An Introduction to Parallel Programming with OpenMP

by

Alina Kiessling

A Pedagogical Seminar

April 2009

Contents

Chapter 1

An Introduction to Parallel Programming with OpenMP

1.1 What is Parallel Computing?

Most people here will be familiar with **serial computing**, even if they don't realise that is what it's called! Most programs that people write and run day to day are serial programs. A serial program runs on a single computer, typically on a single processor[1]. The instructions in the program are executed one after the other, in *series*, and only one instruction is executed at a time[2].

Parallel computing is a form of computation that allows many instructions in a program to run simultaneously, in *parallel*. In order to achieve this, a program must be split up into independent parts so that each processor can execute its part of the program simultaneously with the other processors. Parallel computing can be achieved on a single computer with multiple processors, a number of individual computers connected by a network or a combination of the two.

Parallel computing has been around for many years but it is only recently that interest has grown outside of the high-performance computing community. This is

[1]A *Central Processing Unit* or *processor* is the 'brains' of a computer. Processors are responsible for executing the commands and processing data. Single computers can have more than one processor.

[2]In reality, it is not always as simple as this with the introduction of more sophisticated compilers that have the capability to allow the program to run more efficiently, but for the purposes of this short introduction, it is enough to think of serial programs running in the way described above.

due to the introduction of multi-core[3] and multi-processor computers at a reasonable price for the average consumer.

1.2 Why would you make your codes parallel?

The main reason to make your code parallel, or to 'parallelise' it, is to reduce the amount of time it takes to run.
Consider the time it takes for a program to run (T) to be the number of instructions to be executed (I) multiplied by the average time it takes to complete the computation on each instruction (t_{av})

$$T = I \times t_{av}.$$

In this case, it will take a serial program approximately time T to run. If you wanted to decrease the run time for this program without changing the code, you would need to increase the speed of the processor doing the calculations. However, it is not viable to continue increasing the processor speed indefinitely because the power required to run the processor is also increasing. With the increase in power used, there is an equivalent increase in the amount of heat generated by the processor which is much harder for the heat sink to remove at a reasonable speed[4]. As a result of this, we have reached an era where the speeds of processors is not increasing significantly but the number of processors and cores included in a computer is increasing instead.
For a parallel program, it will be possible to execute many of these instructions simultaneously. So, in an ideal world, the time to complete the program (T_p) will be the total time to execute all of the instructions in serial (T) divided by the number of processors you are using (N_p)

$$T_p = \frac{T}{N_p}.$$

In reality, programs are rarely able to be run entirely in parallel with some sections still needing to be run in series. Consequently, the real time (T_r) to run a program

[3] A *multi-core* processor is one that contains two or more independent cores in a single package built with a single integrated circuit.

[4] Heat must be dissipated in order for the processors to continue working within safe operating temperatures. If the processor overheats, it may have a shortened life-span and also may freeze the computer or cause crashes.

	Speed-up		
N	P = 0.5	P = 0.9	P = 0.99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90

Table 1.1: Scalability of parallelisation

in parallel will be somewhere in between T_p and T, ie $T_p < T_r < T$.
In the 1960's, Gene Amdahl determined the potential speed up of a parallel program, now known as *Amdahl's Law*. This law states that the maximum speed-up of the program is limited by the fraction of the code that can be parallelised

$$\begin{aligned} S + P &= 1 \\ \Rightarrow SU &= \frac{1}{S}. \end{aligned}$$

The serial fraction of the program (S) plus the parallel fraction of the program (P) are always equal to one. The speed-up (SU) is a factor of the original sequential runtime (T).
So, if only 50% of the program can be parallelised, the remaining 50% is sequential and the speedup is

$$\begin{aligned} SU &= \frac{1}{0.5} \\ &= 2, \end{aligned}$$

ie, the code will run twice as fast.
The number of processors performing the parallel fraction of the work can be introduced into the equation and then the speed-up becomes

$$SU = \frac{1}{\frac{P}{N} + S},$$

where S and P are the serial and parallel fractions respectively and N is the number of processors.
The effect of Amdahl's law is shown in Table 1.1 and Figure 1.1. **Just using more**

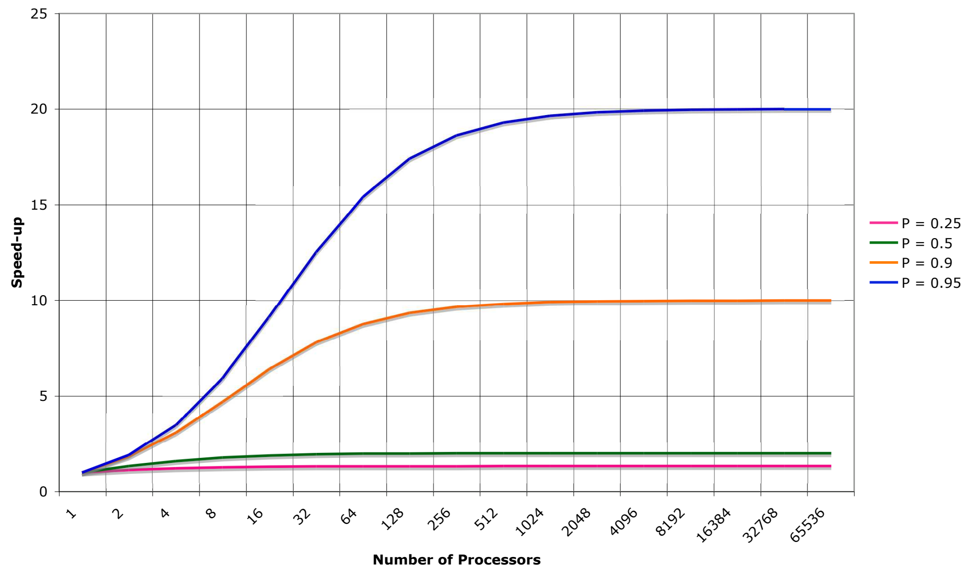

Figure 1.1: This figure shows the scalability of parallelised codes. It clearly shows that simply adding more processors will not always make a significant improvement to the run time of a code.

processors to run your code does not always result in a significant speed-up.

There is also a Modified Amdahl's law that is the same as Amdahl's law but it points out that you can make problems larger with parallelisation. For example, consider a simulation that you can run in series with 256^3 particles but running a larger number of particles in series is inconveniently time consuming. In this case, you may be able to parallelise the code to let you run a much larger realisation. The parallel fraction of this program, P, may not be the same for a 256^3 run as it is for a 1024^3 run. So P is a function of problem size, as the modified Amdahl's law points out.

There are also some 'overheads' that cause parallel codes to run slower than the speed-up that you may be expecting from the equations above. These overheads generally result from the amount of time the processors spend communicating with each other. There is a point where parallelising a program any further will cause the run time to increase, rather than decrease. This is because as you add more processors, each processor spends more time communicating with the master processor, for example, than doing the actual calculations. As a result of this, it makes more sense to parallelise codes that are working on large data sets rather than small ones because the processors will have more work to do in each parallel section. Calculating the overheads is tedious but can be achieved by timing each of the components of the program in serial and in parallel. This should not always be necessary, as long as the programmer factors in the overheads when determining the estimated speed-up of a program.

Before you go to the effort of making your codes parallel, it is beneficial to do a quick calculation with the speed-up equations discussed in this section to establish whether making your codes parallel will actually achieve a significant improvement on the speeds that you were achieving when running the code in serial. In order to determine the fraction of a code that is parallelisable, you need to profile it to identify the bottlenecks. Once the bottlenecks have been identified, the programmer must decide which of them are parallelisable and from this, determine the fraction, P, that can be made parallel. In order to determine the parallel fraction for larger and larger problems, you should profile your code at several smaller increments and interpolate. For example, if you want to determine the parallel fraction to run a simulation with 2048^3 particles but you can only run up to 256^3 particles in serial, you should profile 64^3, 128^3 and 256^3 particles and then interpolate the fraction for 2048^3 particles.

There are a number of different profiling tools available. The Linux systems at the ROE have the profiler **gprof** available. A detailed explanation of how to use profilers is given in the gprof documentation and the website is included at the end of this handout.

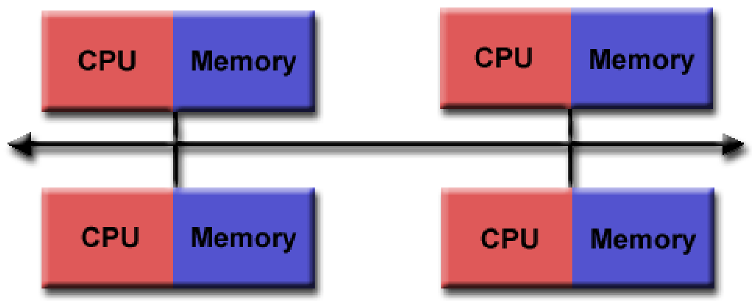

Figure 1.2: An example of a distributed memory configuration. Each processor has its own local memory and is connected to the other processors by some kind of network.

1.3 What is the difference between Shared and Distributed memory?

Distributed Memory

Distributed memory systems can vary widely but they all have a communications network connecting the inter-processor memory. These network connections can be as simple as an Ethernet connection or something more sophisticated. Figure 1.2 shows an example of a distributed memory configuration with each processor having its own local memory and being connected to each other processor through some kind of network. In these configurations, memory addresses on one processor do not map to memory associated with another processor. So, since each processor has its own local memory, it operates independently of all the other processors. Consequently, when a processor is required to do a task, the programmer needs to pass all the required data to that processor explicitly. This requires specific coding in the program, which can sometimes be complicated. An example of a distributed memory system is the desktops at the ROE. Each of our desktops is connected through the network and it is possible to run programs in parallel using several of these desktops together.
The advantage of using a distributed memory parallelisation is that the memory is scalable. If you increase the number of processors, the size of the memory available also increases. Each processor can also access its own memory very quickly which reduces the overhead time of processors communicating with each other. The disadvantage is that it can be quite difficult to write a distributed memory parallel program and a program can not easily be made parallel after it has been written in

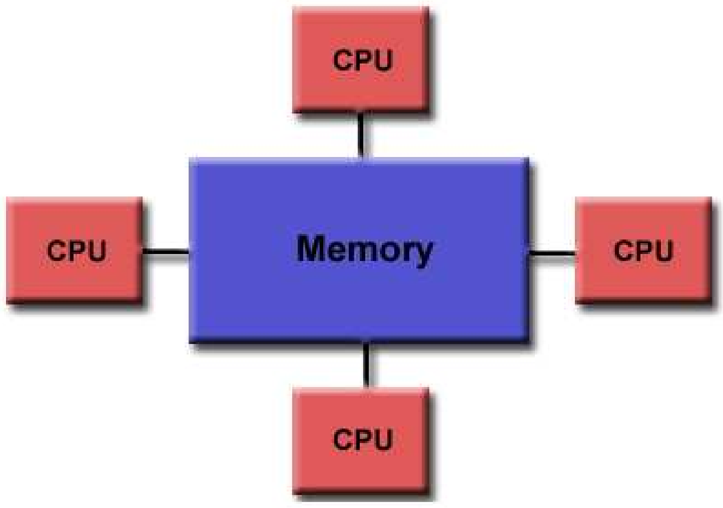

Figure 1.3: An example of a shared memory configuration where each processor has direct access to the global memory.

serial without a major re-write of the code.

Shared Memory

Shared memory systems can also vary widely but they all have the ability for each processor to access all memory as a global address space. Figure 1.3 shows an example configuration for a shared memory system where each of the processors is directly connected to the global memory. This configuration allows multiple processors to operate independently while still having access to all the same memory resources. An example of a shared memory system is a single desktop. So my machine here at the ROE, *quoich*, has two cores in its processor and each core can access all the memory on my machine.
The advantages of using a shared memory parallelisation is that it is relatively simple to make existing serial codes parallel. There are a few disadvantages which include the possibility that multiple cores and processors accessing the shared memory simultaneously could cause a bottleneck which will slow down a program. Also, adding more processors does not increase the amount of memory available which could be a problem. Finally, the programmer is responsible for making sure that writing to the global memory is handled correctly. This is particularly important when a variable is being read and also written in a calculations. If multiple processors are simultaneously reading and writing, this could cause spurious results. This

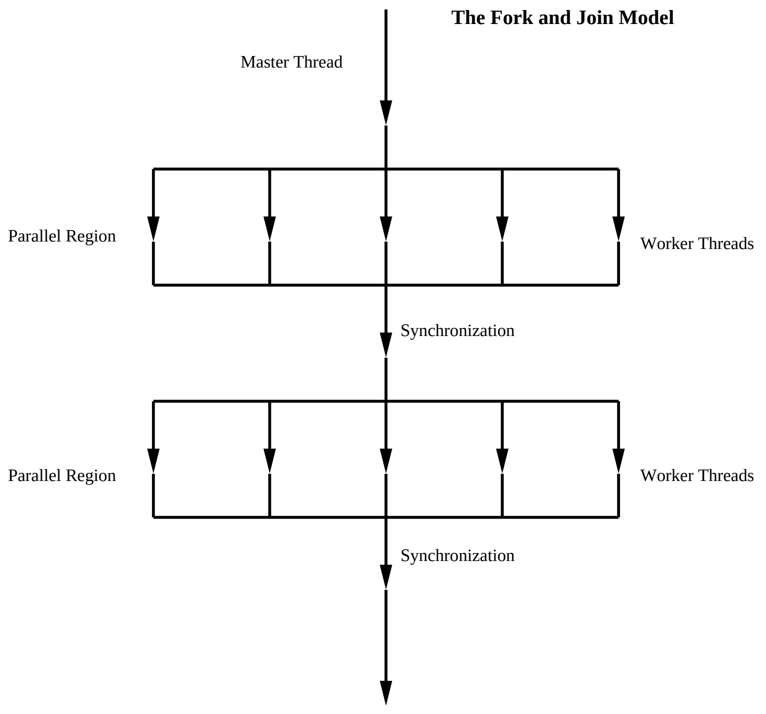

Figure 1.4: The Fork and Join Model. OpenMP programs start with a master thread, running sequentially until they reach a parallel section where they fork into multiple worker threads. At the end of the parallel section the threads re-join into the master thread.

will be discussed in Section 2.3.

1.4 OpenMP

OpenMP (Open Multi-Processing) was first released in 1997 and is a standard Application Programming Interface (API) for writing **shared memory** parallel applications in C, C++ and Fortran. **OpenMP has the advantages of being very easy to implement on currently existing serial codes and allowing incremental parallelisation[5].** It also has the advantages of being widely used, highly portable and ideally suited to multi-core architectures (which are becoming increasingly popular in every day desktop computers).

OpenMP operates on a Fork and Join model of parallel execution and this is shown in Figure 1.4. All OpenMP programs begin as a single process which is called the **master thread**[6]. This master thread executes sequentially until a parallel region

[5]OpenMP allows the programmer to parallelise individual sections of a code, such as loops, one at a time. This allows for testing of each new parallel section before further parallelisation.

[6]A thread is a single sequential flow of control within a program

is encountered. At this point the master thread 'forks' into a number of parallel **worker threads**. The instructions in the parallel region are then executed by this team of worker threads. At the end of the parallel region, the threads synchronise and join to become the single master thread again. Usually you would run one thread per processor, but it is possible to run more.
Parallelisation with OpenMP is specified through **compiler directives** which are embedded in the source code. Examples of these directives will be shown in Chapter 2.

1.5 Who would use OpenMP?

Anyone who uses C, C++ or Fortran can use OpenMP.
If you are using one of these languages and you have a code that takes a while to run in serial, you should consider the advantages of making that code parallel with OpenMP.
Making your code parallel with OpenMP is reasonably easy and if you can already code in C, C++ or Fortran then you most likely already know and understand enough about programming to pick up OpenMP very quickly.

1.6 Availability of OpenMP at the ROE

OpenMP can run on any of the computers at the ROE (and indeed any computer in general). Many of the desktops here at the ROE have dual cores and there are some 4 and 8 core machines[7]. We also have the 32 processor, shared memory machine Cuillin. If you would like an account on Cuillin, you need to contact Eric Tittley, ert@roe.ac.uk. Another option is to get a free account with the Edinburgh Compute and Data Facility (ECDF, down in the Kings Buildings) on their high-performance cluster Eddie. To get an account, see their website at the end of this handout.
The only other thing required to run OpenMP is a compiler that has an OpenMP implementation. If you are coding in C, **gcc** has supported OpenMP since version 4.2 (and greater) and for Fortran, **gfortran** also supports OpenMP. The **Intel** compilers also support OpenMP for C, C++ and Fortran and if you would like access to these

[7]You can see a list of the Linux systems and their specifications at http://intra.roe.ac.uk/atc/computing/hardware/unix/linux.html.

compilers, you need to submit a helpdesk request asking for access.

There are also other compilers that have OpenMP implementations: IBM, Sun Microsystems, Portland Group, Absoft Pro, Lahey/Fujitsu, PathScale, HP and MS. If you wanted to use one of these, you would need to follow up its availability at the ROE with IT support.

As long as you use a compiler that supports OpenMP (gcc, Intel), you can run OpenMP on any computer here at the ROE and anywhere else. Of course, the more cores you have on your machine, the greater the speed-up you can achieve within the limits of scalability.

Chapter 2

How do you make your existing codes parallel?

This chapter will focus on introducing the basics of programming with OpenMP so that you will have the knowledge to go away and make your existing codes parallel. The information provided here is only the very beginning of what there is to learn about OpenMP and if you are serious about making your codes parallel, I strongly suggest that you spend some more time looking up OpenMP tutorials and courses.

The Edinburgh Parallel Computing Center (EPCC) is running a Shared Memory Programming course from **05 - 07 May 2009**. I highly recommend this **free** course to everyone interested in OpenMP. To enroll in this course, see
http://www.epcc.ed.ac.uk/news/epcc-spring-training-courses
and send an email to epcc-support@epcc.ed.ac.uk including your name, the course you wish to attend and your affiliation (University of Edinburgh, Institute for Astronomy).

2.1 How do I compile my code to run OpenMP?

One of the useful things about OpenMP is that it allows the users the option of using the same source code both with OpenMP compliant compilers and normal compilers. This is achieved by making the OpenMP directives and commands hidden to regular compilers.
In this tutorial, I will only consider C and Fortran 90 in the examples. Other versions

of Fortran should use the same OpenMP directives and commands and the examples should be directly translatable. The same holds for C and C++.
The following two sections will show a simple 'HelloWorld' program in C and Fortran 90 and how to compile them.

HelloWorld in C

Here is a very simple little multi-threaded parallel program HelloWorld.c, written in C that will print 'Hello World', displaying the number of the thread processing each write statement.

```
#include 'omp.h'
void main()
{
#pragma omp parallel
  {
    int ID = omp_get_thread_num()
    printf('Hello(%d) ',ID);
    printf('World(%d) \n',ID);
  }
}
```

The first line is the OpenMP include file.
The parallel region is placed between the directive `#pragma omp parallel` {...}.
The runtime library function `omp_get_thread_num()` returns the thread ID to the program.

In order to compile this program from the command line, type

```
> icc -openmp HelloWorld.c -o Hello
```

for the Intel C compiler or

```
> gcc -fopenmp HelloWorld.c -o Hello
```

for the GNU C compiler.

We also need to set the number of threads that the program is going to run on. There are a couple of ways of doing this. First, in the command line you can type

```
> setenv OMP_NUM_THREADS 4
```

for 4 threads in a (t)csh shell or

```
> export OMP_NUM_THREADS=4
```

for 4 threads in a bash shell.

Alternatively, you can specify the number of threads inside the program by including the line

```
omp_set_num_threads(4);
```

before the parallel region directive `#pragma omp parallel`.

A sample output from this program is shown below

```
Hello(5) Hello(0) Hello(3) Hello(2) World(5)
Hello(1) Hello(4) World(0)
World(3)
World(2)
World(1)
World(4)
```

From this output you can see that the program was run on 6 threads (from zero to five) and also that the threads do not necessarily run in any particular order.

HelloWorld in Fortran 90

Here is a very simple little multi-threaded parallel program HelloWorld.f90, written in Fortran 90 that will print 'Hello World', displaying the number of the thread processing each write statement.

```
PROGRAM Hello
  IMPLICIT NONE
  INTEGER ::  OMP_GET_THREAD_NUM, ID
!$OMP PARALLEL
  ID = OMP_GET_THREAD_NUM()
  WRITE(*,*)'Hello(',ID,')'
  WRITE(*,*)'World(',ID,')'
!$OMP END PARALLEL
END PROGRAM Hello
```

The parallel region in this program is defined between the directives `!$OMP PARALLEL` and `!$OMP END PARALLEL`.
The runtime library function `OMP_GET_THREAD_NUM()` returns the thread ID to the program.

In order to compile this program from the command line, type

```
> ifort -openmp HelloWorld.f90 -o Hello
```

for the Intel Fortran compiler or

```
> gfortran -fopenmp HelloWorld.f90 -o Hello
```

for the GNU Fortran compiler.

As with the C program, the number of threads the program is going to run on can be set either in the command line or inside the program. If setting the threads in the command line, the process is exactly the same as the process described in the above section. If you are specifying the number of threads in the program, add the line

```
CALL OMP_SET_NUM_THREADS(4)
```

before the parallel region directive `!$OMP PARALLEL`.
A sample of the output from this program is shown below

```
Hello( 0 )
Hello( 3 )
World( 0 )
Hello( 1 )
World( 3 )
World( 1 )
Hello( 2 )
World( 2 )
```

Compiling Summary

It is very straightforward to compile your programs for OpenMP and I have given examples of how to do this using the command line. For more complex programs with multiple subroutines in separate files, I recommend using a **Makefile** to compile. Detailing how to write a Makefile is beyond the scope of this tutorial but there are many online tutorials to take a user through the process. Just Google 'Makefile Tutorial'.

Note: If you need to compile your codes for serial use, you can just compile without the `-openmp` flag e.g.

Fortran

```
ifort MyProg.f90 -o MyProg
```

C

```
icc MyProg.c -o MyProg
```

They should compile as a regular serial program. When I was testing all of these codes with the Intel compilers, I did get some warnings in the C versions, e.g.

```
Reduc.c(13):  warning #161:  unrecognized #pragma
```

These are just warnings and the lines are ignored so they will not effect the running of the program in serial.
There are however a few things that won't work in a serial compilation. In the `HelloWorld` example above, there are some OpenMP specific calls that are not preceded by the OpenMP sentinels (`!$OMP` for Fortran and `#pragma omp` for C)

Fortran

```
ID = OMP_GET_THREAD_NUM()
```

C

```
int ID = omp_get_thread_num()
```

These calls and any other OpenMP calls that are not preceded by the sentinels will not run in serial mode and should be removed if you plan on running your code in serial.

2.2 How do I decide if a loop is parallel or not?

Loops are the most common parts of a code to parallelise. When run in parallel, a loop will divide its iterations between multiple threads. Not all loops can be parallelised though so it is important to think about what the dependencies are in the loop. One useful test is to consider if the loop will give the same answers if it is run backwards. If this works then the loop is almost certainly parallelisable.

Example 1:

Fortran

```
DO i = 2,n
  a(i) = 2 * a(i-1)
END DO
```

C

```
for (i=1; i<n; i++)
{
  a[i] = 2 * a[i-1];
}
```

In this example, `a(i)` depends on `a(i-1)`, so it can not be parallelised.

Example 2:

Fortran

```
DO i = 2,n
  b(i) = (a(i) - a(i-1)) * 0.5
END DO
```

C

```
for (i=1; i<n; i++)
{
  b[i] = (a[i] - a[i-1]) * 0.5;
}
```

In this case, all of the iterations are independent so it is possible to make this loop parallel.

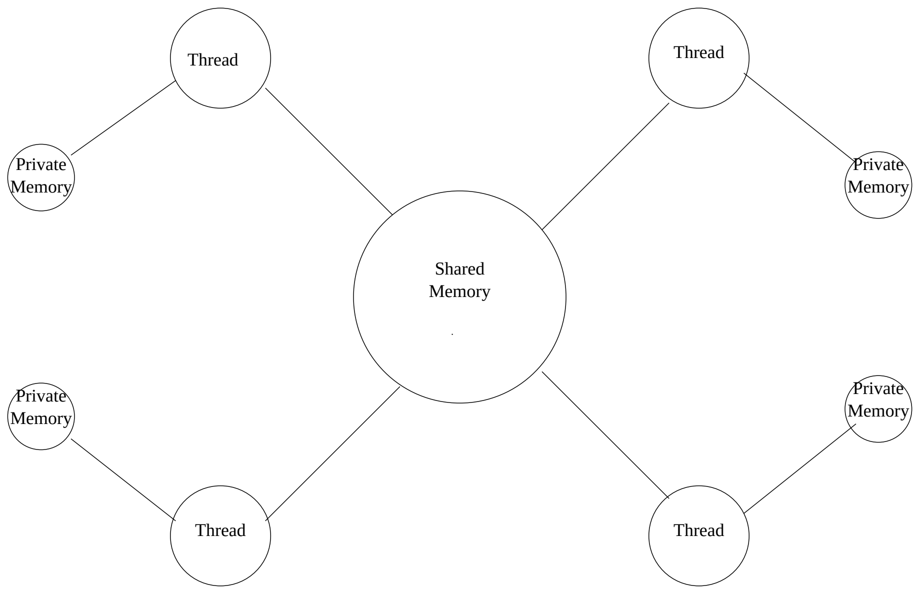

Figure 2.1: An example of the OpenMP memory model where each thread has access to the shared memory as well as its own private memory that none of the other threads can see or touch.

2.3 What are Private and Shared variables?

Inside a parallel region of a program, variables can either be **shared** or **private** or **default**. Figure 2.1 shows an example configuration for an OpenMP program with both shared and private memory addresses.

Fortran

`SHARED(`*list*`)`
`PRIVATE(`*list*`)`
`DEFAULT(SHARED|PRIVATE|NONE)`

C

`shared(`*list*`)`
`private(`*list*`)`
`default(shared|none)`

There are other clauses available in OpenMP but for the purposes of this tutorial, these are the only clauses that I will consider.

In this model, all threads can see the same copy of shared variables and all threads can read or write shared variables.
Each thread has its own copy of a private variable that is invisible to all other threads. A private variable can only be written by its own thread.
We define the scope of the variables because it is up to the programmer to make sure that memory is accessed correctly (otherwise the code may give rubbish outputs).

The Default Clause

The Default clause allows the user to specify the default scope for all of the variables in the parallel region. If this is not specified at the beginning of the parallel region, the default is automatically *shared*. Specific variables can be exempted from the default by using one of the other clauses.

Fortran

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(a,b)
```

C

```
#pragma omp parallel default(private), shared(a,b)
```

The above declarations state that all the variables in the parallel region are private apart from variables `a` and `b` that are shared.
If you specify `DEFAULT(NONE)`, you are required to declare each variable inside the parallel region explicitly. This is similar to using `IMPLICIT NONE` in Fortran. I would recommended this option because it forces the programmer to consider the memory status of each variable specifically and can avoid memory access problems if a variable is forgotten in a default declaration.

The Shared Clause

Most variables are shared and these variables exist only in one memory location and all threads can read or write to that address. When trying to decide whether a variable is shared, consider what is happening to it inside the parallel region. Generally *read only* variables are shared (they can not accidently be over written) and *main arrays* are shared.

Fortran

```
!$OMP PARALLEL DEFAULT(NONE), SHARED(a,b)
```

C

```
#pragma omp parallel default(none), shared(a,b)
```

In this example, the variables `a` and `b` are the only variables in the parallel region and they are both shared. We know that they are the only variables in this region because we specified `DEFAULT(NONE)` first which requires that all variables be explicitly defined.

The Private Clause

A variable that is private can only be read or written by its own thread. When a variable is declared as private, a new object of the same type is declared once for each thread and all references to the original object are replaced with references to the new object.
By default, *loop indices* are private and *loop temporaries* are private.

Fortran

```
!$OMP PARALLEL DEFAULT(NONE), SHARED(a,b), PRIVATE(i,j)
```

C

```
#pragma omp parallel default(none), shared(a,b), private(i,j)
```

In this example, the only variables in the parallel region are `a`, `b`, `i` and `j`. The former pair are shared and the latter two are private.

Example 1:
This first example is a very simple loop that illustrates shared and private variables and the syntax of writing an OpenMP program. The program is initialising two arrays and writing out the contents of these arrays to the screen.

Fortran

```
PROGRAM loop
  IMPLICIT NONE
  INTEGER ::  i
  INTEGER, PARAMETER ::  n=10
  REAL ::  a(n),b(n)

!$OMP PARALLEL DEFAULT(NONE), PRIVATE(i), SHARED(a,b)
!$OMP DO
  DO i = 1,n
    a(i) = i * 1.0
    b(i) = i * 2.0
    WRITE(*,*)i,a(i),b(i)
  END DO
!$OMP END PARALLEL
END PROGRAM loop
```

C

```
#include <omp.h>
#define N 10

int main(void) {
  float a[N], b[N];
  int i;

#pragma omp parallel default(none), private(i), shared(a,b)
  {
#pragma omp for
    for (i = 0; i < N; i++) {
      a[i] = (i+1) * 1.0;
      b[i] = (i+1) * 2.0;
      printf("%d, %f, %f \n",i+1,a[i],b[i]);
    }
  }
}
```

In these examples, the loop index `i` is private. Loop indexes are always private. The two arrays being initialised are shared. This is not a problem because each calculation is independent of the other calculations and at no point are the threads going to try and write to the same memory address.
This very basic loop gives the output:

```
1 1.000000 2.000000
2 2.000000 4.000000
5 5.000000 10.00000
6 6.000000 12.00000
9 9.000000 18.00000
10 10.00000 20.00000
3 3.000000 6.000000
4 4.000000 8.000000
7 7.000000 14.00000
8 8.000000 16.00000
```

The ordering of the numbers will most likely change each time you run the program depending on which threads complete their calculation first.

Example 2:
This example extends on the previous example by adding a new loop and some new variables. The new loop determines the number of threads that it is working on and writes this information to the screen. It then writes to screen when each thread starts its calculations. Then the two arrays initialised in the previous loop are added together to create a new array. The thread number, index and values of all three arrays are then written to screen.

Fortran

```
   PROGRAM Floop2
     IMPLICIT NONE
     INTEGER ::  nthreads, i, TID
     INTEGER ::OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
     INTEGER, PARAMETER ::  n=10
     REAL a(n), b(n), c(n)

     CALL OMP_SET_NUM_THREADS(4)

  !$OMP PARALLEL DEFAULT(NONE), PRIVATE(i), SHARED(a,b)
  !$OMP DO
     DO i = 1,n
       a(i) = i * 1.0
       b(i) = i * 2.0
     END DO
   !$OMP END PARALLEL

   !$OMP PARALLEL DEFAULT(NONE), PRIVATE(i,TID), SHARED(a,b,c,nthreads)
     TID = OMP_GET_THREAD_NUM()
     IF (TID .EQ. 0) THEN
       nthreads = OMP_GET_NUM_THREADS()
       WRITE(*,*)'Number of threads =', nthreads
     END IF

     WRITE(*,*)'Thread',TID,' starting...'

   !$OMP DO
     DO i = 1, n
       c(i) = a(i) + b(i)
       WRITE(*,*) TID,i,a(i),b(i),c(i)
     END DO
   !$OMP END DO
   !$OMP END PARALLEL
   END PROGRAM Floop2
```

C

```
#include <omp.h>
#define N 10
int main(void) {
  float a[N], b[N], c[N];
  int i, TID, nthreads;

  omp_set_num_threads(4);

#pragma omp parallel default(none), private(i), shared(a,b)
  {
#pragma omp for
    for (i = 0; i < N; i++) {
      a[i] = (i+1) * 1.0;
      b[i] = (i+1) * 2.0;
    }
  }
#pragma omp parallel default(none), private(i,TID), shared(a,b,c,nthreads)
  {
    TID = omp_get_thread_num();
    if (TID == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = (%d) \n",nthreads);
    }

  printf("Thread %d starting \n",TID);

#pragma omp for
    for (i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
     printf("%d, %d, %f, %f, %f \n",TID,i+1,a[i], b[i],c[i]);
    }
  }
}
```

This program includes an `IF` statement as well as an extra loop. In this program, we define the number of threads that we would like to use at the start. The variables declared in Example 1 are the same in this program and additionally we have the variable `TID` which is private because it is unique to each thread. The array `c` is shared because the calculation of each element is independent of all other threads, similar to arrays `a` and `b`.
The output of this loop will look something like this:

```
Number of threads = 4
Thread 0 starting...
0 1 1.000000 2.000000 3.000000
0 2 2.000000 4.000000 6.000000
0 3 3.000000 6.000000 9.000000
Thread 3 starting...
3 9 9.000000 18.00000 27.00000
3 10 10.00000 20.00000 30.00000
Thread 1 starting...
1 4 4.000000 8.000000 12.00000
1 5 5.000000 10.00000 15.00000
1 6 6.000000 12.00000 18.00000
Thread 2 starting...
2 7 7.000000 14.00000 21.00000
2 8 8.000000 16.00000 24.00000
```

It can also look like this:

```
Number of threads = 4
Thread 3 starting...
3 9 9.000000 18.00000 27.00000
Thread 0 starting...
3 10 10.00000 20.00000 30.00000
0 1 1.000000 2.000000 3.000000
Thread 1 starting...
Thread 2 starting...
0 2 2.000000 4.000000 6.000000
1 4 4.000000 8.000000 12.00000
```

```
2 7 7.000000 14.00000 21.00000
0 3 3.000000 6.000000 9.000000
1 5 5.000000 10.00000 15.00000
2 8 8.000000 16.00000 24.00000
1 6 6.000000 12.00000 18.00000
```

The output will look different almost every time you run the program because the threads will execute the calculations in different orders and at different speeds.

2.4 How can I do Summations?

A **reduction** clause allows you to get a single value from associative operations such as addition and multiplication.
Without the reduction clause, you would need to define a variable as private or shared. If you make the variable private, all copies of it are discarded at the end of the parallel region which means that you can't use their values to complete the sum on exit from the parallel loop. If you declare the variable as shared, all the threads will be reading and writing to the memory address simultaneously which won't work properly for a summation. You will most likely end up with garbage at the end of the loop.
So, by declaring the variable as a reduction, private copies of the variable are sent to each thread and at the end of the parallel region, the copies are summarised (reduced) to give a global shared variable.

Example:
This example shows how to calculate a dot product using the reduction clause. The program first initialises the arrays that are to be used in the dot product calculation and then it performs the calculation.

Fortran

```
PROGRAM Dot_Product
   IMPLICIT NONE
   INTEGER ::  i
   INTEGER, PARAMETER ::  n=10
   REAL ::  a(n), b(n), result
```

```
  CALL OMP_SET_NUM_THREADS(4)

!$OMP PARALLEL DEFAULT(NONE), PRIVATE(i), SHARED(a,b)
!$OMP DO
  DO i = 1,n
    a(i) = i * 1.0
    b(i) = i * 2.0
  END DO
!$OMP END PARALLEL

  result = 0.0

!$OMP PARALLEL DEFAULT(NONE), PRIVATE(i), SHARED(a,b) REDUCTION(+:result)
!$OMP DO
  DO i = 1, n
    result = result + (a(i) * b(i))
  END DO
!$OMP END PARALLEL

  WRITE(*,*)'Final Result= ',result

END PROGRAM Dot_Product
```

C

```
#include <omp.h>
#define N 10

main () {
  int i;
  float a[N], b[N], result;

  omp_set_num_threads(4);

#pragma omp parallel default(none), private(i), shared(a,b)
```

```
    {
  #pragma omp for
      for (i = 0; i < N; i++) {
        a[i] = (i+1) * 1.0;
        b[i] = (i+1) * 2.0;
    }
  }
    result = 0.0;

  #pragma omp parallel default(none), private(i), shared(a,b) \
    reduction(+:result)
    {
  #pragma omp for
      for (i=0; i < N; i++)
        result = result + (a[i] * b[i]);
    }
    printf("Final result= %f \n",result);
  }
```

The output from this program is:

```
Final result= 770.000000
```

2.5 Summary

This short introduction to OpenMP has attempted to give you the basic knowledge required to make your own codes parallel using OpenMP. Only the very basics have been covered and there are much more sophisticated things that you can do with OpenMP. I strongly encourage people who are interested in using OpenMP in the future to attend a course and look up online tutorials.

2.6 Some useful websites

Official OpenMP website
http://openmp.org/wp/

OpenMP, Lawrence Livermore National Laboratory
https://computing.llnl.gov/tutorials/openMP/

OpenMP, Wikipedia
http://en.wikipedia.org/wiki/OpenMP

Edinburgh Parallel Computing Center training courses
http://www.epcc.ed.ac.uk/news/epcc-spring-training-courses

The Edinburgh Compute and Data Facility (ECDF)
http://www.ecdf.ed.ac.uk/

Intel compiler documentation
http://software.intel.com/en-us/intel-compilers/

gcc compiler documentation
http://gcc.gnu.org/onlinedocs/

gfortran compiler documentation
http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gfortran/

GNU gprof profiler documentation
http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html