Lab Exercises - Producing Parallel Code

Explanation about Activity numbering convention

The activities in this lab are based on chapters 6, 7, 8 and 9 of the book””, Activities 6-1 to 6-5 are based on *Chapter 6 - Where to Parallelize*; activities numbed 7-1 … are based on *Chapter 7 - Implementing Parallelism*; activities numbered 8-1 … are based on *Chapter 8 - Checking for Errors*; activities numbered 9-1 … are based on *Chapters 9 – Tuning Parallel Applications*.

Step 1 Hotspots Analysis

ACTIVITY 6-1: BUILDING THE EXAMPLE APPLICATION

In this activity you build and run the Mandelbrot program.

1. Open an Intel Parallel Studio XE command prompt.
   - **Windows**

     Start | All Programs | Intel Parallel Studio 2013 | Command Prompt | Parallel Studio XE with Intel Compiler | Intel 64 Visual Studio
   - **Linux**

   To make the Parallel Studio XE tools available from a shell, source the following scripts (or add the commands to your ./bash_profile):

```
source /opt/intel/composerxe/bin/compilervars.sh  intel64
source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
source /opt/intel/inspector_xe/inspxe-vars.sh
```

   In Linux export the EDITOR environment variable to point to your favourite editor e.g. :

```
export EDITOR=gedit
```

2. Navigate to the `Chapter6` folder.

3. Build the program with the following command:
   - **Windows**

```
icl /O2  /Qipo  wtime.c  main.cpp  mandelbrot.cpp  ppm.cpp -o  6-1.exe
```

   - **Linux**

```
icc -O2  -ipo wtime.c main.cpp mandelbrot.cpp  ppm.cpp -o 6-1.exe
```

4. Run the program `6-1.exe` you have just created and record the time taken in the table below.

`Time to calc`	
`Time to print`	
`Time (Total)`	

5. Examine the files `main.cpp mandelbrot.cpp ppm.cpp` (Hint, you can open them in visual studio by just typing their names on the command line).

By looking at the source code can you guess which function and loops take the most time to execute? Make a note below:

Function that I think takes the most time to execute	
Loop that I think takes the most time to execute	

6. Examine the generated `default.ppm` file with a PPM viewer (optional).

On Linux systems you should have a default PPM viewer installed, such as gthumb, eog, or gwenview.

On Windows you can view the default.ppm file with the program IrfanView.exe which is available from www.irfanview.com

ACTIVITY 6-4: CONDUCTING A HOTSPOT ANALYSIS WITH AMPLIFIER XE

In this activity you carry out a hotspot analysis on the Mandelbrot program with Amplifier XE.

1. Make sure you have carried out Activity 6-1.

2. Rebuild the application, adding the /Zi flag (Linux –g) to generate debug information, and the /Ob1 flag (Linux: -inline-level=1) which reduces the amount of code that the compiler inlines (Functions can disappear because of compiler inlining. Adding the /Ob1 option to the build improves visibility):

- **Windows**
```
icl /02 /Qipo /Zi /Ob1 wtime.c main.cpp mandelbrot.cpp ppm.cpp
-o 6-4.exe
```
- **Linux**
```
icc -02 -ipo -g –inline-level=1 wtime.c main.cpp mandelbrot.cpp
ppm.cpp -o 6-4.exe
```

3. Start an Amplifier XE GUI from the command line:

```
amplxe-gui
```

4. Create a new project named Chapter 6.

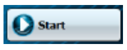
   a. Select File | New | Project.

   b. In the Project Properties dialog, make sure the Application Field points to your Mandelbrot application, then press <OK> to close the dialog box.

5. Carry out a hotspot analysis by

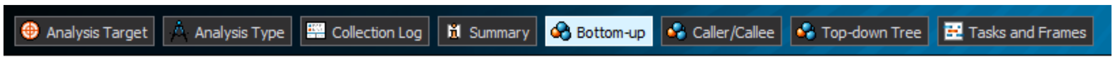
   a. selecting File | New | Analysis . . .

   b. Choose the analysis type Algorithm Analysis | Basic Hotspots

   c. Press the  button to start the analysis

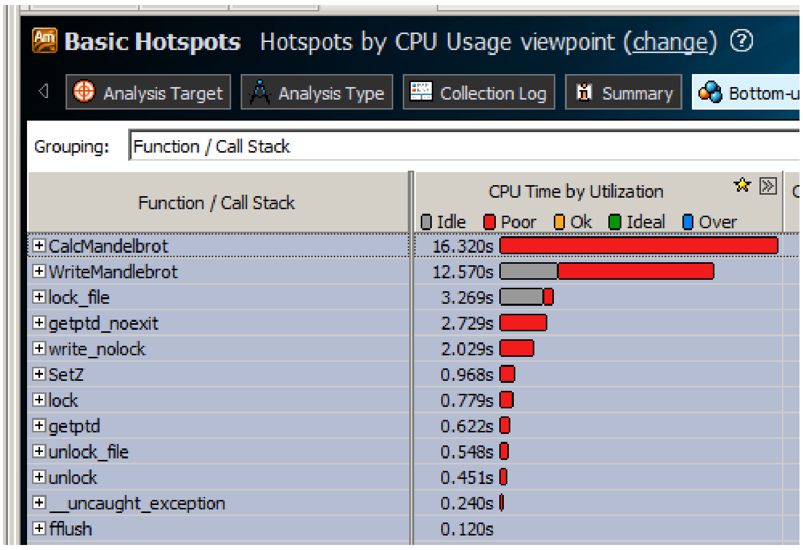
Examining the Hottest Function and Hottest Loops

6. From the results Pane press the bottom-up button view



You should see a graph similar to the picture below.

NB: make sure the ‘Grouping‘ is **`Function/Call Stack`**


Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (▯ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)
⊞ CalcMandelbrot	16.320s
⊞ WriteMandlebrot	12.570s
⊞ lock_file	3.269s
⊞ getptd_noexit	2.729s
⊞ write_nolock	2.029s
⊞ SetZ	0.968s
⊞ lock	0.779s
⊞ getptd	0.622s
⊞ unlock_file	0.548s
⊞ unlock	0.451s
⊞ __uncaught_exception	0.240s
⊞ fflush	0.120s

Double-click the largest hotspot. The source view should be displayed.

Question:

Which line of code takes the most time?

Manual Traversal of the hottest loops

7. In the stack pane (on the right of the source view), manually trace back up the call stack (by double-clicking the call stack entries) until you find code that has a loop in it.

NB: each time you double click on the stack, a new code pane will open!

You should be able to find the best place to add parallelism by doing this manual stack traversal.

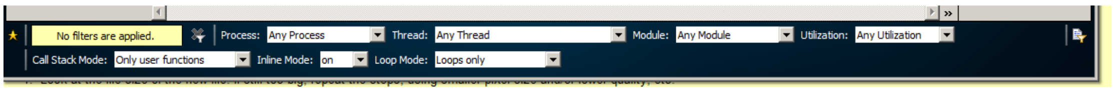
Question:

Is this loop the same as the one you identified in Activity 6-1 (Step 5)?

Turning on the loop mode.

8. Go back to the Bottom-up view.

9. At the bottom of the results pane there are a set of drop down filters, change the Loop Mode from ‘Functions only’ to ‘Loops only’:


No filters are applied.

Process: Any Process | Thread: Any Thread | Module: Any Module | Utilization: Any Utilization

Call Stack Mode: Only user functions | Inline Mode: on | Loop Mode: Loops only

You will find that this is the easiest way to identify the hottest loops.

Double click on the hottest two loops and examine the code.

Question:

Is this loop the same as the one you identified in Activity 6-1 (Step 5)? If not can you think of a reason why not?

10. Optional Activity – Loop Profiling using the Intel Compiler

If you have time, try to find the hotspots using the profiling features of the Intel compiler. The instructions *are at the end of this lab sheet.*

Step 2 – Implementing Parallelism

ACTIVITY 7-1: Adding parallelism

In this activity you add parallelism to the code using an OpenMP for loop.

1. Try adding an OpenMP loop to the outer loop in the function Mandlebrot (Mandlebrot.cpp line 35)

```
#pragma omp parallel for
for (int i=0; i<maxI; i++) {
    for (int j=0; j<maxJ; j++) {
      SetZ(i, j, xinc, yinc);
   }
}
```

2. Build the program with the following command:

- **Windows**

```
icl /O2 /Qipo /Qopenmp wtime.c main.cpp mandelbrot.cpp  ppm.cpp
-o 7-1.exe
```

- **Linux**

```
icc -O2  -ipo –openmp wtime.c main.cpp mandelbrot.cpp  ppm.cpp
-o 7-1.exe
```

3. Run the program `7-1.exe` you have just created and record the time taken in the table below. Calculate the speed up using the formula Speedup = Original Time / New Time.

	Time 6-1.exe (A)	Time 7-1.exe (B)	Speedup (A/B)
Time to calc			
Time to print			
Time (Total)			

4. *Optional Activity – Using Cilk Plus and TBB*

If you have time, try to re-implement this loop parallelism using (see slides for examples).

(a) *Cilk Plus*
(b) *TBB*

Step 3 - Debugging Parallel Errors

Warning- Step 3 uses a different application from the previous steps!

ACTIVITY 8-1: DETECTING AND FIXING A DEADLOCK

In this activity you use Inspector XE to detect and fix a deadlock. You can run this activity on Linux or Windows.

Building and Running the Program

1. Copy the source from the `Chapter8` folder into a new folder.
2. Traverse to the new folder.
3. Build the program with the following command:

- WINDOWS

```
icl   /Od  /Qopenmp   /Zi  pi.cpp  -o  8-1.exe
```

- LINUX

```
icc    -O0 -openmp  -g  pi.cpp  -o  8-1.exe
```

4. Run the program:

```
8-1.exe
```

You should see that the program will hang.

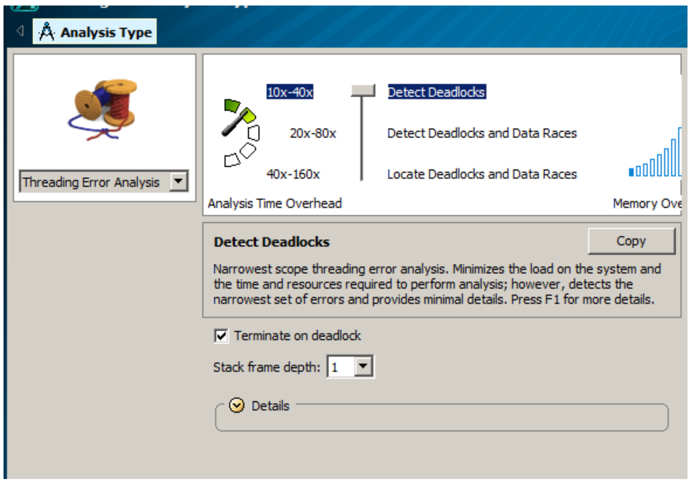
Detecting and Fixing the Deadlock

5. Start the Inspector XE GUI from the command line:

```
inspxe-gui
```

6. Create a new project named Chapter 8:
   - Select File | New | Project.
   - In the Project Properties dialog, fill in the application details.
7. Carry out a Detect Deadlocks analysis:
   - Select File | New | Analysis.
   - Select ‘Threading Error Analysis’ from the drop down box (see diagram below).
   - Set the Analysis depth to be “Detect Deadlocks”
   - Make sure the ‘Terminate on deadlock’ box is selected.
   - Click the Start button.

Important Note: if the program fails to abort after about 10 seconds, cancel the run by pressing the ‘stop’ button



Threading Error Analysis

Detect Deadlocks

Detect Deadlocks

Narrowest scope threading error analysis. Minimizes the load on the system and the time and resources required to perform analysis; however, detects the narrowest set of errors and provides minimal details. Press F1 for more details.

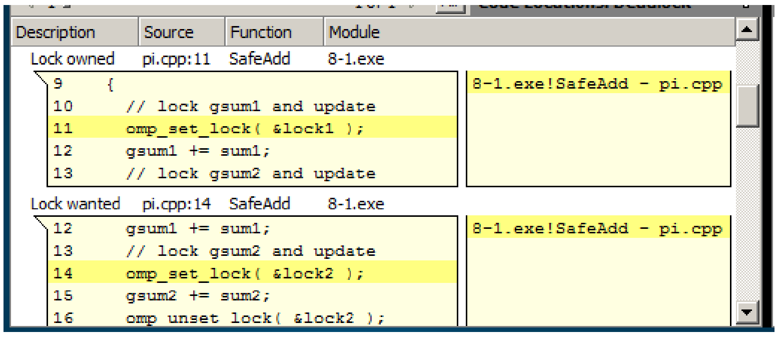
☑ Terminate on deadlock

Stack frame depth: 1

8. In your analysis you should see either a *lock-hierarchy violation* or a *deadlock*. Click on the problem with the right hand mouse button and select ‘explain problem’ from the menu.

Read the description to make sure you understand the nature of the problem.

9. Now double-click on one of he problem(s), you will find that Inspector XE highlights the use of a lock (see below).



Description	Source	Function	Module
Lock owned	pi.cpp:11	SafeAdd	8-1.exe

```
9       {
10      // lock gsum1 and update
11      omp_set_lock( &lock1 );
12      gsum1 += sum1;
13      // lock gsum2 and update
```

8-1.exe!SafeAdd - pi.cpp

Description	Source	Function	Module
Lock wanted	pi.cpp:14	SafeAdd	8-1.exe

```
12      gsum1 += sum1;
13      // lock gsum2 and update
14      omp_set_lock( &lock2 );
15      gsum2 += sum2;
16      omp_unset_lock( &lock2 );
```

8-1.exe!SafeAdd - pi.cpp

The underlying problem is a coding error in the call to the function `SafeAdd()`, where the programmer has accidently swapped the order that the locks are passed into the function.

9. Fix the deadlock issue by editing lines 48 and 61 of `pi.cpp` to look like this:

```
48: SafeAdd(sum1,sum2,lock1,lock2);
61: SafeAdd(sum1,sum2,lock1,lock2);
```

10. Rebuild the application (see step 3 and step 4) and then run the program. The program should run to completion without hanging.

11. Re-run the deadlock analysis (see step 7). No errors should be reported.

In this activity you used Detect Deadlocks analysis to find the deadlock. The program also has a data race, but it was not detected. In Activity 8-2, you use Inspector XE to detect and fix the data race.

ACTIVITY 8-2: DETECTING AND FIXING DATA RACES

In this activity you use Inspector XE to help detect and fix a data race. You can run this activity on Linux or Windows.

Building and Running the Program

1. Continue to work with the modified version of pi.cpp. This step assumes you have fixed the deadlock identified in Activity 8-1.

2. Build the program `8-2.exe` with the following command:

- WINDOWS

```
icl   /Od  /Qopenmp   /Zi  pi.cpp  -o  8-2.exe
```

- LINUX

```
icc    -O0  -openmp  -g  pi.cpp  -o  8-2.exe
```

3. Run the program several times to make sure it works:

4. Set the number of threads to be 1 using the OMP_NUM_THREADS environment variable, and then run the program several times and note its behavior:

- WINDOWS

```
set  OMP_NUM_THREADS=1
```

```
8-2.exe
```

- LINUX

```
export OMP_NUM_THREADS=1
./8-2.exe
```

Does the program run correctly? What about the value of Pi, is it correct?

5. Repeat the previous step, but this time increase value of the OMP_NUM_THREADS environment variable to be 2, then 4.

Detecting and Fixing the Data Race

6. Carry out a Detect Deadlocks and Data Races analysis in Inspector XE (in the Inspector project you created in Activity 8-1):

- ➢ Select File | New | Analysis.
- ➢ In the Project Properties dialog, Change the application name to 8-2.exe.
- ➢ Highlight the Detect Deadlocks and Data Races analysis.
- ➢ Click the Start button.

7. After the analysis is displayed, look at the problem(s) reported. Make sure you understand the issues.

8. Fix the data races by editing line 35 of pi.cpp to look like this:

```
#35:        #pragma omp parallel  sections private(x1,x2,sum1,sum2)
```

9. Repeat steps 2 to 5 to confirm that you have fixed the data races.

NB: Make sure you have deleted the environment variable OMP_NUM_THREADS before running the program

- WINDOWS

```
set OMP_NUM_THREADS=
8-2.exe
```

- LINUX

```
unset OMP_NUM_THREADS
./8-2.exe
```

Step 4 - Tuning a Parallel Program

Warning- Step 4 uses a different application from the previous steps!

ACTIVITY 9-1: DEFINING A BASELINE

In this activity you build the code from Listings 9-4 and 9-5 and use Amplifier XE to look at how parallel the resulting program is. You can run this activity on Linux or Windows.

Building and Running the Program

1. Copy the source code in Listing 9-4 into a file named ParallelPrime.cpp, and the source code in Listing 9-5 into a file named wtime.c 1 (they can also be found already extracted in “Chapter 9” folder).

2. Build the program with the following command:

- WINDOWS

```
icl /O2 /Zi /Qopenmp  /Ob1  ParallelPrime.cpp wtime.c -o 9-1.exe
```

- LINUX

```
icc -O2 -g -openmp -inline-level=1  ParallelPrime.cpp wtime.c -o
9-1.exe
```

3. Run the program and record the time taken:

```
9-1.exe
```

4. If the program does not run for about 2 to 3 seconds, edit the value in the `#define LAST` statement in `ParallelPrime.cpp` and rebuild and run the program until it runs for about 2 seconds.

```
#define LAST 300000
```

Using the Command-Line Version of Amplifier XE to Get a Timestamp

5. Start the command-line version of Amplifier XE, and record the elapsed time and average concurrency:

```
      amplxe-cl -collect concurrency ./9-1.exe
```

Make a note of the results directory (for example, r000cc). You will need this for Activity 9-2.

Does the concurrency look good? (Hint: compare the Average Concurrency value with the Logical CPU Count).

If your result is poor, is there any indication in the results as to what the problem might be?

ACTIVITY 9-2: IDENTIFYING THE CONCURRENCY HOTSPOTS

In this activity you use the GUI version of Amplifier XE to examine the results from Activity 9-1. You can run this activity on Linux or Windows.

1. Open the GUI version of Amplifier XE, using the results directory that you noted in step 5 of Activity 9-1:

```
amplxe-gui r000cc
```

2. Look at the Thread Concurrency Histogram and the CPU Usage Histogram in the Summary page (This page will have automatically been displayed in step 1 of this activity).

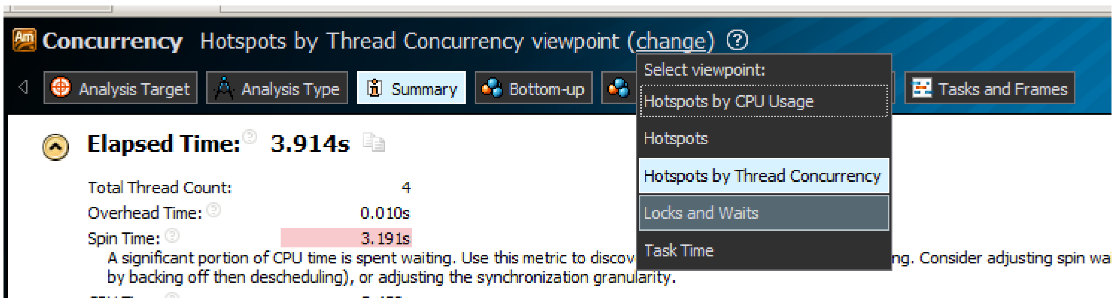
What value is the Target Concurrency?

What value is the Average Concurrency?

Record the names of the top three hotspots along with the CPU time used in the table below.

Hotspot	CPU Time

3. Change the viewpoint of your results to Locks and Waits by pressing (change) :



Elapsed Time: 3.914s

Total Thread Count:	4
Overhead Time:	0.010s
Spin Time:	3.191s

A significant portion of CPU time is spent waiting. Use this metric to discov … ng. Consider adjusting spin wa … by backing off then descheduling), or adjusting the synchronization granularity.

Record the names of the top three waiting objects along with the wait time and wait count in the table below.

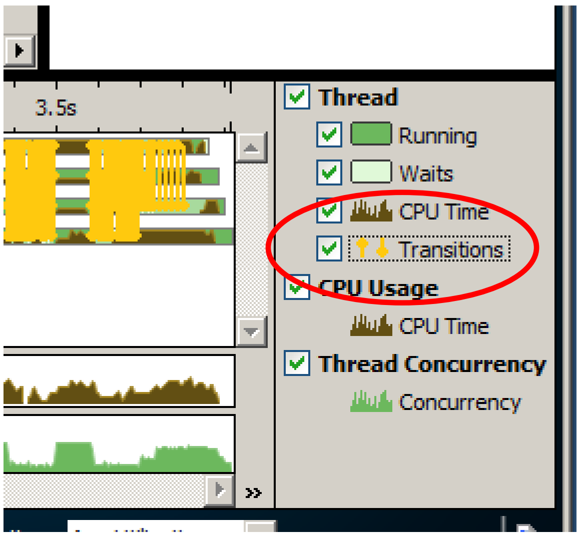
Sync Object	CPU Time	Wait count

The Program has two different problems (you don't know this yet – we are just telling you) – What you should do is fix the problem associated with the Locks and Waits.

ACTIVITY 9-3: ANALYZING THE TIMELINE

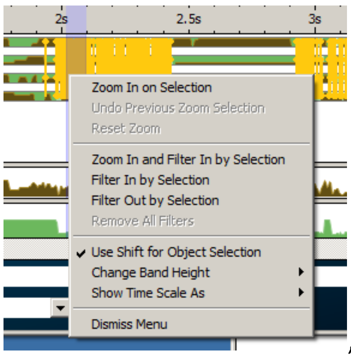
Steps 1-5 (Windows Only)

1. Display the Bottom-up page by clicking the Bottom-up button.
2. Make sure the transitions are displayed by checking the Transitions tick box:



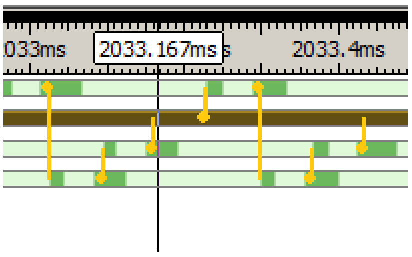
3. In the timeline pane you should see several regions that are densily populated with yellow transition lines.

Highlight on of these dense regions and press the right hand mouse button and select 'Zoom In and Filter by Selection':



,

Keep expanding the view until you can clearly see the individual transition lines. You should see that only one thread is ever running at any one time.



4. Hover your mouse over a transition line and read the information displayed.

5. Double-click the transition line, which should take you to the source lines.

If the transition is in a run time library there may be no source available! – in which case try to follow the stack up to the place in your code where the runtime library is called.

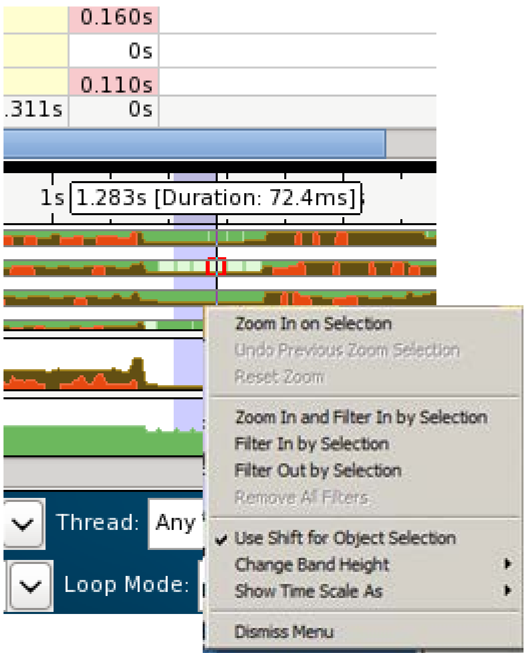
Windows users now Jump to Step 9

Steps 6-8 (Linux Only)

6. Display the Bottom-up page by clicking the Bottom-up button.

7. In the timeline pane you should see several regions that have no CPU activity

8. Highlight one of these dense regions and press the right hand mouse button and select 'Zoom In and Filter by Selection':



When you look in the hotspot, you should find that the hotspot is due to file IO.

Steps 9 onwards (Linux and Windows)

9. You should have found that the performance problem is caused to the call to `printf` in `PrintProgress()` – where print is being called too many times . If this is the case, then modify the code so it will look like this: (The additions are in large **bold**)

```
// Display progress
void PrintProgress(int Range )
{
  int Percent = 0;
  static int lastPercentile = 0;
  #pragma omp critical
  {
    gProgress++;
    Percent = (int)((float)gProgress/(float)Range *200.0f + 0.5f);
  }

  if( Percent % 10 == 0 )
  {
    // we should only call this if the value is new!
      if(lastPercentile < Percent / 10)
      {
         printf("%s%3d%%", CursorBack,Percent);
         lastPercentile++;
     }
  }
}
```

10. Re-build the modified program and re-run the concurrency analysis (same as activity 9-1). Did the program get better? Are there still any other problems?

Record the names of the top three hotspots along with the CPU time used in the table below.

Hotspot	CPU Time

Additional note for when using earlier versions of the Intel compiler …

When using earlier versions of the Intel compiler with this lab, the use of the critical section in section in PrintProgress() was a source of inefficiency:

```
#pragma omp critical
  {
    gProgress++;
    Percent = (int)((float)gProgress/(float)Range *200.0f + 0.5f);
  }
```

To correct this problem, the following change must be made:

In ParallelPrime.cpp, edit lines 16 to 20 so that they look the same as the following:

```
16:   #pragma omp atomic
17:   gProgress++;
18:
19:   Percent = (int)((float)gProgress/(float)Range *200.0f+ 0.5f);
20:
```

Lab Exercises – Optional extras

ACTIVITY 6-2: USING THE COMPILER'S LOOP PROFILER

In this activity you use the Intel compiler to instrument the Mandelbrot program and then find the busiest hotspots using the loopprofileviewer.

1. Make sure you have carried out Activity 6-1.
2. Rebuild the application, adding the /Zi option to generate debug information; the option /Ob0 (linux: -inline-level=0) to reduce inlining; and the /Qprofile options so that the compiler instruments the code:

- **Windows**

```
icl  /Zi /O2  /Qipo wtime.c main.cpp   mandelbrot.cpp   \
ppm.cpp -o  6-2.exe  /Ob0 /Qprofile-functions      \
/Qprofile-loops:all  /Qprofile-loops-report:2
```

- **Linux**

```
icc  -g -O2  -ipo wtime.c main.cpp   mandelbrot.cpp \
ppm.cpp -o 6-2.exe -inline-level=0 -profile-functions\
-profile-loops:all  -profile-loops report:2
```

3. Run the program you have just created and record the time taken:

   6-2.exe

4. Start the loopprofileviewer from the command line, and browse to the XML file that has just been generated.

   Which is the biggest function hotspot?

   What line has the 'hottest' loop?

ACTIVITY 6-5: Further optimisation work

Try to parallelize other loops in the Mandelbrot application, using the techniques you have learned in this lab to debug and tune the application.