different to account for open process groups.

5.1. JRThread and Process Group

In 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's `Thread` class. `JRThread` is used instead of Java `Thread` so that a JR process can change its process group in the middle of its execution; 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. So, `JRThread` contains a (modifiable) attribute to represent the current group to which the thread belongs. It also contains a (modifiable) creation group attribute, which represents the process group for the JR processes that a `JRThread` creates.

Java also provides thread groups to organize a group of threads for management and security purposes. However, Java's `ThreadGroup` class does not provide the necessary support for partial quiescence detection in JR. The JR 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 2.2.3.1, that are used in GQ detection).

5.2. The Centralized Manager and PQ Detection

As in the implementation of GQ (Section 2.2.3), 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 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 4.2.

Figure 13 depicts the interaction between the process groups in each VM and the centralized manager, JRX, when detecting partial quiescence, which is as follows. 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 4.2.

6. Performance

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 data are from our implementation of extended JR that is built on JR version 1.00061, which is based on Java 1.4.)

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

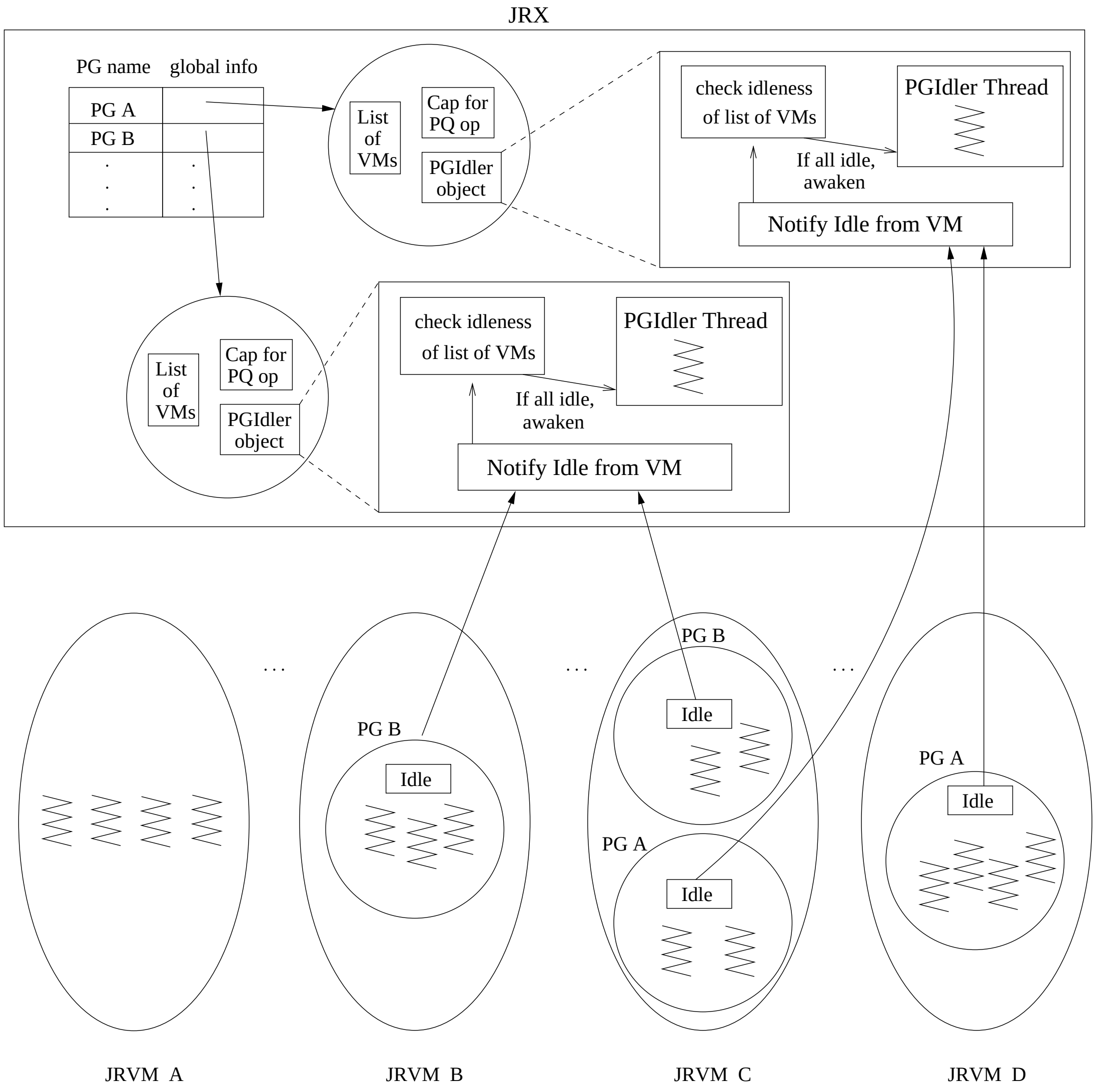

Figure 13. JRX and JRVMs interaction for partial quiescence detection.

Table I. Average elapsed execution time (in seconds) for the non-distributed matrix multiplication program.

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%

dual-processors) running Linux. Specific results, of course, varied according to platform, but the overall trends were the same.

6.1. Multiple Matrix Multiplications

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

Specifically, we compared the PQ matrix multiplication program in Section 4.1.1 (Figure 6) with a variant of the original GQ matrix multiplication program in Section 2.2.2 (Figure 1) that starts two matrix multiplications. Both programs are non-distributed: each executed on a single VM. Table I 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.

We also compared the elapsed time of the distributed versions of these matrix multiplication programs. These programs each create two `MMMultiplier` objects on two different VMs that are located on the same physical machine. Table II 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 programs similar to the previous two, but executed on different numbers of VMs, with each VM created on a different physical host. Table III 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.

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

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

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 III. Average elapsed execution time (in seconds) for the distributed matrix multiplication program using different physical machines.

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 IV. Average elapsed execution time (in seconds) for the non-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%

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 III 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.

6.2. Barrier Synchronization

For the non-distributed barrier, we compare the barrier program that uses PQ in Section 4.1.2 (Figure 9) with a similar 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

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

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%

numbers of rounds for PQ or GQ registration. Table IV shows that the times for the two versions were always within 3% of each other. We also compared the PQ barrier program in Section 4.1.2 (Figure 9) with the original (GQ) semaphore program in Figure 8. Over the same ranges of workers, the PQ version took 4-10% more time than the GQ program.

6.3. Distributed Barrier Synchronization

For the distributed barrier, we compare the barrier program that uses PQ in Section 4.1.3 (Figures 10 and 11) with a barrier program that uses GQ. Each of these programs uses two VMs that are located on 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 V shows that the average elapsed times for the two versions were always within 7% of each other.

We further extended the performance evaluation for the distributed barrier synchronization program to include different numbers of worker processes that are located on different VMs and different physical machines. The total number of workers varies from 16, 40, and 80 and

Prepared using cpeauth.cls

Table VI. 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**			
	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%

are distributed equally among different number of VMs (1, 2, 4, and 8). Table VI 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 IV, V, and VI 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 VII 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 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

Table VII. Average number of RMI calls in 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

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 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 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 program would perform better than a GQ one.

7. Discussion

Our definition of PQ seems to work well in the examples seen in Section 4.1. In the matrix multiplication example (Section 4.1.1), the process groups are closed. In the barrier examples (Sections 4.1.2 and 4.1.3), the process groups are open: 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 messages are 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.

Also in the barrier examples, 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. Similar comments apply if the program requires multiple barriers.

The barrier examples also illustrate one aspect of our work on PQ that is not entirely satisfying. The code assumes that the workers do not quiesce for any other reason, such as waiting for a message while interacting with some other worker (or any other process) for purposes unrelated to the barrier. If they do, then quiescence could be detected early (incorrectly, based on the programmer's intent). The program can be changed to avoid such problems, e.g., by disabling and enabling PQ (Section 4.3) around the non-barrier interactions. However, that seems cumbersome. We plan to investigate this issue in future work.

Our notion of process groups is not new. Beside Java thread groups, many other systems (such as Horus, ISIS, MPI, and PVM) provide support to organize groups of processes [29]. 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.

Our definition of PQ (or even the general definition of GQ) is aimed at message passing languages and systems. As such, it seems that PQ could be adapted to other message passing notations, such as Ada [20], Java with RMI [9], or MPI [21]. The details of how to implement PQ for one of these other systems will vary according to the specific system and its implementation, but the general ideas described in Section 5, such as using idle messages, should carry over. PQ (or GQ) could also be applied to distributed applications that use message oriented middleware for their communications. In theory, PQ (or even GQ) could be adapted for systems where synchronization is based more on shared variables such as OpenMP [31] or systems that effect conditional critical regions. In practice, however, that would likely entail more significant changes to overall design of the implementation. Whether such changes would be feasible is unclear. For example, consider a program that busy waits waiting for a global variable to equal some value. The straightforward implementation of PQ (or GQ) might instrument the busy wait loop with additional code to see whether PQ has been reached. Such code could adversely affect the program's performance.

Our definitions of PQ (or GQ) assume that the underlying distributed system is free from faults. For example, we assume that nodes do not fail and that messages will be delivered. If the system is susceptible to such faults, then our definition (and corresponding implementation) will not work. An interesting question is how fault tolerance might be combined with quiescence detection in such a way that is useful for the programmer. That remains an avenue for future work.

8. Conclusion

This paper introduced the notion of partial quiescence and showed how it can be incorporated into a programming language, in our case, JR. Having such a PQ mechanism can lead to a different style of programming, which in some cases can lead to programs that might be considered simpler, for example, as seen in the barrier example in Section 4.1.2. This paper also discussed the implementation of PQ and its performance, which differs only slightly from GQ's performance. We feel that our early results are promising both qualitatively and quantitatively. However, further experience is needed with using PQ mechanisms to see whether this style of programming is a good one and whether other applications might benefit from using PQ mechanisms. Also, it would be interesting to investigate whether PQ can be applied to other concurrency mechanisms.

Acknowledgements

The anonymous referees for the Euro-Par 2006 conference provided thoughtful comments on an earlier version of this paper (Reference [1]). The anonymous referees for CCP&E offered very helpful suggestions too.

REFERENCES

1. B. Y. Man, H. N. (Angela) Chan, A. J. Gallagher, A. S. Goundan, A. W. Keen, and R. A. Olsson. Toward a definition of and linguistic support for partial quiescence. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, number 4128 in Lecture Notes in Computer Science, pages 655–665, Dresden, Germany, August 2006. Springer–Verlag.
2. J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *The Journal of Systems and Software*, 43(3):pp 207–221, 1998.
3. N. Francez. Distributed termination. *ACM Trans. Programming Languages and Systems*, 2(1):42–55, 1980.
4. E. W. Dijkstra and C. S. Scholten. Termination detection for disffusing computations. *Inform. Process. Lett.*, 11(1):1–4, 1980.
5. Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst.*, 15(1):1–35, 1993.
6. 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.
7. Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *Inf. Process. Lett.*, 30(4):195–200, 1989.
8. Amitabh B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Department of Computer Science, University of Illinois, Urbana-Champaign, 1993.
9. J. Gosling, B. Joy, G. Steele, and G Bracha. *The Java Language Specification*. Java Series. Sun Microsystems, 2005. ISBN 0-321-24678-0.
10. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 578–608, May 2004.
11. R. A. Olsson and A. W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer Academic Publishers, Inc., 2004.
12. JR distribution. `http://www.cs.ucdavis.edu/~olsson/research/jr/`.
13. S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of 1990 International Conference on Parallel Processing*, pages II.78–II.87, University Park PA, 1990.
14. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.
15. U. Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000.
16. K. Okano, S. Hattori, A. Yamamoto, T. Higashino, and K. Taniguchi. Specification of real-time systems using a timed automata model with shared variables and verification of partial-deadlock freeness. In *ICPP Workshop*, pages 576–581, 1999.
17. Billy Yan-Kit Man. The design and implementation of partial quiescence in a concurrent programming language. Master's thesis, University of California, Davis, Department of Computer Science, March 2006. `http://www.cs.ucdavis.edu/~olsson/students/`.
18. E. J. H. Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, July 1982.
19. Steffen Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, number 4128 in Lecture Notes in Computer Science, pages 615–624, Dresden, Germany, August 2006. Springer–Verlag.
20. Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995.
21. Message Passing Interface Forum. `http://www.mpi-forum.org/`.
22. Posix threads programming. `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`.
23. J. S. Vetter and B. D. de Supinski. Dynamic software testing of MPI applications with Umpire. Technical report, Lawrence Livermore National Laboratory, 2000.
24. G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.
25. Y. Kermarrec, L. Pautet, and S. 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.

26. J. Helary, C. Jard, N. Plouzeau, and M. 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, 1987.
27. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
28. G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.
29. L. Liang, S. T. Chanson, and G. W. Neufeld. Process groups and group communications: classifications and requirements. *IEEE Computer*, 23(2):56–66, 1990.
30. R. A. Olsson, G. D. Benson, T. Ge, and A. W. Keen. Fairness in shared invocation servicing. *Computer Languages, Systems and Structures*, 28(4):327–351, December 2002.
31. OpenMP Application Program Interface. `http://www.openmp.org/`.