1

Introduction and overview

The arrival and popularity of multi-core processors have sparked a renewed interest in the development of parallel programs. Similarly, the availability of low-cost microprocessors and sensors has generated a great interest in embedded real-time programs. Ada is arguably the most appropriate language for development of parallel and real-time applications. Since it was first standardized in 1983, Ada's three major goals have remained:

- Program reliability and maintenance
- Programming as a human activity
- Efficiency

Meeting these goals has made Ada remarkably successful in the domain of mission-critical software. It is the language of choice for developing software for systems in which failure might result in the loss of life or property. The software in air traffic control systems, avionics, medical devices, railways, rockets, satellites, and secure data communications is frequently written in Ada. Ada has been supporting multiprocessor, multi-core, and multithreaded architectures as long as it has existed. We have nearly 30 years of experience in using Ada to deal with the problem of writing programs that run effectively on machines using more than one processor. While some have predicted it will be another decade before there is a programming model for multi-core systems, programmers have successfully used the Ada model for years. In February 2007, Karl Nyberg won the Sun Microsystems Open Performance Contest by building an elegant parallel Ada application (Nyberg, 2007). The parallel Ada code he wrote a decade earlier provided the foundation of this success.

It is estimated that over 99% of all the microprocessors manufactured these days end up as part of a device whose primary function is not computing (Turley, 1999). Today's airplanes, automobiles, cameras, cell phones,

GPS receivers, microwave ovens, mp3 players, tractors, and washing machines all depend on the software running on the microprocessors embedded within them. Most consumers are unaware that the software in their new washing machine is far more complex than the software that controlled the Apollo moon landings. Embedded systems interact with the world through sensors and actuators. Ada is one of the few programming languages to provide high-level operations to control and manipulate the registers and interrupts of these input and output devices.

Most programs written for embedded systems are real-time programs. A real-time program is one whose correctness depends on both the validity of its calculations and the time at which those calculations are completed (Burns and Wellings, 2009; Laplante, 2004). Users expect that their wireless phone will convert and transmit their voice quickly and regularly enough that nothing in their conversation is lost. Audio engineers quantify "quickly" into the maximum amount of time that the analog to digital conversion, compression, and transmission will take. They specify a set of deadlines. A deadline is a point in time by which an activity must be completed. Software engineers have the responsibility of ensuring that the computations done by the software are completed by these deadlines. This responsibility is complicated by the parallel nature of most real-time embedded software. One embedded processor may be responsible for a number of related control operations. Our wireless phone should not lose any of the video it captures simultaneously with our talking. The software engineer must ensure that all of the tasks competing for processor cycles finish their computations on time.

1.1 Parallel programming

A parallel program is one that carries out a number of operations simultaneously. There are two primary reasons for writing parallel programs. First, they usually execute faster than their sequential counterparts. The prospect of greater performance has made parallel programming popular in scientific and engineering domains such as DNA analysis, geophysical simulations, weather modeling, and design automation. There are theoretical limits on how much speedup can be obtained by the use of multiple processors. In 1967, Gene Amdahl published a classic paper (Amdahl, 1967) that showed that if P is the fraction of a sequential program that can be run in parallel

on N processors, the maximum speedup is given by the formula

$$\frac{1}{(1-P)+\frac{P}{N}}$$

Thus, for example, if 80% of our program can be run in parallel, the maximum speedup obtained with four processors is

$$\frac{1}{(1-0.8)+\frac{0.8}{4}} = 2.5$$

The second reason for writing parallel programs is that they are frequently better models of the processes they represent. More accurate models are more likely to behave in the manner we expect. As the real world is inherently parallel, embedded software that interacts with its environment is usually easier to design as a collection of parallel solutions than as a single sequential set of steps. Parallel solutions are also more adaptable to new situations than sequential solutions. We'll show you some examples later in this chapter where the parallel algorithm is far more flexible than the sequential version. Before beginning our discussion of parallel programming, let's take a brief look at the hardware on which such programs execute.

1.1.1 Flynn's taxonomy

Michael Flynn (1974) described a simple taxonomy of four computer architectures based on data streams and instruction streams. He classifies the classic Von Neumann model as a Single stream of Instructions executing on Single stream of Data (SISD). An SISD computer exhibits no parallelism. This is the model to which novice programmers are introduced as they learn to develop algorithms.

The simplest way to add parallelism to the SISD model is to add multiple data streams. The Single stream of Instructions executing on Multiple streams of Data (SIMD) architecture is often called array or vector processing. Originally developed to speed up vector arithmetic so common in scientific calculations, an SIMD computer applies a single instruction to multiple data elements. Let's look at an example. Suppose we have the following declarations of three arrays of 100 real values.

```
subtype Index_Range is Integer range 1..100;
type Vector is array (Index_Range) of Float;

A, B, C : Vector;    -- Three array variables
```

We would like to add the corresponding values in arrays B and C and store the result in array A. The SISD solution to this problem uses a loop to apply the add instruction to each of the 100 pairs of real numbers.

```
for Index in Index_Range loop
   A (Index) := B (Index) + C (Index);
end loop;
```

Each iteration of this loop adds two real numbers and stores the result in array A. With an SIMD architecture, there are multiple processing units for carrying out all the additions in parallel. This approach allows us to replace the loop with the simple arithmetic expression

```
A := B + C;
```

that adds all 100 pairs of real numbers simultaneously.

The Multiple stream of Instructions executing on a Single stream of Data (MISD) architecture is rare. It has found limited uses in pattern matching, cryptology, and fault-tolerant computing. Perhaps the most well-known MISD system is the space shuttle's digital fly-by-wire flight control system (Knoll, 1993). In a fly-by-wire system, the pilot's controls have no direct hydraulic or mechanical connections to the flight controls. The pilot's input is interpreted by software which makes the appropriate changes to the flight control surfaces and thrusters. The space shuttle uses five independent processors to analyze the same data coming from the sensors and pilot. These processors are connected to a voting system whose purpose is to detect and remove a failed processor before sending the output of the calculations to the flight controls.

The most common type of parallel architecture today is the MIMD (Multiple stream of Instructions executing on Multiple streams of Data) architecture. Nearly all modern supercomputers, networked parallel computers, clusters, grids, SMP (symmetric multiprocessor) computers and multi-core computers are MIMD architectures. These systems make use of a number of independent processors to execute different instructions on different sets of data. There is a great variety in the ways these processors access memory and communicate among themselves. MIMD architectures are divided into two primary groups: those that use shared memory and those that use private memory.

The simplest approach to shared memory is to connect each processor to a common bus which connects them to the memory. The shared bus and memory may be used for processors to communicate and to synchronize their activities. Multi-core processors reduce the physical space used by placing multiple processors (cores) on a single chip. Each core usually has its

own small memory cache and shares a larger on-chip cache with its counterparts. In more complicated schemes, multiple processors may be connected to shared memory in hierarchical or network configurations.

When each processor in an MIMD system has its own private memory, communication among them is done by passing messages on some form of communication network. There are a wide variety of interconnection networks. In static networks, processors are hardwired together. The connections in a static network can be made in a number of configurations including linear, ring, star, tree, hypercube, and so on. Dynamic networks use some form of programmable switches between the processors. Switching networks may range from simple crossbar connections between processors in a single box to the internet-connecting processors on different continents.

1.1.2 Concurrent programming

With the many different organizations of parallel hardware available, it might seem that a programmer would need a detailed understanding of the particular hardware they are using in order to write a parallel program. As usual in our discipline, we are saved by the notion of abstraction. The notion of the concurrent program as a means for writing parallel programs without regard for the underlying hardware was first introduced by Edsger Dijkstra (1968). Moti Ben-Ari (1982) elegantly summed up Dijkstra's idea in three sentences.

> Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems. Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming. Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details.

When a sequential program is executed, there is a single thread of control. The instructions are executed one at a time in the order specified by the program. A concurrent program is a collection of sequential processes.[1] Each of these processes has its own thread of control that executes its instructions one at a time, its own set of registers, and its own stack for local variables, parameters, etc. Given adequate hardware, each process may execute on its own processor using shared or private memory. A concurrent program may

[1] The term process has specific and sometimes different meanings in the context of particular operating systems. The term process in the context of concurrent programming is a general term for a sequence of instructions that executes concurrently with other sequences of instructions.

also execute on a single processor system by interleaving the instructions of each process. Between these extremes is the more common situation of having N processes executing on M processors where $N > M$. In this case, the execution of instructions from N processes is interleaved among the M processors.

If the processes in a concurrent program are independent, we can write each process in the same manner in which we write a sequential program. There are no special concerns or requirements that the programmer must address. However, it is rare that the processes in a concurrent program are truly independent. They almost always need to share some resources or communicate with each other to solve the problem for which the concurrent program was written. Sharing and communication require programming language constructs beyond those needed in sequential programs. In this chapter we introduce the problems that we must resolve in concurrent programs with interacting processes. Later, we present the Ada language features and techniques for solving them.

Synchronization is a problem faced in any activity involving concurrent processing. For example, suppose that Horace and Mildred are cooperating in the cooking for a dinner party. They expect that the work will go faster with two people (processors) carrying out the instructions in the recipes. For the preparation of scalloped potatoes, Horace has taken on the responsibility of peeling the potatoes while Mildred will cut them into thin slices. As it is easier to peel a whole potato rather than remove peels from individual slices of potato, the two have agreed to synchronize their operations. Mildred will not slice a potato until Horace has peeled it. They worked out the following algorithms:

Horace's scalloped potato instructions

while there are still potatoes remaining **do**
  Peel one potato
  Wait until Mildred is ready for the potato
  Give Mildred the peeled potato
end while

Mildred's scalloped potato instructions

loop
  Wait until Horace has a peeled potato
  Take the peeled potato from Horace
  Slice the potato
  Place the potato slices in the baking dish
  Dot the potato slices with butter
  Sprinkle with flour
  Exit loop when Horace has peeled the last potato
end loop

Communication among processes is another activity present in nearly all concurrent activities. **Communication** is the exchange of data or control signals between processes. In our cooking example, Horace communicates directly with Mildred by handing her a peeled potato (data). Further communication is required to let Mildred know that he has peeled the last potato (a control signal).

What happens when Mildred's potato slicing is interrupted by a phone call? When Horace finishes peeling a potato, he must wait for Mildred to complete her phone call and resume her slicing activities. Horace's frustration of having to hold his peeled potato while Mildred talks to her mother can be relieved with a more indirect communication scheme. Instead of handing a peeled potato directly to Mildred, he can place it in a bowl. Mildred will take potatoes out of the bowl rather than directly from Horace. He can now continue peeling potatoes while Mildred handles the phone call. Should Horace need to answer a knock at the front door, Mildred may be able to continue working on the potatoes that he piled up in the bowl while she was on the phone. The bowl does not eliminate all waiting. Mildred cannot slice if there is not at least one potato in the bowl for her to take. She must wait for Horace to place a peeled potato into the bowl. Horace cannot continue peeling when the bowl is completely full. He must wait for Mildred to remove a potato from the bowl to make room for his newly peeled potato.

Our concurrent scalloped potato algorithm is an example of the producer-consumer pattern. This pattern is useful when we need to coordinate the asynchronous production and consumption of information or objects. In Chapter 4 we'll show you how Ada's protected object can provide the functionality of the potato bowl and in Chapter 5 you will see how Ada's rendezvous allows processes to communicate directly with each other.

Mutual exclusion is another important consideration in concurrent programming. Different processes typically share resources. It is usually not acceptable for two processes to use the same resource simultaneously. **Mutual exclusion** is a mechanism that prevents two processes from simultaneously using the same resource. Let's return to our cooking example. Suppose that Mildred is currently preparing the cake they plan for dessert while Horace is preparing the marinade for the meat. Both require use of the 5 ml (1 teaspoon) measuring spoon. Mildred cannot be measuring salt with the spoon at the same time Horace is filling it with soy sauce. They must take their turns using the shared spoon. Here are the portions of their cooking algorithms they worked out to accomplish their sharing:

Horace's marinade instructions

Wait until the 5 ml measuring spoon is on the spoon rack

Remove the 5 ml measuring spoon from the rack
Measure 10 ml of soy sauce
Wash and dry the 5 ml measuring spoon
Return the 5 ml measuring spoon to the spoon rack

Mildred's cake instructions

Wait until the 5 ml measuring spoon is on the spoon rack
Remove the 5 ml measuring spoon from the rack
Measure 5 ml of salt
Wash and dry the 5 ml measuring spoon
Return the 5 ml measuring spoon to the spoon rack

This example illustrates a set of steps for using a shared resource. First, some **pre-protocol** is observed to ensure that a process has exclusive use of the resource. In our cooking example, Horace and Mildred use the fact that the rack on which they hang their measuring spoons cannot be accessed by two people at the same time in their small kitchen. Second, the resource is used for a finite amount of time by a single thread of control (a person in our cooking example). This stage of mutual exclusion is called the critical section. A **critical section** is a sequence of instructions that must *not* be accessed concurrently by more than one thread of execution. In our example, the critical section is the use of the 5 ml measuring spoon. After the critical section, the fourth and final step, the post-protocol, is carried out. The **post-protocol** signals an end of the exclusive use of the shared resource. In our example, the post-protocol consists of cleaning the measuring spoon and returning it to the spoon rack.

Mutual exclusion is a static safety property (Ben-Ari, 1982). The requirement that each critical section excludes other processes from using the resource (Mildred and Horace cannot both be filling the 5 ml spoon) does not change during the execution of the instructions. It is possible to have safety features that adversely affect the dynamic behavior of a system. For example, we could make a hand-held electric power saw completely safe by cutting off the power cord. However, such a drastic safety measure precludes us from using the saw for its primary purpose. The safety provided by mutual exclusion can also disrupt our system so that its goals remain unfulfilled. Returning to our cooking example, Mildred and Horace have devised the following algorithms for sharing two different measuring spoons.

Horace's deadly embrace

Wait until the 5 ml measuring spoon is on the spoon rack
Remove the 5 ml measuring spoon from the rack
Wait until the 15 ml measuring spoon is on the spoon rack
Remove the 15 ml measuring spoon from the rack
Measure 20 ml of soy sauce

Wash and dry the 15 ml measuring spoon
Return the 15 ml measuring spoon to the spoon rack
Wash and dry the 5 ml measuring spoon
Return the 5 ml measuring spoon to the spoon rack

Mildred's deadly embrace

Wait until the 15 ml measuring spoon is on the spoon rack
Remove the 15 ml measuring spoon from the rack
Wait until the 5 ml measuring spoon is on the spoon rack
Remove the 5 ml measuring spoon from the rack
Measure 20 ml of salt
Wash and dry the 5 ml measuring spoon
Return the 5 ml measuring spoon to the spoon rack
Wash and dry the 15 ml measuring spoon
Return the 15 ml measuring spoon to the spoon rack

These steps certainly prevent Horace and Mildred from using the same measuring spoon simultaneously. The cooking for most dinner parties goes without problem. However, one day, when the guests arrive they find no meal and Horace and Mildred waiting in the kitchen. This is an example of deadlock. **Deadlock** means that no process is making progress toward the completion of its goal. Can you see what happened in the kitchen that fateful day?

Use of scenarios is a common way to uncover the potential for deadlock in a concurrent program. A **scenario** is one possible sequence of events in the execution of a concurrent set of instructions. Here is one scenario that results in the deadlock observed in the kitchen.

1. Mildred finds that the 15 ml measuring spoon is available
2. Mildred removes the 15 ml measuring spoon from the rack
3. Horace finds that the 5 ml measuring spoon is available
4. Horace removes the 5 ml measuring spoon from the rack
5. Horace finds that the 15 ml measuring spoon is not available and waits
6. Mildred finds that the 5 ml measuring spoon is not available and waits

Our two cooks become deadlocked when each of them holds the measuring spoon that the other needs to continue. This problem may arise any time that a process requires multiple resources simultaneously. How is it possible that previous dinner parties have come off without problem? Here is a scenario in which everything goes smoothly with the sharing of the measuring spoons.

1. Mildred finds that the 15 ml measuring spoon is available
2. Mildred removes the 15 ml measuring spoon from the rack
3. Mildred finds that the 5 ml measuring spoon is available
4. Mildred removes the 5 ml measuring spoon from the rack

5. Horace finds that the 5 ml measuring spoon is not available and waits
6. Mildred measures 20 ml of salt
7. Mildred washes and dries the 5 ml measuring spoon
8. Mildred returns the 5 ml measuring spoon to the spoon rack
9. Horace finds that the 5 ml measuring spoon is available
10. Horace removes the 5 ml measuring spoon from the rack
11. Horace finds that the 15 ml measuring spoon is not available and waits
12. Mildred washes and dries the 15 ml measuring spoon
13. Mildred returns the 15 ml measuring spoon to the spoon rack
14. Horace finds that the 15 ml measuring spoon is available
15. Horace removes the 15 ml measuring spoon from the rack
16. and so on . . . *Each cook has used both spoons successfully*

The simplest solution to the form of deadlocking seen in the first scenario is to forbid the simultaneous use of multiple resources. In our kitchen we could restructure the two algorithms so that each cook takes one measuring spoon, uses it, and returns it before taking the second spoon. There is no reason that each cook needs both measuring spoons at a given time. However, there are times when processes do need multiple resources simultaneously. For example, a cook may need both a measuring spoon and a bowl to complete a particular mixing chore. A common solution for avoiding deadlock in this situation is to require that all processes obtain the resources in the same order. If our two cooks are required to obtain the 5 ml spoon before obtaining the 15 ml spoon, they will avoid deadlock. The first cook to take the 5 ml spoon will be able to obtain the 15 ml spoon as well. The second cook will wait for the 5 ml spoon to be returned to the rack and will not, in the meantime, try to obtain the 15 ml spoon.

Now that you understand scenarios, you may notice another potential problem in our mutual exclusion examples. Suppose one cook observes that the 5 ml measuring spoon is on the rack. But before they remove it, the other cook also observes that the spoon is on the rack and removes it. The first cook is now baffled by the disappearance of the spoon they recently observed. Our pre-protocol consisted of two separate steps: observing and removing. For this example and for all other mutual exclusion protocols to succeed, the pre-protocol must be completed as an atomic action. An **atomic action** is an action that cannot be interrupted. The observation of the spoon must be immediately followed by its removal. Fortunately for our cooks, the kitchen is so small that when one cook is checking the spoon rack, there is no room for the other cook to approach it. Our observe and remove is done as an atomic action.