Comp3402 Barriers, Functions and Races

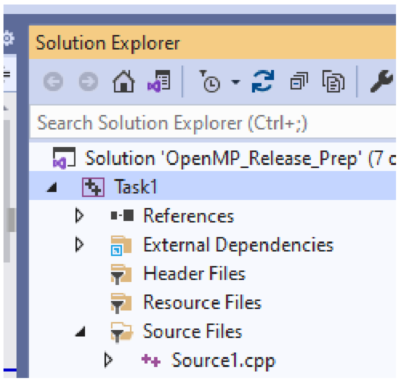
C.B.Price March 2022

Purpose	(i) To investigate the barrier concept (ii) to consider the concept of thread-safe functions (iii) to observe a data-race condition where a loop may not be parallelized
Files Required	Visual Studio Solution comprising various tasks
ILO Contribution	6
Send to Me	
Assignment Info.	
Homework	Read Chapter 8

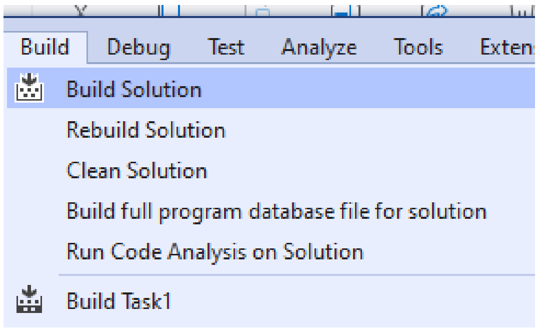
Activities

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

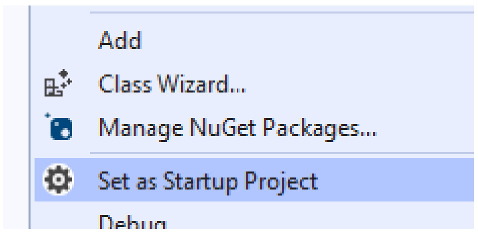
(2) Select the Task



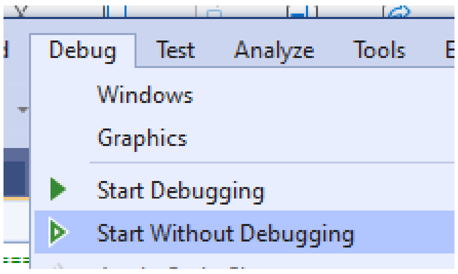
(4) Select Build the Task



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



(5) Start without debugging



1 Investigating the Barrier Construct

A *barrier* is a point in execution flow where all threads wait for each other. There are many constructs which produce an *implicit* barrier (next activity), here we shall look at a small toy problem where a barrier is inserted *explicitly* using the **#pragma omp barrier** construct.

(a) Open up **Task 8** where you will see some strange code. This code assume we have more than 4 threads, so you may need to change it. In the parallel region, threads 0 to 3 are put to sleep for 1000 ms. All threads print out their run time. The threads with out delay hit the barrier first, and they wait for the delayed threads to catch up. After the barrier, the entire team of threads runs again and prints out their run-times.

(b) Compile and run the task. Look at the timings and convince yourself the timings show that the barrier is working.

2 Implied Barriers

(a) Open up **Task 9** where various pragmas have been commented out. Don't change these, yet. There are two parallel loops: the first assigns values to vector **a** and the second uses these to compute the values of vector **b.** You will see that the computations of both loops get mixed up. That's because some threads are finished in the first loop so proceed to the second while other threads are still in the first loop.

(b) Now uncomment the **//#pragma omp barrier** to put in an explicit barrier to make all threads wait until they have all finished the first loop. You should find that all first loop assignments are completed before the second loop starts.

(c) Now re-comment out the explicit barrier so it is not there. Then uncomment the **//pragma omp for** on the *first* loop. You should find the results are the same as in (b). So this demonstrates that the construct **#pragma omp for** inserts an ***implicit barrier*** at the end of the for loop.

3 Thread-Safe Functions (or not)

We often use functions from libraries which we have not coded, so we do not know what (in detail) may be inside. "Thread Safe" functions are those which can be accessed at the same time from multiple threads and return correctly. A typical case where a function is not thread-safe is where it makes use of global variables. This must be explicitly fixed.

(a) Open up **Task 10** where you will find a toy library function **lib_func()** which makes use of the global variable **count.** The function is called in the parallel region. Run the code several times and you should see that **count** does not stay the same. What value should it have.

(b) You can fix the problem by protecting the update to count, **count++.** This can be done by putting it in a critical region using **#pragma omp critical**. Try it out.

4 Loops that can’t be parallelized – Data Races

Not all loops can be parallelized. Here’s an example **a[i] = a[i + 1] + b[i];** The problem is that the loop iterations are dependent on each other. To calculate **a[i]** we need the *current* value of **a[i+1]**. But perhaps the thread responsible for the a[**i+1]** update has already executed before the thread responsible for updating **a[i]**, in which case the value of **a[i+1]** is incorrect.

Look at **Task11** where this situation is explored. You should find the processing yields different results for each iteration of the while loop (which calls the parallel code).

5 The Sections construct

This is the easiest of the work-sharing approaches, where we specify an areas of code each of which is to be executed by a single thread. It’s most common use is to *execute functions in parallel*.

(a) Open up **Task 12** where you will see there are two functions **func1()** and **func2()** which (if this code was not parallelized) would run in sequence. Each function simply prints out which thread is executing its body and contains a dummy loop. The **sections** construct has been commented out. Run the code and look at the order of execution of the functions and which thread is working on each function.

(b) Re-instate the **sections** construct, compile and re-run. You should find that each function is executed by a single thread.