Comp3402 Intro to OpenMP Multiprocessing

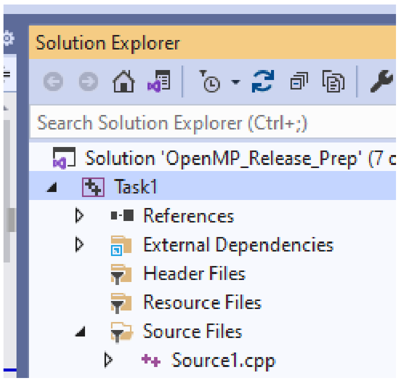
C.B.Price March 2022

Purpose	(i) To investigate basic concepts of the OpenMP (ii) To investigate its application to vector and matrix problems
Files Required	Visual Studio Solution comprising various tasks
ILO Contribution	6
Send to Me	
Assignment Info.	
Homework	Read Chapter 8

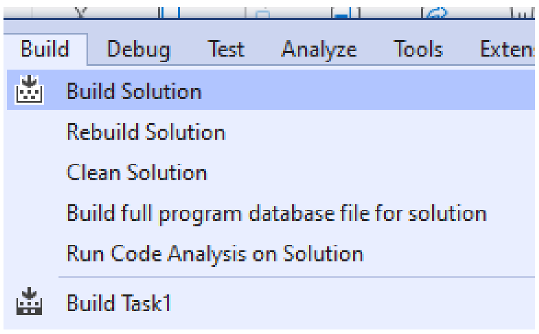
Activities

Workflow. Here you will proceed as follows:
(1) Open up the Visual Studio solution (sln), then follow the following steps.

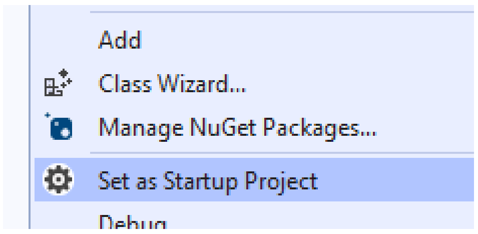
(2) Select the Task



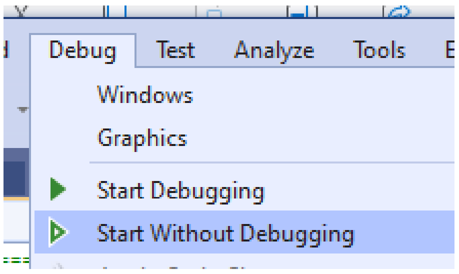
(4) Select Build the Task



(3) Right-click on task and set as Startup Project



(5) Start without debugging



1 Hello Cat

(a) Open up **Task1** and Source1.cpp. Look for the following

```
#pragma omp parallel
{
        tid = omp_get_thread_num();
        printf("In parallel region: Thread %d printf Hello Cat\n", tid);

}
```

The pragma causes OpenMP to start a team of threads. “Hello Cat” is printed out together with the thread id (tid) which is printing it.

(b) Run the task and you will be surprised.

2 Proof that updating array elements is shared between threads.

(a) Open up **Task2** and have a look at the source code. The computation here is simply the assignment of values of an array of length ARRAY_LENGTH like this

```
a[i] = 3 * i;
```

(b) Run the code and interpret the info sent to the console. Do you agree that this ‘proves’ that each array assignment is done by a different thread? This is the meaning of ‘parallel processing’.

3 Using a Critical Region

We’ve already seen the need to protect a shared resource or a read-update-write variable for the RTOS system. Here we shall review this for multitasking. It’s a bit of a toy problem, but it’s worthwhile exploring.

We have an array **a[i]** and we wish to find its sum. We divide the work across a team of threads which each create a partial sum of some array elements. We do this like this

```
#pragma omp for
for(i=0; i< n; i++)
  sumLocal += a[i];
```

where **sumLocal** is *private* so each thread has its own copy. Finally we have to add the **sumLocal** partial sums to give the total sum. We would do this

```
sum += sumLocal;
```

(a) Open up **Task3** and look at the code. You will see that the calculation of the total sum is protected by a critical region so that *only one thread at a time* can update sum.

(b) Run the task several times, and you should always get the correct sum (45).

(c) Comment out the critical region pragma and its braces. Recompile and run several times. You will not get the correct result.

4 The Reduction directive

Calculating such as sum as in activity 3 is quite common, so OpenMP provides an explicit clause for doing this. It’s appended to the parallel pragma. Let’s have a look.

(a) Open up **Task 4** and look at the code. Here in the pragma omp for, you will see that the sum is computed directly without partial sums. Look for **#pragma omp parallel reduction(+:sum)** which gets the compiler to correctly parallelize the summation.

(b) Run the code with the above pragma in place and you should get the correct sum every time.

(c) Now replace the full pragma with this **#pragma omp parallel** no longer directing for reduction. The sums should be all over the place.

5 Vector Multiplication and Efficiencies of multiprogramming

Here we are going to parallelize the vector operation **c[i] = a[i]*b[i]** and see how the run time varies with the number of threads

(a) Open **Task 5** and make sure you can find the parallel region. Also look for where you can set the number of threads in a #DEFINE at the top of the code and also the PROBSIZE.

(b) Set the threads to 1 and time the operation for an increasing PROBSIZE. This will be machine dependent, but I managed a range of PROBSIZE 1000 – 20000. Use the Octave script provided to plot time vs. PROBSIZE.

(c) Now increase the number of threads and repeat the above, for a range of numbers of threads. This will generate a lot of data, so you may like to finish this work for the Paper Section 2.

(d) Now set the PROBSIZE to something very small, say 10 or 20. Time the program for 1 thread and for the maximum you have on your system. You might find something interesting. Can you explain this?