Lab: Threading with OpenMP for Windows

Time Required: Thirty minutes

Objective:

In this lab, you will make the Hello World program parallel. Then, you will thread a numerical integration code to compute the value of Pi. Finally, the last lab will have you thread a version of the Pi using a Monte Carlo technique and the Math Kernel Library.

At the successful completion of this lab, you will be able to:

- Use the most common OpenMP* C statements
- Compile and run an OpenMP* program
Activity 1: Hello Worlds

In this activity, you make a "Hello, Worlds" program parallel.

Initial Compile

1. Within a "Build Environment for IA-32 applications" command window, move to the HelloWorlds directory:
   ```bash
cd OpenMP/HelloWorlds
   
```
2. Compile serial code using the Intel compiler:
   ```bash
   icl HelloWorlds.c
   
```
3. Run the program:
   ```bash
   HelloWorlds.exe
   
```

Add OpenMP Directives

1. Add an OpenMP parallel directive to run the first four lines of main in parallel, but not the last line:
   ```c
   printf("GoodBye World\n");
   #pragma omp parallel
   {
      ...[Code to run in Parallel goes here]...
   }
   
```
2. Compile in "Serial Mode" using the Intel compiler:
   ```bash
   icl HelloWorlds.c
   
```
3. Notice the pragma warning statements.
4. Fix any syntax errors.
5. Run the program:
   ```bash
   HelloWorlds.exe
   
```

OpenMP Compile

1. Compile in OpenMP mode using the Intel compiler:
   ```bash
   icl /Qopenmp HelloWorlds.c
   
```
2. Enable the environment for multiple OpenMP threads:
   ```bash
   set OMP_NUM_THREADS=2
   
```
3. Run the program in a multithreaded environment:
4. Run the program multiple times, verifying if the results are the same every time.

Extra Activities

1. Change the code that you make parallel (for example, put the int i outside the parallel region).
2. Play with the OMP_NUM_THREADS settings. As you change the number of threads, do you get the results you expect?
Activity 2: Computing Pi with Numerical Integration

In this activity, you will make the Pi program parallel:

1. Within Windows Explorer, move to the Pi directory (classfiles\OpenMP\Pi).


3. From the Debug menu, choose the "Start without Debug (Ctrl+F5)" command to run the serial application.

4. Record elapsed time: ________________.

Add OpenMP Directives

1. Determine the section of code to make parallel and add the OpenMP parallel directive:
   
   ```
   #pragma omp parallel
   {
   ....[Code to run in Parallel goes here]...
   }
   ```

2. Find the loop to make parallel and insert a worksharing pragma:
   
   ```
   #pragma omp for
   for(xxx; yyy; zzz)
   {
   //Loop body
   ```

3. Examine all variables and determine which ones need to be specially declared. The following may be handy:
   
   ```
   #pragma omp parallel private(varname,varname)\ 
   reduction(+:varname,varname) \ 
   shared(varname,varname)
   {
   ....[Code to run in Parallel goes here] ....
   }
   ```

4. Depending on your implementation you may need the following. For any remaining shared variables add appropriate locks, if you update that variable.
#pragma omp critical
{
    ...[Code in Critical section goes here] ...
}

OpenMP Compile and Run

1. Add the /Qopenmp flag to the compilation of the application. Since the project is using the Intel compiler, you will find an "Intel Specific" section in the project Property Pages -> C/C++ -> Language -> "Process OpenMP Directives". Set this to "Generate Parallel Code (/Qopenmp)" and click OK.
2. Build and run the program in a multithreaded environment:
3. Record the time: ________________

What is the speedup? (serial time / parallel time): ________________
Activity 3: Monte Carlo Pi

In this activity, you will use the Monte Carlo technique to approximate the value of Pi parallel. The Intel® Math Kernel Library will be used to generate random numbers.

Setup

1. With Windows Explorer, locate the Monte Carlo Pi directory:
   classfiles\OpenMP\Monte Carlo Pi

Initial Compile

1. Double click on the Microsoft* Visual Studio* Solution icon (Monte Carlo Pi.sln)
2. Build the solution; from the Debug menu, choose the "Start without Debug (Ctrl+F5)" command to run the application.
3. Record Pi: __________________
4. Record the serial time: __________________

Add OpenMP Directives

1. Determine the section of code to make parallel and add the OpenMP parallel directive:
   #pragma omp parallel
   {
      ...[Code to run in Parallel goes here]...
   }
2. Find the loop to make parallel and add the following:
   #pragma omp for
   for(...) {
   }
3. Examine all variables and determine which ones need to be specially declared. There are hints within the source code with regards to some variables and arrays that need to be private to each thread. The following may be handy:
   #pragma omp parallel private(varname,varname) \ 
   reduction(+:varname,varname) \
   shared(varname,varname)
   {
      ...[Code to run in Parallel goes here]...
   }
4. Depending on your implementation you may need the following. For any remaining shared variables 
add appropriate locks, if you update that variable.

```
#pragma omp critical [name]
{
    ...[Code in Critical section goes here]...
}
```

**OpenMP Compile**

1. Add the /Qopenmp flag to the compilation of the application. Since the project is using the Intel 
compiler, you will find an "Intel Specific" section in the project Property Pages -> C/C++ -> Language 
-> "Process OpenMP Directives". Set this to "Generate Parallel Code (/Qopenmp)" and click OK.

2. Run the program in a multithreaded environment:

3. Record Pi: ______________________

4. Record the parallel time:_______________

5. What is the speedup? ________________

**Extra Activities**

1. Replace vsl routines with ascii rand, random, or rand48 functions, and try to get parallel speedup (this 
maybe impossible).

2. Play with the BLOCK_SIZE settings.
Review Questions

Name the pragma or directive that would split a loop into multiple threads.

What website has the very readable OpenMP spec?