Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Solving the Santa Claus Problem: a Comparison of Various Concurrent Programming Techniques

Jason HURT and Jan B. PEDERSEN
School of Computer Science, University of Nevada
jleehurt@gmail.com, matt@cs.unlv.edu

Abstract. The Santa Claus problem provides an excellent exercise in concurrent programming and can be used to show the simplicity or complexity of solving problems using a particular set of concurrency mechanisms and offers a comparison of these mechanisms. Shared-memory constructs, message passing constructs, and process oriented constructs will be used in various programming languages to solve the Santa Claus Problem. Various concurrency mechanisms available will be examined and analyzed as to their respective strengths and weaknesses.

Keywords. concurrency, distributed memory, shared memory, process-oriented programming.

Introduction

Concurrent or parallel computing has always been an area of interest in the computer science community. Historically computer clusters and multiprocessor computers that provide hardware environments for parallel applications were expensive and used only in highly specialized development environments such as weather prediction, environmental modeling, and nuclear simulation. With the popularity of the Internet, distributed applications have become more widely used. *PlanetLab* [1] and *BOINC* [2] which powers *SETI* [3] are two examples of this. There are an increasing number of developer APIs that have a distributed architecture and applications that are implemented as a set of disparate, functional components connected via web services [4,5]. In addition, due to the physical limitations of silicon, chip makers such as Intel and AMD have begun moving towards multi-processor/multi-core architectures as opposed to increasing the clock speed of a single CPU, and consequently parallel computing is moving further into mainstream application development [6]. In the past, the increase in clock speed for single CPU systems meant that programmers got "free" speedup in their applications with every new generation of CPU. Now, programmers have to find ways to utilize multiple-CPU architectures in order to improve application performance. Simply moving an old sequential program to a multi-core architecture will not utilize the full potential of the processor. A number of ways to write parallel programs will be examined here, as well as the advantages and disadvantages of each method.

First, we will look at the popular shared memory model (threads). Next, a message passing architecture known as MPI will be explored, and lastly, a process oriented approach using the JCSP library for Java is examined. The metrics for comparing these models will be the readability, writability, and reliability of programs written using each model. In order to compare the metric properties of the models, a simple description of a problem is explored that lends itself naturally to a concurrent solution, the Santa Claus Problem, introduced by

John Trono in 1994 [7]. In addition to the Polyphonic C# [8] and Ada [9] versions, solutions to the problem have been written in Java using Java threads, Groovy using Java threads, C# using the .NET threading mechanism, C using pthreads [10], C using MPI, and Java using JCSP.

1. Problem Definition

The Santa Claus problem is stated in [7] as follows: "Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He then performs one of two indivisible actions. If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation. If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshaling the reindeer or elves into a group must not be done by Santa."

1.1. Correctness

As noted by Ben-ari [9], John Trono's solution using semaphores is incorrect because it incorrectly "assumes that a process released from waiting on a semaphore will necessarily be scheduled for execution." In a concurrent context, correctness refers to a system that is free from the possibility of deadlocks, livelocks, and race conditions. The system in question may have bugs, but they will not be due to the concurrency of the system. Note that the original problem description is up for interpretation, so in order to test the correctness of our solutions we will add a set of messages that can be reported by Santa, the Elves, and the Reindeer. Each message is an indication of the state of a particular entity: Santa, an Elf, or a Reindeer. In this way a visual representation of the states of the entities at runtime is available for debugging and testing purposes. We will impose a partial ordering on the union of three sets of messages: Santa, Elf, and Reindeer messages. The union set of these messages, which we will call S, is the set of all messages that the program can possibly report. In the real world, complex systems may be controlled as opposed to printing messages to a console. Keep in mind that the ordering of these messages may be important in order to not cause chaos to the system under control, and therefore the correctness of a solution would be necessary in order to trust a system is free from bugs due to concurrency. Let SR be the subset of S that includes the set of messages that the Reindeer can print and that Santa can print while dealing with the Reindeer. We define here a message ordering on SR, and by the transitivity property we get a complete ordering on SR:

1. Reindeer $< id >$: on holiday … wish you were here, :)
2. Reindeer $< id >$: back from holiday … ready for work, :(
3. Santa: Ho-ho-ho … the reindeer are back!
4. Santa: harnessing reindeer $< id >$ …
5. Santa: mush mush …
6. Reindeer $< id >$: delivering toys
7. Santa: woah … we're back home!
8. Reindeer: $< id >$: all toys delivered
9. Santa: un-harnessing reindeer $< id >$ …

In addition to the above, all Reindeer must report 2 before Santa can report 3, Santa must report 4 to all Reindeer before reporting 5, and all Reindeer must report 6 before Santa can report 7.

Let SL be the subset of S that includes the set of messages that the Elves can print and that Santa can print while dealing with the Elves. We define here a message ordering on SL, and by the transitivity property we get a complete ordering on SL:

1. Elf $< id >$: need to consult santa, :(
2. Santa: Ho-ho-ho … some elves are here!
3. Santa: hello elf $< id >$ …
4. Elf $< id >$: about these toys … ???
5. Santa: consulting with elves …
6. Santa: OK, all done - thanks!
7. Elf $< id >$: OK … we'll build it
8. Santa: goodbye elf $< id >$ …
9. Elf $< id >$: working, :)

In addition, three Elves must report 2 before Santa can report 3 and the same three Elves must report 5 before Santa can report 6.

Note that the intersection of SR and SL is the empty set. In addition to the message ordering, the Reindeer have priority over the elves, and only three Elves at a time may consult with Santa. Moreover, freedom from deadlock and livelock are necessary; no process may halt its execution indefinitely and the states of the entities must proceed as per the problem description.

2. Shared Memory

Shared memory solutions are often implemented in the form of threads. The thread model is a shared memory model that provides a way for a program to split itself into two or more tasks of execution. Since the threads operate on shared data, the programmer must be careful not to modify a piece of data that another thread may be reading or modifying at the same time. Constructs such as barriers, locks, semaphores, mutexes, and monitors can be used for this purpose and for two or more threads to synchronize at a certai n point.

Threads on a single processor system will be swapped in and out very fast by the underlying operating system and scheduled for execution in an interleaved manner, giving the appearance of parallelism, while those in multi-processor or multi-core systems will actually run in parallel (i.e., true parallelism). In order to ensure message ordering, threads must have a way to synchronize (i.e., pause at specific points of execution).

Shared memory models do not offer an easy way to derive a correctness proof for the solutions; heavy testing is usually done on the system to build confidence, but often possible paths of execution are missed in testing environments. Formally, to prove correctness, all possible interleavings must be considered. To make matters worse, the scheduling mechanisms of the Java Virtual Machine (JVM) [11], Common Language Runtime (CLR) [12], Linux, Solaris, and Windows are complex and do not offer much guarantee in terms of when a thread is interrupted and when it will get its next burst of CPU time. Thread libraries are available in many popular programming languages. The focus here is on C, Java, and C#.

2.1. Java

In Java the threading mechanism is tightly coupled to the language. Every object has an implicit monitor that can be used both as a locking mechanism and a wait/notify mechanism. In addition, method signatures may contain the $synchronized$ keyword, which tell the JVM to use the current object's implicit monitor for locking to facilitate transactional method calls, or methods that will ensure the atomicity of their instruction sets with respect to other methods of the same object that contain the $synchronized$ keyword.

A thread library is available for Java that is based on the threading mechanism of the Java Virtual Machine. Sun [13] added this library in Java 1.5 to help ease the pains of writing multi-threaded code. Of particular interest with respect to thread synchronization is the addition of a *CyclicBarrier* [14] type whose job it is to provide a re-entrant barrier for multiple threads to synchronize on. Here is code from the Santa thread that shows the use of two $CyclicBarrier$ objects to synchronize with Elf threads:

```
// notify elves of "OK" message
try {
    m_elfBarrierTwo.await();
}
catch (InterruptedException e) {
    e.printStackTrace();
}
catch (BrokenBarrierException e) {
    e.printStackTrace();
}

// wait for elves to say "ok we'll build it"
try {
    m_elfBarrierThree.await();
}
catch ...  // exception handling logic
```

And the corresponding Elf code:

```
// wait for santa to say "ok all done"
try {
    m_elfBarrierTwo.await();
}
catch ...  // exception handling logic

System.out.println("Elf " + this + ": OK ... we'll build it\n");

// notify santa of "ok" message
try {
    m_elfBarrierThree.await();
}
catch ...  // exception handling logic
```

Note the two checked exceptions that the *await()* method of a $CyclicBarrier$ can throw, $InterruptedException$ and $BrokenBarrierException$. Proper handling of these exceptions requires additional effort if the program needs to recover from errors due to single points of failure in a single thread. Handling exceptions from failed calls to the thread library can become increasingly difficult as an application grows in size. In our Java solution if either of the CyclicBarrier exceptions are thrown in a single thread, a stack trace will be printed but the other threads will not be made aware of this failure, they will continue operating. However, if the system depended on the thread that failed it will eventually come to a state of deadlock. In this case, if the Santa thread fails, the program eventually comes to a halt, but if a single Elf thread fails and it is not in the waiting queue at the time, the program will continue operating minus the failed Elf. If the desired behavior is to restore threads to an acceptable state then additional thread logic would have been needed.

Although $CyclicBarrier$ has eliminated the need for shared state when synchronizing, a wait and notify mechanism is needed when either a group of Elves or a group of Reindeer is ready to see Santa. Due to the asynchronous nature of the $Object.notify()$ method, in the

Java version Santa must query the Elves and Reindeer in case of missed notifications. This leads to code that is more coupled among multiple threads than we would prefer:

```
if (elfQueue.size() % 3 == 0) {
    synchronized (m_santaLock) {
        m_santaLock.notify();
        notifiedCount++;
    }
}
```

In addition, a JVM implementation is permitted to perform spurious wake-ups in order to remove threads from wait sets and thus enable resumption without explicit instructions to do so [11]. This requires all calls to $Object.wait()$ to have extra logic to check if the condition the thread was waiting on is true or it was a spurious wakeup by the JVM. This hurts the readability of the wait/notify mechanism. It also requires both the notifying thread and waiting thread to share some state so that the notifier can set a condition before notifying, and the waiting thread can check the condition in the case of a wakeup.

2.2. Pthreads

Pthreads are a POSIX standard for threads. Pthread libraries are available for a number of operating systems including Windows, Linux, and Solaris. Pthreads provide developers access to mutexes, semaphores, and monitors for thread synchronization. For the Santa Claus problem a custom partial barrier implementation similar to $CyclicBarrier$ in Java was implemented in order to improve the readability of the solution and because the library does not have a built-in partial barrier implementation. For the synchronization between Santa and the Reindeer, the pthread libraries' $join$ function is sufficient, but the $join$ function cannot be used for Santa to wait for a group of elves because Santa cannot foresee which three elves will need to consult with him. A simple implementation of a barrier can be defined with the following:

```
void AwaitBarrier(barrier_t *barrier) {
    /* ensure that only one thread can execute this method at a time */
    pthread_mutex_lock(&barrier->mutex);
    /* if this barrier has reached its total count */
    if (++barrier->currentCount == barrier->count) {
        /* notify all threads waiting on the barrier's condition */
        pthread_cond_broadcast(&barrier->condition);
    }
    else { /* at least one thread has not entered this barrier */
        /* wait on the barrier's condition */
        pthread_cond_wait(&barrier->condition, &barrier->mutex);
    }
    /* allow other threads to enter the body of AwaitBarrier */
    pthread_mutex_unlock(&barrier->mutex);
}
```

A mutex and a condition variable from the library are used in order to synchronize a group of threads. Each thread will wait on a condition variable, and the last thread to synchronize will notify each of the threads waiting on the condition variable. A mutex is used to ensure no two threads will ever be in the body of $AwaitBarrier$ at a time. This will ensure that exactly $N - 1$ threads in the group will wait for the condition, and only the N^{th} thread will notify the group of waiting threads. Here is a portion of the Santa thread code that uses the barriers to ensure message ordering:

```
/* notify elves of "OK" message */
AwaitBarrier(&elfBarrierTwo);

/* wait for elves to say "ok we'll build it" */
AwaitBarrier(&elfBarrierThree);
```

Here is the corresponding code for the Elf threads:

```
/* wait for santa to say "ok all done" */
AwaitBarrier(&elfBarrierTwo);

printf("Elf %d: OK ... we'll build it\n", elfId);

/* notify santa of "ok" message */
AwaitBarrier(&elfBarrierThree);
```

In addition to thread synchronization to ensure message ordering, the program must also satisfy the "three elves at a time" constraint. The Elf threads share state in the form of a counter so that every third Elf in the waiting room will go and wake Santa. When one thread signals another the signal will be lost if the thread on the receiving end of the signal was not currently waiting on the condition. This can be handled a number of ways. Here, as in the Java version, Santa queries the Elves to see if they are ready in case Santa was with the Reindeer and at the time an Elf notification was sent. A different implementation could remove the query and instead have a waiting group of Elves send notifications to Santa until he responds to one. An Elf thread uses the following code to wake Santa:

```
pthread_mutex_lock(&santaMutex);
pthread_cond_signal(&santaCondition);
pthread_mutex_unlock(&santaMutex);
```

Missed notifications are recorded via use of a shared memory counter. Both the Santa and the Elf code must be aware of the fact that Santa needs a way to query for a group of three elves, which leads to tightly coupled code between the Santa and Elf threads.

2.3. C#

.NET provides a threading library that is tied to the memory model of the Common Language Runtime (CLR). The memory model for the CLR [12] provides an additional layer of abstraction over the underlying operating system's memory model, including its own thread scheduling mechanism. Unlike Java, C# objects do not have implicit monitor associations. In the C# version we use the provided threading library to write a barrier for thread synchronization. A monitor object is used for locking and unlocking of code blocks:

```
public void Await() {
    // ensure only one thread is in this method at a time
    Monitor.Enter(_awaitLock);

    // if all threads have arrived at the barrier
    if (++_lockValue == _count) {
        // notify the earlier threads waiting on this barrier
        Monitor.PulseAll(awaitLock);
    } else { // at least one thread has not entered the barrier
        // wait for a notification
        Monitor.Wait(awaitLock);
    }
```

```
        // allow another thread to call Await()
        Monitor.Exit(awaitLock);
    }
```

Here is a sample usage of the barriers in the Santa code:

```
// harness reindeer
deerBarrierOne.Await();

// wait for all deer to say "delivering toys"
deerBarrierTwo.Await();
```

And the corresponding reindeer code:

```
// wait for santa to harness us
Santa.deerBarrierOne.Await();

m_Form.WriteMessage("Reindeer " + this.ToString()
                   + ": delivering toys\n");

// notify santa of "delivering toys"
Santa.deerBarrierTwo.Await();
```

Note that in both the C and the C# barrier implementation a wait/notify mechanism is used along with a shared variable to keep track of state. The wait/notify behaves like an asynchronous messaging system, in that a thread may notify even if there is a thread that is not waiting yet, so lost notifications are possible. Here we use a monitor to avoid this in our barrier implementation, in the C version a mutex and a condition variable are used for the same purpose. The monitor ensures that only the N^{th} thread will call $notify$ after the other $N-1$ threads are waiting for notification.

2.4. Groovy

Groovy [15] is touted as “an agile dynamic language for the Java Platform”. Groovy compiles to Java byte code and adds a number of features not present in Java, most notably dynamic typing and closures. Closures are anonymous chunks of code that may take arguments, return a value, and reference and use variables declared in its surrounding scope. Although Groovy does not expand upon Java’s threading mechanism, the Groovy version is more readable than the Java version due to the use of closures to remove the replication of exception handling code. Thread related calls are wrapped in closures, allowing them to be passed to methods or other closures which use a different closure for the exception handling logic. Here, a method named $performOperation$ is used which takes a closure as an argument that is the operation to be executed and uses a member variable closure which handles the exception handling logic. In this case we are using a closure named $simpleExceptionHandler$ which simply prints the reason for the exception to the console. More complicated exception handlers can be defined in a similiar manner. Here is the Groovy version of the above Java Elf code:

```
// wait for santa to say "ok all done"
barrierAwait(m_elfBarrierTwo)

println("Elf " + id + ": OK ... we'll build it\n")

// notify santa of "ok" message
barrierAwait(m_elfBarrierThree)
```

2.5. Differences

Due to possible spurious wake-ups by a JVM, whenever a call to $Object.wait()$ is made in the Java and Groovy versions, condition checking code, typically in the form of a $while$ loop, must be added in order to differentiate between a call to $Object.notify()$ and an unexpected spurious wakeup. Spurious wake-ups are cited in the Java Language Specification [11], this may be due to a bug in the JVM or a flaw in operating systems such as older versions of Linux that used LinuxThreads [16]. In the description of the Native POSIX Thread Library for Linux [17] spurious wake-ups and "the misuse of signals to implement synchronization primitives" add to the problems of the Linux Threads library, concluding that "delivering signals is a very heavy-handed approach to ensure synchronization."

With regards to error handling, the Java and Groovy versions must do something with the checked exceptions that a $CyclicBarrier$ can throw. These problems make the Java code less readable and even more so if the program must attempt to recover from these types of errors. Often the simplest, and therefore easiest to read, solution is to restart all threads in the program. Thanks to closures in Groovy some of the readability can be recovered. In the C and C# versions, the respective library call errors are ignored, as these languages do not require the handling of error conditions at compile time.

In Java, threads are more tightly coupled to the language than C#. Aside from this, Java's spurious wake-ups, C#'s lack of a built-in partial barrier, and the lack of checked exceptions in C#, the Java and C# differences are syntactic. One example is Java's use of the $synchronized$ keyword which is comparable to C#'s explicit $Monitor$ type. An advantage of the pthread library is the ability to distinguish between the condition and mutex primitives. This decouples the locking mechanism from the notification mechanism.

2.6. Ada

A solution to the problem was proposed in Ada [9] that used two constructs that were added to Ada 95 [18]. One was a construct that enables a group of processes to be collected and then released together, similar to the barriers that were presented above. The second is a rendezvous that enables a process to wait simultaneously for more than one possible event. This can be done in Java with $Object.wait()$, C# with $Monitor.Wait()$, and C with a mutex and condition variable.

2.7. Polyphonic C#

A solution which claims to be easier to reason about than the Ada solution has also been written using Polyphonic C# [8]. Here a concept known as a chord is used, which associates a code body with a set of method headers, and the body of a chord can only run once all of the methods in the set have been called. This solution shows how a wait/notify mechanism that can prioritize notifications can be implemented with shared memory if chords are available to the programmer.

2.8. Haskell STM

A solution to the problem has been written in Haskell using Software Transactional Memory, or STM [19]. STM provides atomic transactions to protect against race hazards and choice operators for prioritizing actions. The concept of a gate is used for marshalling the Reindeer and Elves. The choice operator allows for a simple way to give the Reindeer priority over a group of Elves. This solution shows how these constructs can help to de-couple and modularize multi-threaded code.

2.9. Summary

The readability, writability, and reliability of shared memory solutions is crippled by the amount of time it can take in order to search a large codebase for all threads that may possibly cause race conditions, deadlock, livelock and starvation. Preventing race conditions means ensuring the atomicity of a set of one or more instructions and usually involves the use of mutexes, semaphores, and monitors. In each of the shared memory solutions, locks are used to ensure data integrity between each thread. These locks ensure the atomicity of one or more instructions in a thread. This can slow down performance when many threads are waiting on the same lock, in addition to adding complexity to the code. When a Reindeer or Elf is added or removed from a queue, it must happen as an atomic transaction, and in cases where the logic that follows depends on the size of a queue, this logic must be included in the atomic transaction. Preventing deadlock and livelock involves examining all possible instruction interleavings that a program can generate, a daunting task. In addition, the more state that threads share the harder refactoring becomes. Constraints such as "three elves at a time" or "Reindeer have priority over Elves" require the use of wait/notify and shared counter variables. For example, a shared counter variable is used between the Reindeer threads because Santa must have the ability to query the Reindeer to see if they are ready to deliver toys after consulting with a group of elves. A wait/notify mechanism is also used by the Reindeer and Elves to knock on Santas door. Thus refactoring the Elf or Reindeer threads requires knowledge of the Santa thread. Worse, refactoring the Santa thread requires knowledge of the Elf and Reindeer threads. In small programs such as the Santa Claus solution this is manageable, but mutli-threaded code can become entangled in a larger application. The modification of this code is time consuming and error prone, and there is no quick and simple way to check if a code change has introduced concurrency bugs into the code.

Verifying the correctness of a multi-threaded application is also complicated. Attempts at examining all possible instruction interleavings have been made [20] and shown to be feasible under certain conditions. "Given N threads, each with a local store of size L, and the threads communicate via a shared global store of size G, if each thread is finite-state (without a stack), the naive model checking algorithm requires $O(G * L_n)$ space, but if each thread has a stack, the general model checking problem is undecidable." Successful thread verification has been shown feasible when threads are largely independent and loosely coupled [21], requiring only $O(n * G * (G + L))$ space.

In addition to the complexity of using shared state and the various problems with wait-/notify, the thread scheduling mechanisms introduce non-determinism into various measures of thread execution. These include when and for how long a thread will get CPU time, and when and for how long it will get interrupted and wait for CPU time.

3. Distributed Memory

MPI [22] is one of the more popular libraries designed for taking advantage of a distributed memory architecture, such as a cluster, or a shared memory multiprocessor architecture with a large amount of processors, such as those found in supercomputers. Here, an application's N separate tasks are separated into N processes and appropriate code for each process is written. In the case of MPI, the runtime will ensure the application behaves like a distributed memory model regardless of the underlying hardware. In MPI, groups of processes can be formed at runtime, and a $MPI\_Barrier$ method can be used to synchronize a particular group of processes. Synchronous receives can be used in place of wait/notify thus removing the need to handle the more complicated asynchronous logic. These techniques are used in the MPI solution to the Santa Claus problem in order for the Santa, Elf, and Reindeer processes to synchronize with each other, pausing at various points of execution. Notice how the same barrier is used for two different synchronization points without an explicit method call to reset the barrier. The barrier will implicitly reset itself after the required number of processes has reached the barrier. Here is part of Santa process code that uses a $MPI\_Barrier()$ to synchronize with the Reindeer:

```
// wait for all reindeer to say "delivering toys" message
mpiReturnValue = MPI_Barrier(commSantaReindeer);
CHECK_MPI_ERROR(globalRank, mpiReturnValue);
printf("Santa: woah ... we're back home!\n");
```

Here is part of the Reindeer process code that uses a $MPI\_Barrier()$ to synchronize with Santa:

```
// wait for santa to harness us
mpiReturnValue = MPI_Barrier(commSantaReindeer);
CHECK_MPI_ERROR(globalRank, mpiReturnValue);
printf("Reindeer %d: delivering toys\n", id);
```

Due to the lack of shared memory, solutions to problems in MPI typically have more processes running than a C, C# or Java program would have threads. In the MPI solution to the Santa Claus Problem, there is a separate process for a Reindeer queue for when the Reindeer come back from vacation, and a separate process for an Elf queue for when the Elves get confused and join the waiting room. Instead of using shared memory synchronized data, each process gets data, performs operations on that data, and passes data on to one or more other processes. The same piece of data is never operated on by more than one process at a time, thus eliminating the need for data protection.

3.1. Erlang

There are other solutions to the problem written in Erlang [23,24]. Erlang is a functional language with built-in message passing semantics. Although Erlang processes can share memory via use of the *ets* [25] module, the solutions here use message passing semantics. Similiar to the MPI version, these solutions show the lack of shared state can simplify concurrent programming.

3.2. Summary

The lack of shared state improves the readability of MPI beyond that of shared memory models because it will not allow two or more processes to modify the same piece of data at a time, allowing programmers to focus on one process. Understanding the implications of the use of asynchronous sends and/or receives, however, can be even more challenging than the wait/notify problem for shared memory. With a wait/notify, only the notifying thread behaves asynchronously, while a waiting thread will block until it receives a notify. With the use of $MPI\_IRecv()$, a receive will not block, but can be probed later in the code using $MPI\_Probe()$. In our solution we chose to use only synchronous messages due to the constraints of the problem.

Writing distributed memory code is also made simpler because of the lack of shared state. As long as the corresponding receives and sends in a process are handled properly, MPI code is easier to refactor than thread code. An example in the Santa Claus problem is the way that the three elves at a time constraint is handled. A separate Elf queue process handles this constraint, decoupling the Santa and Elf code further than the shared memory solutions.

For distributed systems, reliability can be hard to measure. The time to send data between nodes on the network can vary greatly, making it difficult to determine whether or not a node has gone down or is just taking a long time to respond. Peer to peer systems are more reliable because of the lack of specialized nodes, but for many applications the idea of every node executing the same code does not fit the problem. In this case, there are requirements that dictate one Santa process, nine Reindeer processes, and ten Elf processes. On Shared Memory Multiprocessor (SMP) machines, reliability is greatly enhanced by eliminating the network. One difficulty that exists in both SMP and distributed environments is the inability to verify asynchronous message behavior. Another problem with asynchronous messaging is that the channel buffers can become full and then start behaving like synchronous channels, causing errors that can only be seen in unfortunate runtime scenarios. Tools such as SPIN [26] are available for model checking the correctness of MPI programs. Traditionally, SPIN has been used for checking only deterministic MPI primitives, but recently work has been done to use the tool for checking non-deterministic primitives [27].

4. Process Oriented

Yet another way to write parallel applications is by using a process oriented architecture, based on the ideas of CSP [28]. Communicating Sequential Processes, or CSP, is a process algebra used to describe interactions in concurrent systems [29]. Relationships between processes and the way they communicate with their environment can be described using CSPs process algebraic operators. Parallelism, synchronization, deterministic and nondeterministic choices, timeouts, interrupts, and other operators are used to express complex process descriptions in CSP. Languages such as occam [30] are based on CSP and there are libraries that provide support for the CSP model for many other languages. Here we focus on one such library for Java, JCSP [31]. The JCSP library is open source and is implemented using Java threads, so it is portable across JVM implementations. Similar to MPI, the system is viewed as a set of independent processes that communicate via channels. There is no concept of shared memory, even if the underlying library is implemented in this way. Channels can be shared by many processes. In the JCSP version, synchronous messages and the library's built-in barrier type are used to ensure message ordering. In addition, we implemented a barrier that will synchronize Santa and a group of three elves using two many-to-one channels to enforce the "three Elves at a time" constraint. The $MyBarrier$ class is implemented using two shared channels, and will read, in succession, two sets of messages for each process on the writing end of the barrier:

```
class MyBarrier implements CSProcess {
    private int count;
    private ChannelInput inA, inB;

    public MyBarrier(int n, ChannelInput inA, ChannelInput inB) {
        this.count = n;
        this.inA = inA;
        this.inB = inB;
    }

    public void run() {
        while (true) {
            for (int i = 0; i < count; i++) {
                inA.read();
            }
            for (int i = 0; i < count; i++) {
                inB.read();
            }
        }
        }
    }
```

The corresponding *Sync* class represents only the writing end of the barrier. Note that only when all members of the barrier have sent their first message will a process start to send its second message to the reading end of the barrier:

```
class Sync implements CSProcess {
    private SharedChannelOutput outA, outB;

    public Sync(SharedChannelOutput outA, SharedChannelOutput outB) {
        this.outA = outA;
        this.outB = outB;
    }

    public void run() {
        outA.write(Boolean.TRUE);
        outB.write(Boolean.TRUE);
    }
}
```

Here is part of the Santa code that uses the barrier:

```
// wait for Elves to say "about these toys"
new Sync(outSantaElvesA, outSantaElvesB).run();
outReport.write("Santa: consulting with Elves ...\n");
```

And the corresponding Elf code:

```
outReport.write("\t\t\tElf: " + id + ": about these toys ... ???\n");
// notify Santa of "about these toys"
new Sync(outSantaElvesA, outSantaElvesB).run();
```

One part of occam that is missing from JCSP is extended rendezvous, an addition to standard occam available in KRoC [32]. Although long standing, these extensions are to some extent experimental. This is a way to force a process A to wait for another process B to do some work before returning from a blocking send or receive. This allows two processes to synchronize without having to explicitly send/receive another message, which is what is done in the JCSP version in replace of an extended rendezvous. The Reindeer waits to be unharnessed with a blocking receive:

```
// Reindeer waits to be unharnessed
inFromSanta.read();
```

For each Reindeer, Santa outputs the unharness message and does a blocking send to the Reindeer to unharness the Reindeer:

```
outReport.write("Santa: un-harnessing reindeer " + id + " ...\n");
// unharness this Reindeer
channelsSantaReindeer[id - 1].out().write(0);
```

In order to ensure priority of the Reindeer over the Elves a construct known as an alternation is used. An alternation enables a process to wait passively for and choose between a number of guarded events, and in this case the guarded events are a notification to Santa from a Reindeer or an Elf. Priority can be given to a guarded event an alternation will choose, and in this case priority is given to the Reindeer notification. In this way, if both the Reindeer and a group of Elves are ready at the same time, Santa will handle the Reindeer first.

Table 1. Program line counts.

	SM				DM	PO
	C#	C	Java	Groovy	MPI	JCSP
Total	642	420	564	315	352	315
Synchronization/Communication	48	49	46	46	34	27
Prevent Race Condition	14	8	8	8	N/A	N/A
Exception/Error Handling	35	0	177	18	41	0
Custom Barrier Implementation	42	35	N/A	N/A	N/A	55
GUI	145	N/A	N/A	N/A	N/A	N/A

SM = Shared Memory, DM = Distributed Memory, PO = Process Oriented

4.1. Summary

Assuming familiarity with the JCSP library, the readability of the JCSP version is better than the shared memory Java solution due to the lack of shared state. Similar to MPI, the code for a process can be looked at without worrying about other processes which might possibly access the same instance data. One advantage that JCSP has over MPI is the ability to specify one-to-one and many-to-one channels on which to send and receive messages. This ensures that a data read will only happen when the sending process has the writing end of a channel, and that a data write will only happen when the receiving process has a reading end of the channel. With MPI, the transportation mechanism for sending and receiving data is abstracted away from the programmer, allowing any process that wishes to communicate with the system to do so. Thus, there is no easy way to ensure message integrity in MPI.

A process oriented application can be refactored easily. The way a process interfaces with the rest of the system is readily available by looking at the input and output channels associated with a process. This allows the programmer to focus on a processes input and output channels, that is, the data a process sends and receives, and not on any of the data it reads or modifies. Again, JCSP's use of channels improves refactorability over the MPI version. A channel's reading and writing ends must be explicitly referenced in order for communication to happen over the channel. This makes it clearer to the programmer which processes are communicating.

In addition, the mapping between JCSP and CSP allows JCSP code to be formally reasoned about using the process algebra. A tool called FDR [33] exists which is a model checker for CSP. The CSP that corresponds to a piece of occam or JCSP code can be run through the tool to check for possible concurrency errors, such as existence of deadlock, livelock, starvation, and fairness. Therefore, JCSP and programs written using different implementations of the CSP model can be formally verified for correct multi-parallel behavior before or after development, and the mapping is almost a one-to-one mapping in both directions, reducing the risk of introducing errors in the model checking/verification code.

5. Results

5.1. Line Count Comparisons

Comparing the line count for each solution can give an indication of the readability and writability of each solution. To be fair, the line counts for the Ada, Polyphonic C#, Haskell STM, and Erlang solutions are omitted because they did not include the message ordering constraints that we imposed on the problem. Table 1 gives an overview of the line count for each version:

5.1.1. Synchronization and Communication

The shared memory solutions all have roughly the same number of lines of synchronization and communication code. The line counts are higher than the distributed memory and process oriented counterparts due to the fact that wait/notify must be wrapped in a lock and unlock statement and extra logic is required in case of lost notifications. The line count here for all of the solutions would be much higher without the use of barriers, and would require duplicated synchronization logic each time a group of threads or processes needed to synchronize with each other.

5.1.2. Preventing Race Conditions

Each of the shared memory solutions includes locking and unlocking code to prevent race conditions whenever inserting and removing a Reindeer or Elf into a queue. Locks must also be in place anytime there is logic that depends on the size of the Elf and Reindeer queue, in this case when the last Reindeer or every third Elf must notify Santa. For the distributed memory and process oriented solutions there is no shared memory so there is no code needed to prevent a race condition.

5.1.3. Exception and Error Handling

In each solution the error handling code will simply print an error message to the console. For a real world problem, the error handling code would include logic that attempts to recover from errors. Gracefully recovering from errors in a concurrent system often requires additional communication and synchronization so that a particular thread or process can determine the state of the system as a whole and then bring itself to an appropriate state. Due to checked exceptions and the two exceptions that a $CyclicBarrier.await$ can throw the Java error handling line count is much higher than the other solutions. For the Groovy solution, closures have been used to wrap $CyclicBarrier.await$ and $Object.wait$, consolidating the error handling into the closure and reducing the error handling line count. For the C# version the unchecked exceptions in the .NET threading library were ignored. The C version ignores the errors that the pthread library calls can generate. The MPI version includes a macro for error handling which is used to check for errors every time a call to a method in the library takes place. The parts of JCSP that are used in this solution do not throw checked exceptions so there is no exception handling code in the JCSP version.

5.1.4. Custom Barrier Implementation

For C, C# custom barriers were implemented in order to reduce the amount of code required for thread synchronization and also prevent errors. If a barrier implementation is broken in can be fixed once without having to change any of the Santa, Elf, or Reindeer code. Custom barriers were implemented in the JCSP solution both to simplify process synchronization and to place the logic for the "three elves at a time" constraint into a special barrier type, eliminating the need for a separate Elf queue process like the one used in the MPI solution. The Java thread library and MPI version have built-in barrier implementations that are robust enough to solve the Santa Claus Problem and so no custom barriers were implemented in the Java, Groovy, and MPI solutions.

5.1.5. GUI

The additional line count for the C# version can be attributed to two things. The first is the additional lines added for a Windows Forms GUI. The second is that Visual Studio formats code with beginning braces on their own line, while in the other solutions they are not.

6. Conclusions

We have implemented solutions to the Santa Claus Problem using various concurrent programming models as a basis for comparing the three models, shared memory, distributed memory, and process oriented.

For the shared memory solutions two difficulties were identified. The first issue was the overly complicated use of mutexes, condition variables, and monitors to ensure mutual exclusion between various sets of instructions when using multiple threads. The second was the difficulty of using the wait/notify mechanism due to the asynchronous nature of the notify and the possibility of lost notifications by the receiver. Comparisons between the shared memory models themselves were discussed to show that the minor implementation differences do not alleviate the issues with the model.

We have shown how a distributed memory model such as MPI can simplify process ordering with the use of synchronous messages and how a a distributed memory model also gives built-in data integrity and allows for much simpler implementations of various program constraints. Furthermore, we have introduced JCSP and shown how the use of libraries and languages based on CSP further simplify the development of a concurrent application in a distributed memory model. The additional benefit of a one-to-one mapping with CSP, an algebra designed to describe process interaction, allows for the ability to formally verify various correctness measures of the application. A final benefit of JCSP over MPI is the use of channels which allows for increased trust between communicating processes.

The code for all of the solutions we implemented is available at:

`http://www.santaclausproblem.net`

7. Future Work

The Santa Claus Problem requires heavy synchronization, but future work should include the comparison of all three models for a wider range of problem types. A more rigorous comparison should be done between the three models, including the performance of the models in various settings. In addition to comparing the models themselves, it may be useful to compare the model checking tools available for each model, such as MPI-Spin and FDR. Ease of use and what can and cannot be proved are two interesting criteria for comparison here.

Acknowledgements

We acknowledge the comments of one of the referees who suggested that a solution that exploits Groovy Parallel [34] and which also employs the JCSP synchronisation primitives; Alting Barrier [35] and Bucket might be even shorter. In fact such a solution is about 75 lines shorter than the JCSP solution described previously.

References

[1] PlanetLab home page. `http://www.planet-lab.org`.
[2] BOINC home page. `http://boinc.berkeley.edu`.
[3] SETI Institute home page. `http://www.seti.org`.
[4] Google APIs. `http://code.google.com/more`.
[5] Yahoo! Developer Network. `http://developer.yahoo.com`.
[6] C. Ajluni. Multicore Trends Continue to Drive the Embedded Market in 2007. *Chip Design*, Jan/Feb 2008.

[7] J. A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[8] N. Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#. Technical report, Microsoft Research, Cambdridge UK, 2003.

[9] M. Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[10] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming, First Edition.* O'Reilly Media, 1996.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Professional, 2005.

[12] R. F. Stark and E. Borger. An ASM specification of C# threads and the .NET memory model. In *ASM 2004*, pages 38–60, 2004.

[13] Sun Microsystems home page. `http://www.planet-lab.org`.

[14] CyclicBarrier API. `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/CyclicBarrier.html`.

[15] D. Koenig, A. Glover, P. King, G. Laforge, and J.Skeet. *Groovy in Action.* Manning Publications Co., 2007.

[16] LinuxThreads library. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[17] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux. 2005.

[18] ANSI/ISO/IEC. *Ada 95 Language Reference Manual*, 1995.

[19] S. P. Jones. *Beautiful concurrency.* O'Reilly, 2007.

[20] C. Flanagana and S. N. Freund and S. Qadeerc and S. A. Seshia. Modular verification of multithreaded programs. In *Theoretical Computer Science*, 2005.

[21] C. Flanagana and S. Qadeerc. Thread-modular model checking. In *Model Checking Software*, 2003.

[22] J. Dongarra. MPI: A message passing interface standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[23] The Santa Claus Problem (Erlang). `http://www.crsr.net/Notes/SantaClausProblem.html`.

[24] Solving the Santa Claus Problem in Erlang. `http://www.cs.otago.ac.nz/staffpriv/ok/santa/index.htm`.

[25] ets. `http://www.erlang.org/doc/man/ets.html`.

[26] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[27] S. R. Siegel. Model Checking Nonblocking MPI Programs. *Verification, Model Checking, and Abstract Interpretation*, 4349:44–58, 2007.

[28] P. H. Welch. Process Oriented Design for Java: Concurrency for All. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, page 687, London, UK, 2002. Springer-Verlag.

[29] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[30] occam 2.1 reference manual. `http://www.wotug.org/occam/documentation/oc21refman.pdf`, 1995.

[31] P. H. Welch. Communicating Sequential Processes for Java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp`.

[32] P. H. Welch and F. R. M. Barnes. occam-π: blending the best of CSP and the π-calculus. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[33] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 manual*, 1998.

[34] J. M. Kerridge, K. Barclay, and J. Savage. Groovy Parallel! A Return to the Spirit of occam? In *Communicating Process Architectures*, pages 13–28, 2005.

[35] P. H. Welch, N. C. C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In *Communicating Process Architectures*, pages 349–370, 2007.