Chapter 8
Parallel Computing

A brief Introduction

We are all very familiar with our own PCs, our desktops or laptops, especially the gaming ones with glowing keyboards. Perhaps 'familiar' is not quite the right word; surely, we all have 'personal relationships' with our machines, because we live with them every day. Our own machine is like a spouse, we are effectively married until hardware or software improvement and demands makes us divorce!

What triggers such an event? A significant *economic* cycle where sales of computing devices depend on increase of computing power (operations per second) and increased power enables more complex applications to run. Such applications and the computing power both sit together on a knife edge; the applications use the full power and demand even more. So, computers become more powerful, and the applications become more complex. The knife edge has not disappeared, it has only sharpened.

As developers (and not chip designers) there are a couple of ways we can write our applications to increase their performance. First on a multi-core machine we can write our programs, so they run on all cores simultaneously, this is *multi-processing*. We are thinking of a single box here where the cores *share memory*, and this requires a particular style of parallel programming. We shall not discuss *distributed* parallel processing where each core has its own memory, think about running an application on all PCs in our lab in parallel.

The second approach we can take is to parallelize our application on a *single-core* machine, such as a robot microcontroller, this is *multi-tasking*. This approach to parallel programming provides a strong way of structuring

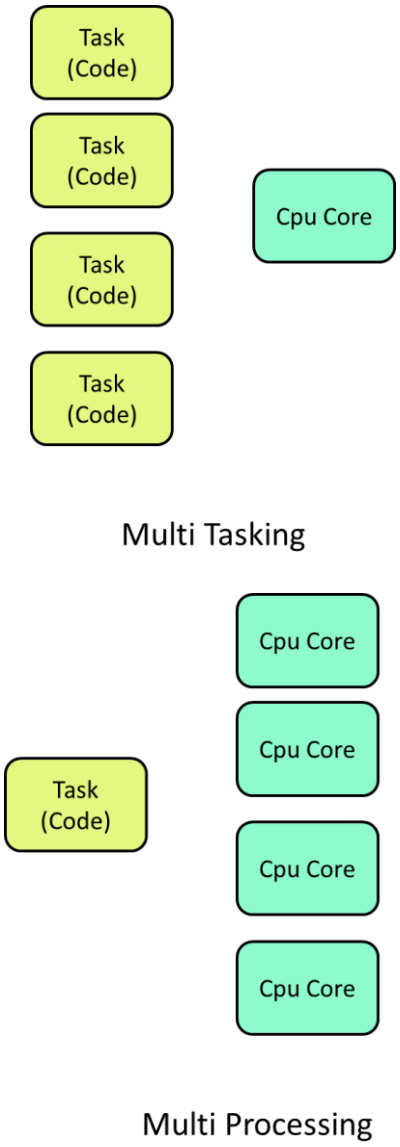


Figure 1 Distinction between multitasking and multiprocessing.

our code most efficiently, cleanly and encourages a more OO structure with resulting code reusability. The two approaches we shall unravel in this chapter are shown in Figure 1.

Our study of multitasking will take place using our single core Arduino where we shall look at a couple of features of a *Real Time Operating System* (RTOS) which allows us to run several tasks in parallel and in real time. Application of a RTOS abound in systems driven by microcontrollers, especially those with sensor inputs and actuator outputs. The RTOS allows us to organize our code into coherent units (called *threads*). We can then partition our code into threads, one will handle inputs, another will handle outputs. So this sounds like object-oriented programming in the C-language which is flat not OO. Clearly threads need to communicate with each other, and some may need to respond more quickly than others; the thread reading a sensor value must not miss a reading, so this thread should run at a higher *priority*. All this (and more) is achieved by using a RTOS. Perhaps the closest you have seen to an RTOS is a Finite State Machine, though there are some conceptual differences.

Multiprocessing is a different beast. This is all about how to run *a single program or a single function* on multiple hardware cores. It is not about running a separate program on each core. This involves working with existing code and looking for sections where computations can be done in parallel. Consider the following operation on some arrays.

```
for (int i=0; i<10; i++){
   c[i] = a[i] + b[i];
}
```

It is straightforward to understand how this code can be spread over multiple cores. Say we have 10 cores, then core 1 could compute `c[1] = a[1] + b[1];` and core 2 could compute `c[2] = a[2] + b[2];` and so on. All computations would occur at the same time *in parallel*. So we have effectively unrolled the loop and spread it over the cores. Nice eh?

1. Real Time Operating Systems

1.1 Background

To understand how a RTOS works on a microcontroller (our Arduino) we need to refresh our understanding of typical microcontroller hardware. Figure 2 presents a very simplified description; you can look up the ATmega328 datasheet from Atmel if you like. You recognize the CPU and RAM, also there is I/O electronics, and a timer that can be programmed to emit output pulses. The crucial part for the RTOS is the part of the CPU that handles interrupts. These come from input signals (an input pin may change state from HIGH to LOW) and this change is used to rapidly change which part of the user's program is executed. Here's some Arduino code. In setup() pin 2 is attached to the Interrupts system and is associated with the *interrupt service routine* (ISR) here named **ISR1** That routine is coded at the end. Within loop() there is a call to another function which makes the LEDs blink.

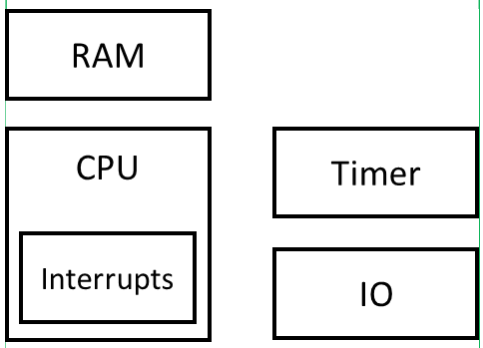


Figure 2 Greatly simplified microcontroller system.

```
void setup(){

attachInterrupt(digitalPinToInterrupt(2),ISR1,
CHANGE);
}

void loop(){
  blink_LEDs();
}

void ISR1() {
   bRunning = false;
}
```

So, let's say you have a push button connected to pin 2. If you leave it alone, the code in loop() will merrily churn around. Then you press the button and change the voltage on pin 2. The CPU Interrupts unit recognizes this and *immediately jumps out of the executing loop() code and vectors to the ISR and executes code there.* Then it returns to the loop() code at the point it left off. This happens extremely quickly, on an

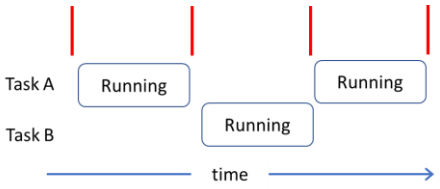


Figure 3 Interrupts selecting taskA then taskB to run

ATMega328 it takes 4 CPU clock cycles, running at 16MHz this equates to a quarter of a microsecond! It's also important to note that this code interruption occurs at the level of *machine instructions*. That, as we shall see has some interesting consequences.

Now, an interesting feature of microcontrollers is that their Timers can be used to change the state of an I/O pin. So, if that pin has an interrupt attached, then the timer can raise a periodic interrupt. So, we can write code which will regularly interrupt itself and go off and do something else. This is just what is needed for a RTOS to be able to switch processing from one task to another. Such code forms part of the RTOS *kernel* (core code) and is used to run the RTOS *Scheduler*. Figure 3 shows the idea where the red bars show the interrupts generated by the scheduler; when an interrupt occurs, the CPU is vectored to code from either task A or task B in turn. Here I have shown regular *slices* of time.

1.2 A simple two-task program

Now let's start to add some more detail. Consider the following code comprising 2 tasks. I've omitted task setup code. There are two LEDs attached to the microcontroller; it's easy to see what the code will do. Since Task1 runs for a slice, then Task 2 runs for a slice, then if the time slices are short enough we shall perceive that both tasks run in parallel; LED1 blinks with a period of 1 second and LED2 with a period of 2 seconds.

```
void Task1() {
   while(1) {
      turnLED1(on);
      delay(500);
      turnLED1(off);
      delay(500);
   }
}
```

```
void Task2() {
   while(1) {
      turnLED2(on);
      delay(1000);
      turnLED2(off);
      delay(1000);
   }
}
```

The timing in Figure 3 agrees with this example

1.3 The Scheduler

It's clear that a task can exist in one of two states *running* or *not-running*. When a task is running, we say that the scheduler has 'swapped it in' and when it is not running, the scheduler has 'swapped it out'. There is actually a finer distinction; when a task is not running, it exists in one of two substates, *ready* or *blocked*. A *ready* task is ready to be swapped in at the next time slice, on the other hand a *blocked* task is not ready and will not be swapped in. It may be waiting for some input. In this case, the task just swapped out will be immediately swapped in again (providing it is ready!). The task states are shown in Figure 4.

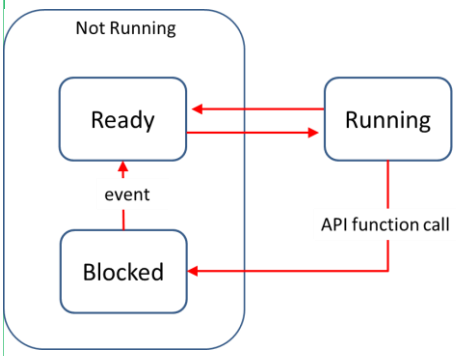
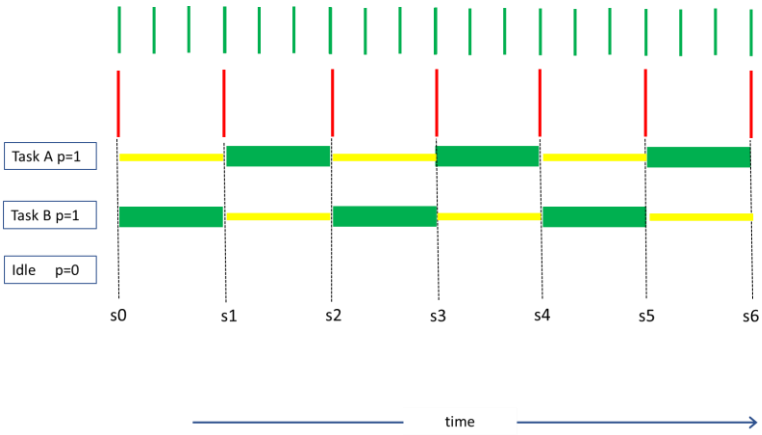


Figure 4 States of a Task

Tasks are put into a blocked state using some RTOS API call. for example, a request for a time delay, or waiting for some data to arrive from another task. They can transit to a ready state when some event occurs, like the time delay has expired or data arrives, or there is an external interrupt from a button push.

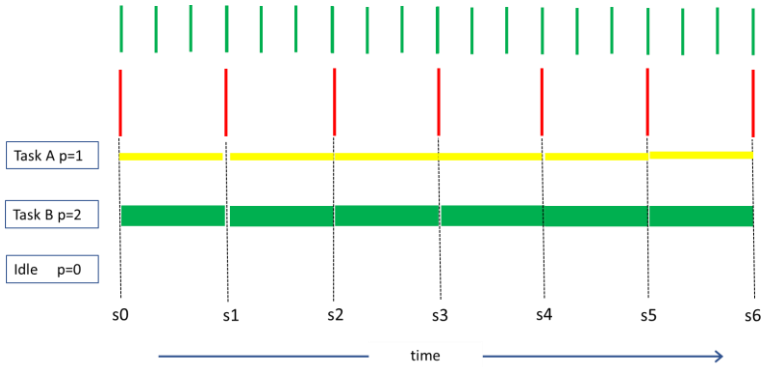
We can now introduce a more complete timing diagram for the code example presented above.



Time is shown running left to right. The green bars show the CPU clock or 'sysTick' and the red bars show the time slices. Horizontal green bars show when the task is running and yellow bars when it is ready. There are no blocked times for either thread. On the left you can see that both tasks have been

set to have the same priority. There is also an *Idle Thread* which we'll come onto shortly. Since both threads have the same priority, it is guaranteed that the scheduler will select each thread to run in turn (unless one blocks); this is called the *Round Robin* scheduling algorithm.

Now let's see what happens when the priority of task B is raised.



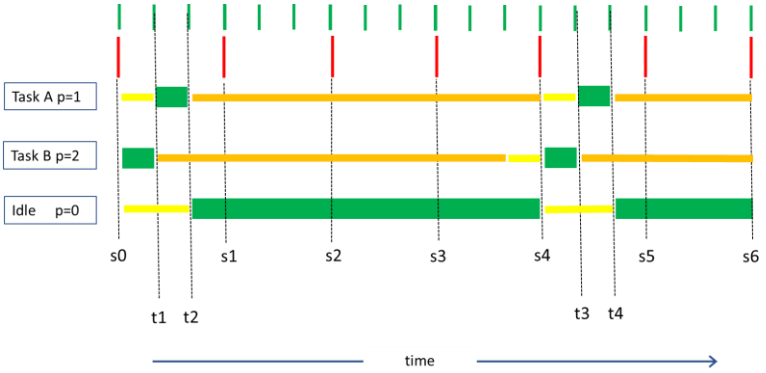
Here both tasks are ready, but the scheduler is guaranteed to swap in the task with higher priority, if both tasks are ready.

While this all sounds reasonable, both situations shown in the timing diagrams above are quite horrible, and should never happen. You can see why; at all times the CPU is running a task, it is totally consumed and has no time for anything else. Also both tasks need to have the same priority to be able to be swapped in. The *cause* of the problem was the calls to the **`delay()`** function in the code, which simply burned up CPU cycles in some dastardly horrible loop, very bad.

This is where the *blocked* state comes into play. FreeRTOS has an API call **`vTaskDelay()`** which puts the task into the blocked state for a certain number of ticks, so effectively providing the delay. Here's how you would use it

```
while(1) {
   turnLED1(on);
   vTaskDelay(1000/portTICK_PERIOD_MS);
   turnLED1(off);
   vTaskDelay(1000/portTICK_PERIOD_MS);
}
```

and here's what the timing diagram will look like. Orange represents blocked.



So, what is happening here ? Task B has the higher priority and is ready, so the scheduler swaps this in. It runs and then calls the **`vTaskDelay()`** API function which puts taskA into the blocked state. The scheduler sees this and also that the lower priority task A is ready so it swaps it in. Task A then runs until it hits its **`vTaskDelay()`** function at which point it too blocks and is swapped out.

At this point neither tasks A or B are ready, so the scheduler chooses the Idle Task which you can see runs merrily along. The idle task of lowest priority is created when the scheduler starts up and it is put into the ready state, so it has something to run.

After a while, Task B's delay ends, and it becomes ready so is swapped in at s4. It runs a bit more and at t3 calls its again and re-enters the blocked state. Fortunately, Task A is ready so it can run again.

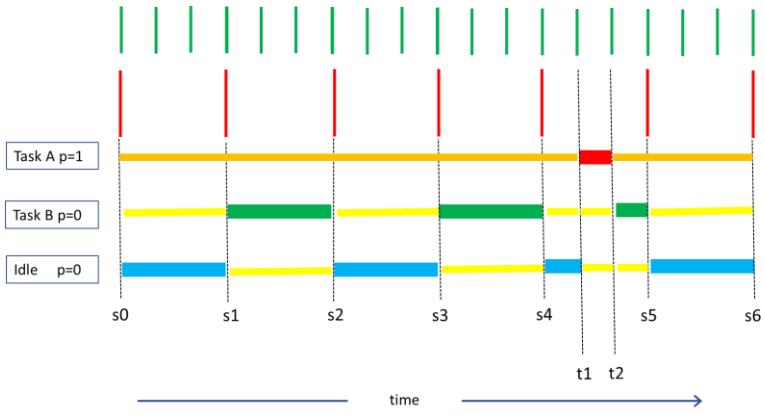
One more thing; the scheduler is not restricted to do swaps to the time slices s1, s2, … but can also act at sysTick intervals t1, t2, …

1.4 More Scheduling Algorithms

This section is moving into advanced territory but is important since we discuss how the majority of microcontroller RTOSs actually are configured. This uses the

Fixed Priority Pre-emptive Scheduling with Time Slicing. In other words, forget Round Robin. What does this mean? Well, 'Fixed Priority' means that the scheduler cannot change the priority of tasks (although tasks may change their priority and that of other tasks). 'Pre-emptive' means that if a low priority task is running and a higher priority task becomes ready, then the lower priority task will be swapped out (even though it does not want to). We know what time slicing is, if two tasks have equal priority then they are swapped at regular time-slice intervals.

You can imagine, the combination of time-slicing and pre-emption is quite powerful. Let's look at a hypothetical set of three tasks to see this in action.



TaskB and the Idle task both run at low priority and task A is event-driven and spends most of its time in the blocked state until its event arrives at t1. You can see that the scheduler time-slices task B and the idle task (same priorities) for quite a few slices. But in the middle of slice s4-s5 where idle starts running, task A becomes unblocked, hence ready, and pre-empts the idle task. Task A runs from t1 to t2 to complete its business, then the scheduler chooses to swap the idle task in according to the Round Robin policy.

1.5 Possible Issues with swapping.

There are a couple of potential issues in a multitasking system. The first concerns *shared resources* such as writing

to the serial port. Let's say task A wants to print out the string "Hello Cat" and task B wants to print out the string "rapture". Task A starts executing and manages to print "Hello C" then it is swapped out and task B prints "rapture". Then task A is swapped in and completes its output "at". So, what you actually get is "Hello Craptureat". Clearly, we have to protect the printing from a context switch.

Another problem is when 'read, modify, write' operations are encountered. For example, the following C-code reads the value of the variable **max** stored in memory, then ORs the value with 0x01, then writes the value back to the memory. The assembler code is also shown

```
max = max | fred;

LOAD R1, [#max]
LOAD R2, [#fred]
OR R1, R2
STORE R1, [#max]
```

The problem comes from the fact that a single line of C-code produces 4 lines of assembler (therefore machine code) and it is this code that is interrupted by the scheduler. Consider the case where task A and task B attempt to modify the same variable **portA**

- Task A loads the value of **max** into R1
- It is then pre-empted by task B. The scheduler saves all of taskA register values including R1 which contains the value of **max**
- Task B runs all 4 lines of machine code (read, modify, write) and updates **portA** then blocks.
- Task A is swapped in and its registers restored including R1 which contains the old value of **max**
- Task A runs to completion and update the value of **max** then writes it back.

The trouble is task A has used an out of date value for **max** and it overwrites the value calculated by task B.

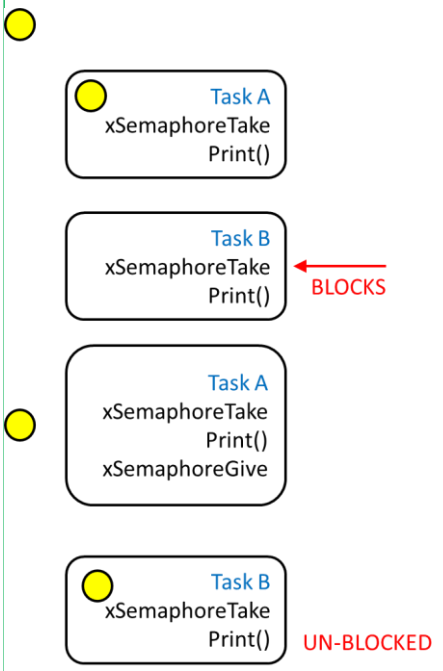


Figure 5 Yellow Mutex gives Task A exclusive access to the printer since task B gets itself blocked.

To prevent these issues, we must protect critical sections of code from being accessed by more than one task. The crudest is to use the critical section construct. In the example below a print statement is wrapped in a couple of macros. These compile into code which disable interrupts and therefore suspends the scheduler.

```
taskENTER_CRITICAL();
   Serial.println("Hello Cat");
taskEXIT_CRITICAL();
```

Turning off interrupts is never a good thing it renders the microcontroller blind to its inputs, and so it may miss a critical event, such as a warning of an impending collision.

A better way is to use a semaphore called a Mutex (from mutual exclusion). Think of a mutex as a single token which only one task can possess at any time, shown in Figure 5 as a yellow ball. Initially no task has the mutex, then task A tries to take it and it succeeds. It then calls its print() function. Shortly after, as task A is in the middle of its printing, task B wants to print, so it tries to take the mutex. But it can't (since task A has it) so task B enters the blocked state. When task A has completed printing, it returns the mutex, then it is given to task B which unblocks and does its printing.

1.6 A few loose threads

Reading around the subject, you might come across different terminology, some texts refer to 'threads' where we have referred to 'tasks'. Another name for 'swapping in and out' is 'context switching'. We have only hinted at what actually happens during a context switch. Remember this occurs at machine instruction level, so the exact state of the machine needs to be saved (the contents of its registers, any local variables and the instruction pointer) before it's swapped out. Only then can the task be later restored. This saving is done using a stack located in each thread.

2. Multiprocessing with OpenMP

2.1 A Brief Introduction

While there are many supercomputer architectures, two significant 'poles' of design stand out. First is the 'shared memory' architecture. Here the fundamental design principle is to share memory between each processor. While each processor has its own local cache (system) there is shared memory available to all processors Fig.6 (a) and that's where they collaborate. For example, if several processors update individual elements of a vector in parallel, then that vector will be in shared memory. OpenMP supports a shared memory architecture and provides the ability to set up teams of processing threads (operating on different processors) and share the work between them. OpenMP is not a language, as we shall see it comprises a number of compiler 'directives' which the programmer uses to identify regions of code they wish to parallelize.

The other end of the pole is 'distributed memory' such as in a networked or 'cluster' processing configuration, Fig.6 (b). Here, a different programming model is needed, which is referred to as 'message passing'. Since memory is not shared, each nodes needs to message other nodes to coordinate processing of variables.. The industry standard is 'MPI' which is often used by the hi-tech community where clusters of machines (serious boxes the size of a sub-post office) are common.

MPI programming can be tricky and normally requires re-programming of existing code to parallelize it, and certainly does not support an incremental port from sequential to parallel code. It is here that OpenMP shines, since (as mentioned) it consists of a set of 'directives' which the developer can add into existing code, to tell the compiler what sections should be parallelized.. Visual Studio has OpenMP built in for C, C++ and Fortran, not for C#. So, you can write parallel programs, today!

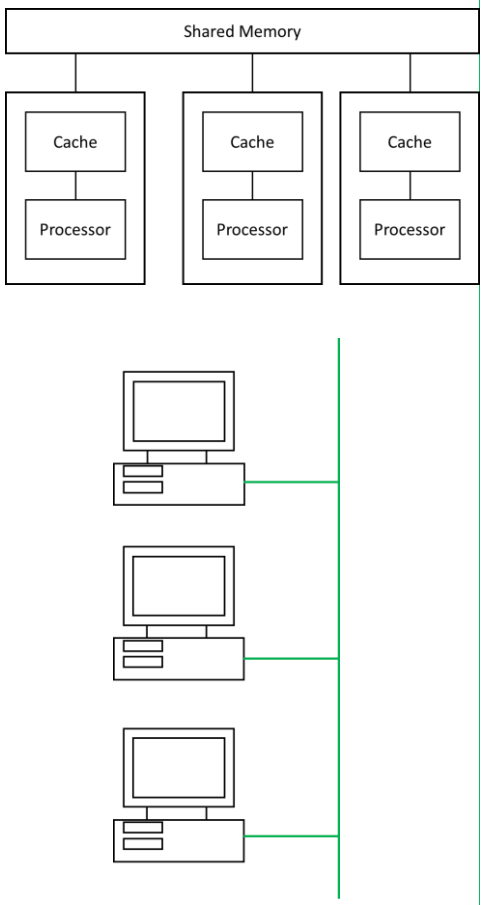


Figure 6 Shared memory (top) and distributed memory (bottom) multiprocessing architectures

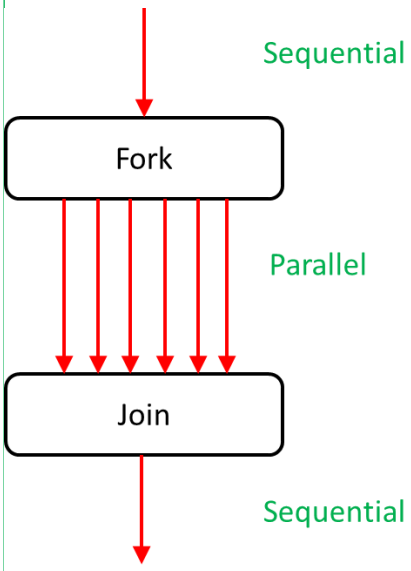


Figure 7 Fork-Join software architecture

2.2 The OpenMP Model

The OpenMP API consists of a number of 'directives' that tell the compiler when and how to parallelize code; we shall see some of these below. These directives take the form of #pragma statements, inserted into existing sequential code to parallelize it. In addition, OpenMP provides the programmer with (i) the means to create a 'team' of threads, (ii) structures to enable work sharing between the team members, (iii) specification whether variables are *shared* across all threads, or *private* to each thread, and of course (iv) a load of ways to synchronize all of this. OpenMP uses a 'fork-join' approach to execution which you may be familiar with if you were brought up on UNIX. This is shown in Fig.7 where the program execution flow starts from top and runs to bottom. Starting with one processing thread, there is a *fork* which creates a team of threads and does some useful work in parallel. When this is done, all threads agree to rendezvous at a point in time and to *join* after which one thread takes over the processing which becomes sequential again. For example, the initial sequential part could be to gather user input, the parallel part could be a multi-body simulation, and the final serial part could be the output of the body final positions.

Let's have a look at a couple of important OpenMP *directives*, and then give some examples along the way.

2.2.1 The Parallel Construct

To move from a single sequential thread, and create a *team* of threads *(fork*) running in parallel we wrap the code we wish to parallelize, inside this construct.

```
#pragma omp parallel
{

   our existing code which we want to run in parallel

}
```

At the end of this parallel region there is an implied ***barrier*** which forces all threads in the team to wait until they have completed their work. Then one thread will take on the remaining sequential programming. While this construct guarantees we have a team of threads, it does not specify how their work is to be shared. There are four work-sharing constructs, we shall encounter only a couple.

2.2.2 The Loop Construct

This is perhaps the most useful and important way to parallelize programs. Just think of how many loops you code on a daily basis. Here's how you would parallelize your loop. Let's say you have this in your existing code

```
for(i=0; i < 4; i++) {
   a[i] = b[i]*c[i];
}
```

In your sequential code a single core would calculate the value of **a[i]** in turn. So, it would calculate **a[0]**, then **a[1]**, then **a[2]**, then **a[3]**. That works fine. But we can do this 4 times as fast if we use 4 cores, where one core calculates each **a[i]**. Here's how we use the **#pragma omp for** construct.

```
#pragma omp parallel
{
   #pragma omp for
   for(i=0; i < 4; i++) {
      a[i] = b[i]*c[i];
   }
}
```

Here we first declare a parallel region **#pragma omp parallel**. Then we indicate that our for loop is to be parallelized using the **#pragma omp for** directive.

So ,what does this do? Say we have 4 cores on our machine. The following happens in parallel

core 0 calculates **a[0]**
core 1 calculates **a[1]**
core 2 calculates **a[2]**
core 3 calculates **a[3]**

So, we have obtained a x4 speedup in our computations, and we have done this by 'wrapping' our existing code in a couple of OpenMP compiler directives.

2.2.3 The Sections Construct

This is perhaps a little easier to understand than parallelizing loops (but perhaps not as useful). The idea is that we may have sections of our code which are independent. Yep, think *functions*. Let's say we have two functions in our code which we call sequentially like this

```
main() {
    func1();
    func2();
}
```

If we have two threads (or more) then we can execute both functions in parallel effectively assigning the processing of each function to an independent thread. Here we make use of the *sections* construct, placing this in a parallel region

```
#pragma omp parallel
{

   #pragma omp sections
   {
      #pragma omp section
      func1();
      #pragma omp section
      func2();
   }
}
```

So, one thread will execute **funct1()** *at the same time* as the second thread is executing **funct2().** One potential problem called 'load imbalance' occurs which reduces the parallelization speed-up. If one function has more work to do than the other, then the second thread will be hanging around waiting for the first, effectively doing nothing. Also if you had 5 functions and only 4 threads then one thread would do twice the amount of work and the other 3 would be twiddling their thumbs.

2.2.4 The Barrier Construct

A barrier is a point in the execution of code where all threads wait for each other; no thread is allowed to proceed until all threads have reached the barrier. While there is an *explicit* baddier construct **#pragma omp barrier** which can be inserted into code, other constructs provide an *implied* barrier which provides us with some very useful synchronization tools. Consider the code below which effectively contains two sections! In the first, the values of vector **a[i]** are initialized in parallel, *then when all have been initialized* they are used in the second section in some computation.

```
#pragma omp parallel
{
    #pragma omp for
    for(i=0;i<N;i++)
        a[i] = i;

    #pragma omp for
    for(i=0;i<N;i++)
        b[i] = 2*a[i];

}
```

Here there is an *implied* barrier at the end of the first **omp for** so all the values of **a[i]** are initialized before they are used in the second **omp for**.

One case where a barrier can be explicitly used is in our classic read-modify-write situation, which we could protect using a critical section or semaphore. If we inserted a barrier between writes to and read from a shared variable, that would avoid a data-race condition.

2.3 When OpenMP goes wrong or can't ever work.

2.3.1 Code which cannot be parallelized.

Let's compare the two loops shown below. The first can be parallelized but the second cannot.

```
for(i=0;i<N;i++)
      a[i] = a[i] + b[i];
```

```
for(i=0;i<N;i++)
      a[i] = a[i+1] + b[i];
```

In the first loop, the iterations are independent, and so can be shared across a whole team of threads, each thread dealing with the i'th update. The second loop has a dependency; the value of **a[i]** depends on the next value in the array **a[i]**. Say one thread is updating **a[i]** and expects to find the value of **a[i+1]** *at that time.* It might find it, but it might not, because another thread may have already updated **a[i+1];** We just can't know. If we ran this code it might work correctly but it might not; the behaviour is non-deterministic. This is another example of a *data race* condition, here introduced by an incorrect parallelization of a loop.

2.3.3 Thread-Safe and non-Thread-Safe Functions

We have stressed that one strength of OpenMP is that it allows us to use existing code. More than likely, our code uses a library containing functions which we have not written, and so have no idea what is in the library. In particular we may not know if the library makes use of global variables. This can lead to a problem. Here's a toy example. The main function calls a single library function which updates a global variable.

```
int nastyGlobal;

void lib_func() {
     nastyGlobal++;
     // some meaningful stuff
}

main() {
     #pragma omp parallel
     {
          lib_func();
     }
}
```

Since we are calling the library function from a parallel region multiple threads may try to access **nastyGlobal** at the same time; we have a read-modify-write operation. Here the problem is easily solved putting the update into a critical region,

```
void lib_func() {
     #pragma omp critical
     nastyGlobal++;
     // some meaningful stuff
}
```