



Multi-core Programming **—Student Workbook**

Yang Quansheng (杨全胜)

Base on Intel's manual

School of Computer Science & Engineering
Southeast University

Lab. 1: Intel® Compiler Switches

In this activity, you will set up environment and compile with both Microsoft* Visual C++.NET (MSVC*) and Intel® C++ Compiler (icl).

Setting up

1. Open an Intel compiler command prompt window (Start -> All Programs -> Intel(R) Software Development Tools -> Intel(R) C++ Compiler 10.0 -> Build Environment for IA-32 Applications).
2. Using Windows Explorer, navigate to the Compiler Switches folder (this is in the directory where you initially copied the class files) and unzip the file RayTrace2.zip, if it has not already been unzipped.

Compiling with MSVC*

1. Using the Intel compiler command prompt, change to the Raytrace2 directory:

```
> cd source \raytrace2
```

2. Make the files and clean up:

```
> nmake /f raytrace2.mak clean
```

3. Complete the process:

```
> nmake /f raytrace2.mak CPP=c1.exe
```

Rendering the Image

1. Render the image by executing the following script:

```
> raytrace2 320 240
```

```
> Press 'g' to begin the render
```

```
> Press 'q' to quit the application
```

2. Record time elapsed _____

Compiling with Intel C++ Compiler

1. Make the files and clean up:

```
> nmake /f raytrace2.mak clean
```

2. Complete the process:

```
> nmake /f raytrace2.mak
```

3. Render the image by executing the following script:

```
> raytrace2 320 240
```

```
> Press 'g' to begin the render
```

```
> Press 'q' to quit the application
```

4. Record time elapsed _____

Using High Level Optimizer (-O3)

1. Compile for high-level optimizations (HLO):

```
> nmake /f raytrace2.mak clean
```

```
> nmake /f raytrace2.mak CF="-O3"
```

2. Render the image.

3. Record time elapsed _____

Using Inter-procedural Optimization (-Qipo)

1. Compile for inter-procedural optimizations:

```
> nmake /f raytrace2.mak clean
> nmake /f raytrace2.mak CF="-Qipo" LF="-Qipo"
```

2. Render the image.

3. Record time elapsed _____

Using Profile-guided Optimization (-Qprof_gen, -Qprof_use)

1. Compile to create PGO instrumented binary:

```
> nmake /f raytrace2.mak clean
> nmake /f raytrace2.mak CF="-Qprof_gen -Qprof_dir ..\RayTrace2"
```

2. Render the image.

Note: Time period reported for the instrumented executable would be of considerable length.

3. Record time elapsed _____

4. Compile to 'use' PGO information:

```
> nmake /f raytrace2.mak clean
> nmake /f raytrace2.mak CF="-Qprof_use Qprof_dir ..\RayTrace2"
```

Note: Ignore messages stating "no .dpi information."

5. Render the original image (from Step 2).

6. Record time elapsed _____

Using Vectorization (-QxP)

1. Compile for vectorization:

```
> nmake /f raytrace2.mak clean
> nmake /f raytrace2.mak CF="-QxP"
```

2. Render the image.

3. Record time elapsed _____

Using All Optimization Options (-O3, -QxP, IPO and PGO)

1. Compile using the following options: -O3, -QxP, IPO, and PGO:

```
> nmake /f raytrace2.mak clean
> nmake /f raytrace2.mak CF="-O3 -QxP -Qipo -Qprof_use - Qprof_dir .. \RayTrace2"
    LF="-Qipo"
```

Note: You need not collect additional profile information; use the existing profile from "Using Profile-guided Optimization".

2. Render the image.

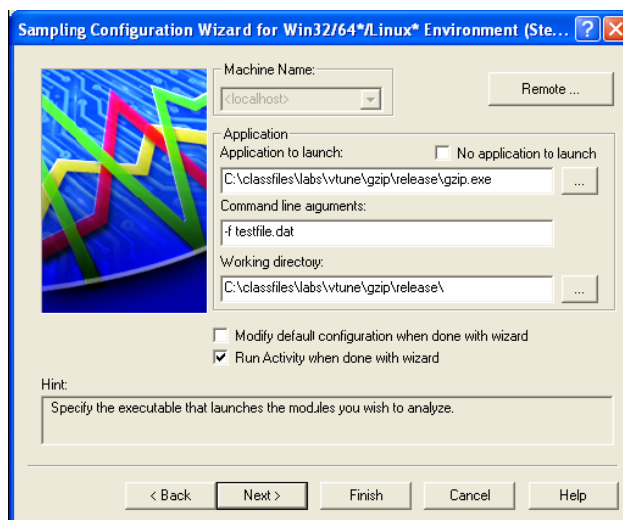
3. Record time elapsed _____

Lab. 2: Basics of Intel® VTune™ Performance Analyzer

Collect Sampling Data for the Clockticks Event

1. Make sure that the virus scanner is disabled.
2. Double click the **VTune Performance Analyzer** icon on the desktop.
3. Click **New Project**.
4. Click **Sampling Wizard**.
5. Click **OK**.
6. Select **Window*/Windows* CE/Linux Profiling**.
7. Uncheck **Automatically generate tuning advice**.
8. Click **Next**.
9. In the **Application To Launch** text box, type:
`D:\classfiles\VTuneBasics\gzip\release\gzip.exe`
10. In the **Command Line Arguments** text box, type: `-f testfile.dat`
11. Click **Finish**. The VTune analyzer will now launch and profile gzip.

Figure Sampling Configuration Wizard






Questions

- What function in gzip.exe takes the most time?*
- Which function in gzip.exe has the highest CPI?*
- Which line of source in gzip.exe has the most clocktick samples?*
- Is gzip.exe multithreaded?*

Create a Sampling Activity

1. Create a new Activity by clicking on **Activity->New Activity**.
2. Click **Sampling Wizard**.
3. Click **OK**.
4. Select **Windows*/Windows* CE/Linux Profiling**.
5. Uncheck **Automatically generate tuning advice**.

6. Click **Next**.
7. In the **Application To Launch** text box, type:
`D:\classfiles\VTuneBasics\matrix_mt\release\matrix.exe`
8. Click **Finish**.
9. After the VTune analyzer finishes collecting data, click the **Process** button.
10. Click **CTRL+A** to select all processes.
11. Click the **Display Over Time View** button , shown at left.
12. You can now see when each of the different processes was running. You can do this for the Process, Thread, and Module views.
13. To zoom in on a particular time region, select the time region by clicking and dragging over it. Then click the **Zoom In** button , shown at left.
14. To see the regular sampling view for that time region, click the **Regular Sampling View** button , shown at left. You will now be able to drill down into your code for that time region.

Collect the call graph data

1. Create a new Activity by clicking on **Activity->New Activity**.
2. Double click **Call Graph Wizard**.
3. Click **Windows*/Linux* Profiling**.
4. Click **Next**.
5. Type: `D:\classfiles\VTuneBasics\gzip\release\gzip.exe` in the **Application to launch** text box.
6. Type `-f testfile.dat` in the Command Line Arguments text box.
7. Click **Finish**.

What function has the most time spent in it and what functions are calling it?

Collect Samples Based on the Clockticks Event using the

Command Line Interface

1. Open the command prompt by clicking Start->Programs->Accessories->Command Prompt.
2. Go to the following directory by typing: `D:\classfiles\VTuneBasics\gzip\release\`
3. Create a new Activity that will launch gzip and collect samples based on the Clockticks event by typing: `vtl activity gzip -c sampling -app gzip.exe, "-f testfile.dat"`
4. Run the last Activity created by typing `vtl run`.
(Alternatively, you can append the word "run" to the end of the command in Step 3 to run the Activity immediately after it is created).

View the Profiling Data for gzip

1. Type: `vtl view -modules` to see the number of samples for each module system-wide.
2. Type: `vtl view -hf -mn gzip.exe` to see the function level breakdown of the samples in gzip.exe.

Pack the Data and View It in the GUI

1. Type `vtl pack gzip`.

This creates a file called `gzip.vxp`. This file can be transported from computer to computer and also opened in the VTune analyzer's GUI.

2. Start **VTune Performance Analyzer** (For example, by double-clicking the desktop icon, if present.)

3. Click **Browse for Existing File**.

4. Open: `D:\classfiles\VTuneBasics\gzip\release\gzip.vxp`

5. Click **OK**.

6. Click **OK**.

The GUI displays the performance data.

Lab. 3: Intel® Math Kernel Library

Matrix Multiply Sample

This activity demonstrates performance characteristics of BLAS levels 1, 2 and 3, compared to C source code. In this activity you will inspect, build and run a matrix multiply sample using source code, DDOT, DGEMV and DGEMM.

1. Navigate to the folder **MKL_Overview\DGEMM**, and open the file **mkl_lab_solution.c** using any editor of your choice. Go through the code quickly to confirm the 4 implementations of matrix multiply.

2. Examine the **Makefiles** supplied, and identify the key link steps to enable using MKL.

If necessary, edit the file so that all include or library paths are correct. Use these Makefiles to build the demo. Run the programs and record the timings.

3. Note differences in timings among the different implementations:

rool_your_own:_____
using DDOT:_____
using DGEMV:_____
using DGEMM:_____

4. **MKL** functions assume a default threads value of 1. Change this value by setting the environment variable `OMP_NUM_THREADS`, for example:

```
set OMP_NUM_THREADS=2
```

5. Observe the performance at different numbers of threads.

What happens then that the number exceeds the physical processors?

Monte Carlo Calculation of Pi

In this activity, you will modify the Monte Carlo computation of pi to use the random number feature of the Vector Statistical Library (VSL). You will also make use of the multithreading capabilities of VSL.

1. Navigate to the folder **MKL_Overview\MonteCarloPi**, and open the file **pimonte.c** using any

editor of your choice. Go through the code quickly to understand how the `rand()` function is implemented.

Thought question: Could this loop be threaded?

2. Examine the file **pimonte_VSL.c**, and understand the changes necessary to implement a library call to replace `rand()`.

Thought questions: Why is this not a 1:1 substitution for `rand()`?

What is the purpose of, and sensitivity to, `blocksize`?

What are the parameters `BRNG` and `VSL_BRNG_MCG31`?

Are they the best choices for this computation?

Could this implementation be threaded?

3. Examine the **Makefiles** supplied, and identify the key differences. Use these Makefiles to build versions of this application with `rand()` and with VSL (recall the syntax "make -f"). Note the impact of the "-xP" switch in the compiler report, in the two versions. Note also the difference in results for pi, and for the run times of each image.
4. As in the previous exercise, change the number of threads used and observe the changes in both performance and in the value of p.

Lab. 4: Programming with Windows* Threads

Build & Run HelloThreads Program

1. Close Microsoft Visual Studio, if it is started.
2. With Windows Explorer*, open the folder **D:\classfiles\Win32 Threads\HelloThreads**.
3. With Microsoft Visual Studio, open the file **HelloThreads.sln** by double-clicking it.
4. From the **Build** menu, select **Configuration Manager** and then select **Debug** build.
5. From the **Project** menu, click **Properties** and then click the **C/C++** folder.
6. Make sure that **Debug** options are selected, as shown in [Figure 4.1](#).
7. Make sure that **Optimization** is disabled, as shown in [Figure 4.2](#).
8. Make sure that thread-safe libraries are selected, as shown in [Figure 4.3](#).
9. Make sure that **Debug** symbols are preserved during the link phase, as shown in [Figure 4.4](#).
10. From the **Build** menu, select **Build Solution** to build your project.
11. From the **Debug** menu, select **Start Without Debugging** to run the program.
12. In Microsoft Visual Studio's Solution Explorer, expand the **HelloThreads** project and select the file **main.cpp** to open it, as shown in [Figure 4.5](#).
13. Modify the thread function to report the thread creation sequence (that is, "Hello Thread 0", "Hello Thread 1", "Hello Thread 2", and so on).

Hint: Use the `CreateThread()` loop variable to give each thread a unique number.

14. Build and execute your program.

In what order do the threads execute?

Do the results look correct?

Why or why not?

Review Questions

The execution order of threads is unpredictable.

True False

What build options are required for *any* threaded software development?

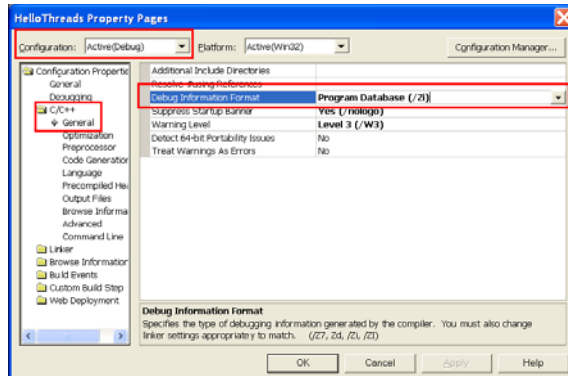


Figure 4.1. Project Setting – C/C++ Folder – Debug Options

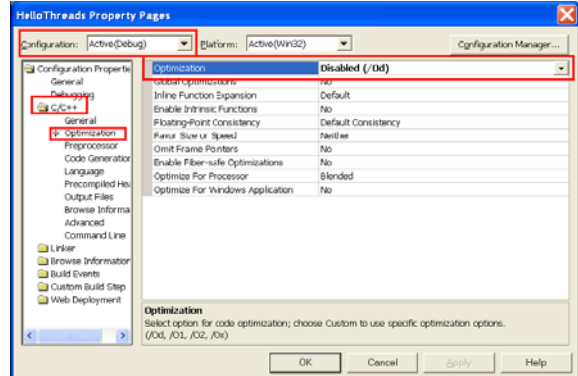


Figure 4.2. Project Settings – C/C++ Folder – Optimization Options

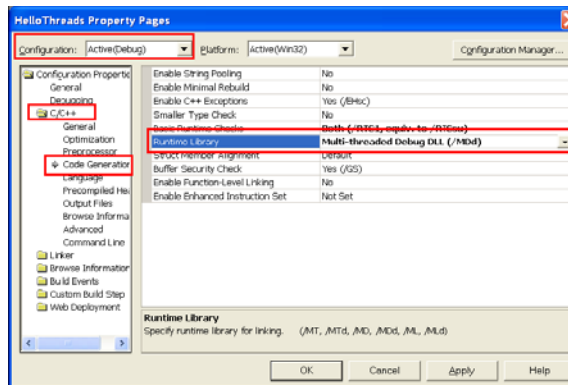


Figure 4.3. Project Settings – C/C++ Folder – Thread-safe Libraries Options

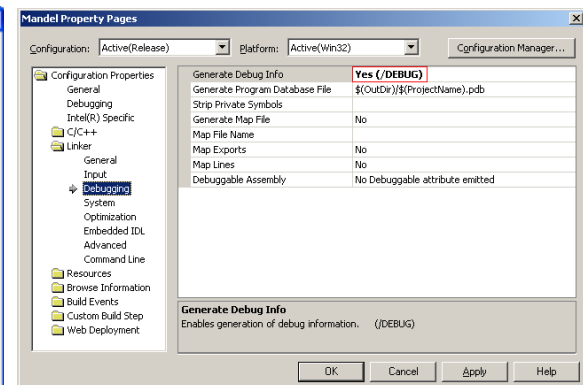


Figure 4.4. Linker Settings – Debugging Folder

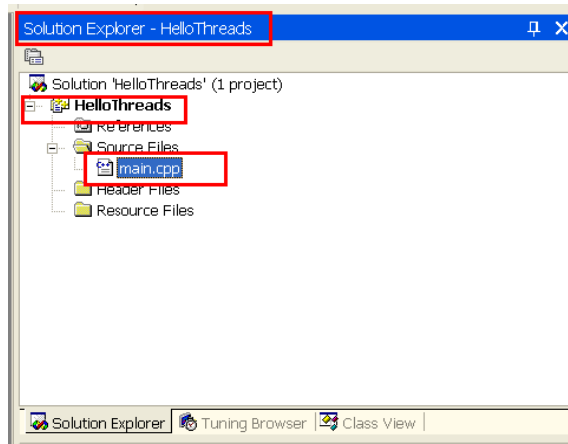


Figure 4.5. Solution Explorer – HelloThreads Project

Approximating Pi with Numerical Integration

Build and Run the Serial Program

1. With Windows Explorer, open the folder C:\classfiles\Win32 Threads\Pi\.

2. Start **Pi.sln** by double-clicking it.
3. From the **Build** menu, select **Set Active Configuration** and then select **Debug** build.
4. From the **Project** menu, click **Properties** and click the **C/C++** folder. From the **Build** menu, select **Build Solution** to build your project.
5. From the **Debug** menu, select **Start Without Debugging** to run the program.

Is the value of the Pi (3.1415...) printed correct?

Why or why not?

Correct Errors and Validate Results

1. In Microsoft Visual Studio's Solution Explorer, expand the **Pi** project and open the file, **Pi.cpp**.
2. On the **C/C++** folder, make sure that thread-safe libraries are selected, as shown in [Figure 4.6](#).

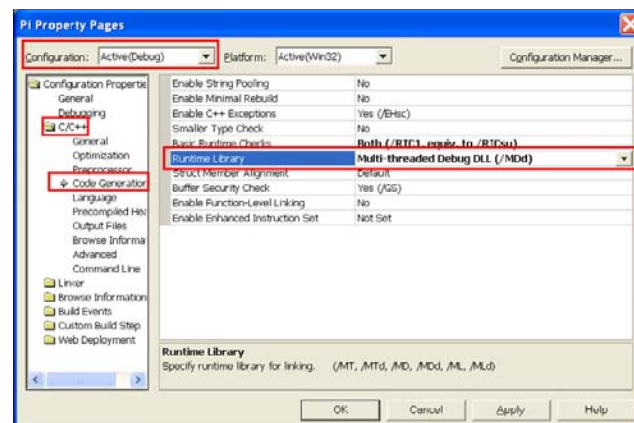


Figure 4.6. Project Settings – C/C++ Folder – Thread-safe Libraries Options

3. Thread the serial code to compute Pi using four threads. The bulk of the computation being done is located in the body of the loop. Encapsulate the loop computations into a function and devise a method to ensure that the iterations are divided amongst the threads such that each iteration is computed by only one thread.
4. Use a **CRITICAL_SECTION** to protect shared resources accessed by more than one thread. Locate any data races in your program and correct those errors. Some logic changes from the serial version might be required to create code that is both correct and safe.

Challenge: Minimize the number of lines of code in the critical section(s).

Hint: Think about local variables.

5. When you have changed the source code, from the **Build** menu, select **Build Solution** to rebuild and then from the **Debug** menu, select **Start Without Debugging** to execute.
6. Keep correcting your source code until you see the correct value of Pi being printed.

The correct value of Pi is 3.14159.

Hint: A complete solution to the lab is provided in the file **PiSolution.cpp**, which is located in the following directory: **D:\classfiles\Multi-Core\Windows\Win32 Threads\PI**

Review Questions

All threads should use the same **CRITICAL_SECTION**.

True False

Threading errors in software can always be corrected by using only synchronization objects.

True False

CRITICAL_SECTION objects should always be declared as global variables.

True False

What build options are required for *any* threaded software development?

Lab. 5: Threading with OpenMP for Windows

Hello Worlds

Initial Compile

1. Within a "Build Environment for IA-32 applications" command window, move to the HelloWorlds directory:

```
cd Open MP/HelloWorlds
```

2. Compile serial code using the Intel compiler:

```
icl HelloWorlds.c
```

3. Run the program:

```
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:

```
printf("GoodBye World\n");  
#pragma omp parallel  
{  
...[Code to run in Parallel goes here ] ...  
}
```

2. Compile in "Serial Mode" using the Intel compiler:

```
icl HelloWorlds.c
```

3. Notice the pragma warning statements.

4. Fix any syntax errors.

5. Run the program:

```
HelloWorlds.exe
```

OpenMP Compile

1. Compile in OpenMP mode using the Intel compiler:

```
icl /Qopenmp HelloWorlds.c
```

2. Enable the environment for multiple OpenMP threads:

```
Set OMP_NUM_THREADS=2
```

3. Run the program in a multithreaded environment:

```
HelloWorlds.exe
```

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?

Computing Pi with Numerical Integration

1. Within Windows Explorer, move to the Pi directory (classfiles\OpenMP\Pi).
2. Double-click on the Microsoft* Visual Studio* Solution icon (pi.sln) to launch Visual Studio. Build the pi application.
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.

```
# pragmaomp 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): _____

Monte Carlo Pi

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.

Lab. 6: Correcting Threading Errors with Intel® Thread Checker for Explicit Threads

Find Prospective Data Races

The application computes the potential energy of a system of particles based on the distance, in three dimensions, of each pairwise set of particles. The code is small enough that you may be able to identify the potential data races and storage conflicts by visual inspection. If you identify and make a list of problems, check your list with the list that Thread Checker identifies.

Build and Run Potential Serial Program

1. With Windows Explorer*, open the folder **D:\classfiles\Thread Checker\potential_serial**
2. With Microsoft Visual Studio, open the file **potential_serial.sln** by double-clicking it.
3. From the **Build** menu, select **Configuration Manager** and then select **Debug** build.
4. From the **Build** menu, select **Build Solution** (or use **Ctrl+Shift+B**) and compile the executable.
5. Run the executable (**Debug** menu -> **Start Without Debugging**, or **Ctrl+F5**).

Build and Run Potential Threaded Program

1. With Windows Explorer*, open the folder **D:\classfiles\Thread Checker\potential_win**
2. With Microsoft Visual Studio, open the file **potential_win.sln** by double-clicking it.
3. From the **Build** menu, select **Configuration Manager** and then select **Debug** build.
4. Ensure that the following compile and link flags are set correctly:
 - a. Debug options are set (**/Zi**)
 - b. Debug symbols are preserved during linking (**/DEBUG**)
 - c. Optimization is disabled (**/Od**)
 - d. Thread safe system libraries are used (**/MDd**)
 - e. The binary is re-locatable (**/fixed:no**)
5. From the **Build** menu, select **Build Solution** to compile the executable.

6. From the **Debug** menu, select **Start Without Debugging** to run the program.

Note: The number of particles and iterations has been reduced from the serial version of the code in order to facilitate Thread Checker runs.

7. With Windows Explorer*, start the Intel VTune Performance Analyzer.

8. Start a **New Project** by clicking on the icon.

9. From the **Category Threading Wizards**, select the **Intel® Thread Checker Wizard**.

10. Set up the **potential_win.exe** application to be run by using the browse “...” button and then click **Finish** to start Thread Checker.

11. Scan through some of the diagnostics displayed. Check the source code lines from some of the diagnostics.

Can you see why there is a conflict on those lines of code?

[Resolve Data Races](#)

Resolve the Problems

1. Fix the problems identified by Thread Checker in the earlier lab. Which variables can be left to be shared between threads? Which variables can be made local to each thread? Which variables must be protected with some form of synchronization?

2. Be sure to re-run the corrected application through Thread Checker until you have no more diagnostics being generated from the code.

3. Once you have eliminated all the threading problems, re-set the number of particles and iterations that were used in the serial version. Re-build and run the code.

Do the answers from the threaded code match up with the output from the serial application?

[Identifying Deadlock](#)

Build and Run the Program

1. With Windows Explorer, open the folder **D:\classfiles\Thread Checker\Deadlock**.

2. Open **Deadlock.sln** by double-clicking it.

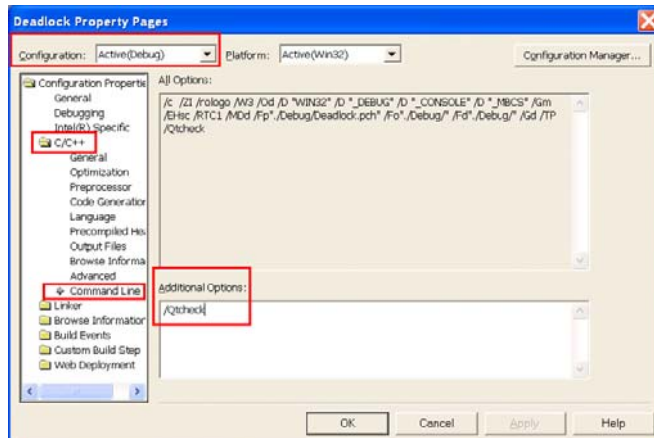
3. From the **Build** menu, select **Configuration Manager** and then select the **Debug** build.

4. Using the Debug configuration project, ensure that the following compile and link flags are set properly.

- a. Debug options are set (**/Zi**)
- b. Debug symbols are preserved during linking (**/DEBUG**)
- c. Optimization is disabled (**/Od**)
- d. Thread safe system libraries are used (**/MDd**)
- e. The binary is re-locatable (**/fixed:no**)

5. On the **C/C++** menu, select the **Command Line** folder and add **/Qtcheck** by editing the **Additional Options**, as shown in [Figure 7.1](#).

[Figure 7.1. Project Settings – C/C++ Folder – Compiler Options](#)



6. From the **Build** menu, select **Build Solution** to build your project.

Is this application using:

Source instrumentation? _____

Binary instrumentation? _____

How can you tell which instrumentation type is used?

7. From the **Debug** menu, select **Start Without Debugging** to run the program.

Do the results look correct?

Why or why not?

Run Application within Intel® Thread Checker

1. Start VTune Performance Analyzer and select **New Project**.

2. From the **Category Threading Wizards**, select the **Intel® Thread Checker Wizard**.

3. Set up the application to be run by using the browse “...” button and then click **Finish** to start Thread Checker.

You should see either one of the two displays shown in [Figure 7.2](#) and [Figure 7.3](#), because deadlock does not occur during every run.

[Figure 6.2](#) shows the diagnostics list when the deadlock does not occur during the run. Intel Thread Checker still catches it as a potential deadlock and it is listed with YELLOW bullets (circled in red), which indicates CAUTION.

Figure 6.2. Diagnostic List When Deadlock Does Not Occur During a Run

Context[Best]	ID	Short Description	Severity	Description	Count	Filtered
"Deadlock.cpp":1	1	Read -> Write data-race		Memory write at "Deadlock.cpp":43 conflicts with a prior memory read at "Deadlock.cpp":67 [anti dependence]	1	False
"Deadlock.cpp":41	2	Read -> Write data-race		Memory write at "Deadlock.cpp":45 conflicts with a prior memory read at "Deadlock.cpp":67 [anti dependence]	1	False
Whole Program 1	3	Thread termination		Thread Info at "Deadlock.cpp":64 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 2	4	A sync object was acquired in the wrong order		A synchronization object "Deadlock.cpp":44 was acquired in the wrong order at "Deadlock.cpp":42	1	False
Whole Program 2	5	A sync object was acquired in the wrong order		A synchronization object "Deadlock.cpp":31 was acquired in the wrong order at "Deadlock.cpp":29	1	False
Whole Program 3	6	Thread termination		Thread Info at "Deadlock.cpp":63 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 4	7	Thread termination		Thread Info at "Deadlock.cpp":54 - includes stack allocation of 1048576 and use of 4096 bytes	1	False

[Figure 6.3](#) shows the error list when the deadlock occurs during a run. When this occurs, the user **MUST** kill the application by closing the DOS window opened by the application.4. Double-click on one or more of the deadlock diagnostics (error or warning) in the **Diagnostics** window. Explore what lines Thread Checker points out as being involved in the deadlock diagnostics.

Figure 6.3. Diagnostic List When Deadlock Occurs During a Run

Context[Best]	ID	Short Description	Severity	Description	Count	Filtered
"Deadlock.cpp":27	1	Read -> Write data-race		Memory write at "Deadlock.cpp":29 conflicts with a prior memory read at "Deadlock.cpp":67 (anti dependence)	1	False
"Deadlock.cpp":34	2	Read -> Write data-race		Memory write at "Deadlock.cpp":42 conflicts with a prior memory read at "Deadlock.cpp":29 (anti dependence)	1	False
"Deadlock.cpp":34	3	Write -> Read data-race		Memory read at "Deadlock.cpp":42 conflicts with a prior memory write at "Deadlock.cpp":29 (flow dependence)	1	False
"Deadlock.cpp":34	4	Write -> Write data-race		Memory write at "Deadlock.cpp":42 conflicts with a prior memory write at "Deadlock.cpp":29 (output dependence)	1	False
"Deadlock.cpp":	9	The application was forcefully...		The application was forcefully terminated	1	False
Whole Program 1	5	A Thread is deadlocked		A Thread at "Deadlock.cpp":68 is deadlocked trying to acquire a resource owned by a thread at "Deadlock.cpp":64	1	False
Whole Program 2	6	A Thread is deadlocked		A Thread at "Deadlock.cpp":68 is deadlocked trying to acquire a resource owned by a thread at "Deadlock.cpp":63	1	False
Whole Program 3	7	A Thread is deadlocked		A Thread at "Deadlock.cpp":43 is deadlocked trying to acquire a resource owned by a thread at "Deadlock.cpp":28	1	False
Whole Program 4	8	A Thread is deadlocked		A Thread at "Deadlock.cpp":30 is deadlocked trying to acquire a resource owned by a thread at "Deadlock.cpp":41	1	False
Whole Program 5	10	Thread termination		Thread Info at "Deadlock.cpp":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 6	11	Thread termination		Thread Info at "Deadlock.cpp":63 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 7	12	Thread termination		Thread Info at "Deadlock.cpp":64 - includes stack allocation of 1048576 and use of 4096 bytes	1	False

Correct Errors and Validate Results

1. Return to Microsoft Visual Studio to correct the program.
 2. From Microsoft Visual Studio, edit the program and re-build it.
- Hint:** Notice the order that the `CRITICAL_SECTION` objects `cs0` and `cs1` are used in each thread function `work0()` and `work1()`.
3. When you have changed the source code, from select **Build Solution** the **Build** menu to rebuild.
 4. In the VTune analyzer environment, click the **Activity** menu and select **Run** to run Intel Thread Checker again to validate that you have corrected all threading errors.

You can re-use the same VTune analyzer environment project; in fact, have the VTune analyzer environment started and running while you change the source code in Microsoft Visual Studio. Once you have no threading errors, the display will contain only informational messages.

Hint: Minor logic changes will be required to **correct all** threading errors.

A complete solution to the lab is provided in the file, **DeadlockSolution.cpp**.

Review Questions

1. What build options are required for *any* threaded software development?
2. What build options are required for binary instrumentation?
3. What build options are required for source instrumentation?
4. Under what category is the Intel Thread Checker found in the VTune analyzer environment?
5. Using a larger data set (workload) causes Intel Thread Checker to find more information (errors, warnings, and so on). True False
6. Threading errors in software can always be corrected by using only synchronization objects. True False
7. If a deadlock is present in an application, it will always occur at run time. True False

[Testing Libraries for Thread Safety](#)

Setting-up and Compilation for Thread Safety Testing

1. With Windows Explorer, open the folder `D:\classfiles\Thread Checker\Thread Safe Libraries\`.
2. Double click the **Thread Safe Libraries.sln** Microsoft Visual Studio solution file to open the projects for this lab. You will find two projects in this solution:
 - a. **Library** – this project will build a DLL containing some functions to be tested for thread safety. This DLL can be thought of as either a user library or a third-party library.
 - b. **Library Tester** – this project is used to call the library functions and will be modified to test the library routines for thread safety.
3. Using the Debug configuration, build and run the Library Tester project to better understand the library routines. (This will also build the Library DLL.)

Note: Since we will be using OpenMP to generate threaded calls to the library routines involved and Intel Thread Checker to determine if there are any data conflicts or other problems, you must first convert the Library Tester project to use the Intel compiler.

4. Right-click on the Library Tester Project and select “Convert to use Intel[R] C++ Project System.” Click “Yes” in the Confirmation dialog box.
5. Add some OpenMP *parallel sections* code to run each of the 6 pairwise combinations of the three library routines on threads.

For example, to test the thread safety of a library routine, say `foo()`, with itself, use the following code:

```
# pragma omp parallel sections
{
# pragma omp section
    foo (x ) ;
# pragma omp section
    foo (y ) ;
}
```

6. Before building the threaded library testing application, be sure to set the compilation to use OpenMP. Within the **C/C++** configuration properties folder of the **Language** category, you will see a sub-pane labeled “Intel Specific”. Under this, change the “**Process OpenMP Directives**” item to “**Generate Parallel Code (/Qopenmp)**.”
7. Rebuild the application.
8. After launching VTune, create a new project and define a new Thread Checker activity for the **library_tester.exe** application. Run the Thread Checker activity. Were any diagnostics found?

Modifying Binary Instrumentation Levels

Even though the library DLL was compiled with debug symbols and information, Thread Checker may not find any diagnostics - even if there are data conflict errors within the library. This is due to the level of binary instrumentation used by Thread Checker.

The default binary instrumentation for user DLLs is “All Functions,” which will instrument each instruction of those portions that have debug information. However, Thread Checker is unable to locate the debug information data base file and lowers the instrumentation level to “API Imports”. This reduced level will not instrument user code, but will instrument selected system API functions.

Thus, to ensure that the library functions are thread safe, we must manually raise the instrumentation level of the DLL and re-run the analysis.

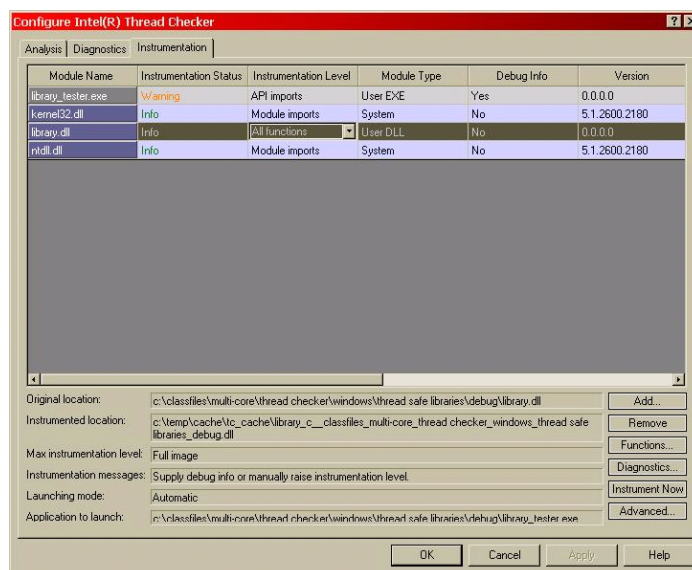
1. In the Thread Checker “Tuning Browser” pane, right click on the activity and chose the “Modify Collectors” option.
2. In the dialog box that pops up, select the “Modify the selected Activity” radio button and click OK.
3. Click on the Instrumentation tab in the “Configure Intel Thread Checker” box.
4. Left-click on the “Instrumentation Level” entry for the “library.dll” row. Select “All Functions” from the pull-down menu. In the lower right corner, click on the “Instrument Now” button to perform the binary instrumentation of this DLL at the chosen level.
5. Click OK.
6. Re-run the Thread Checker analysis.

Were any diagnostics found this time?

Review Questions

1. Which combinations of functions are not thread safe?
2. If this were a third-party library, rather than a set of function to which you have source code access, can you determine which functions were not thread safe in order to report this problem to the library developers?

Figure 6.4. Setting binary instrumentation levels for DLLs and other modules.






Lab. 7: Tuning Threaded Code with Intel® Thread Profiler for Explicit Threads

Getting Started

The application computes the potential energy of a system of particles based on the distance, in

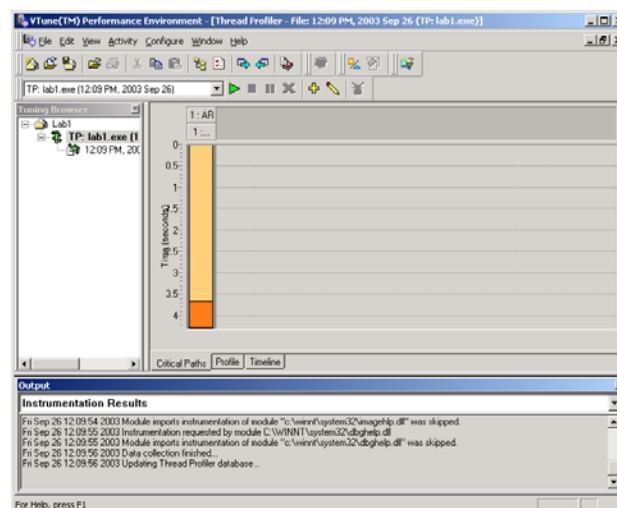
three dimensions, of each pairwise set of particles. This lab is designed to give you an introduction to running an application through the Thread Profiler and seeing what views are available within the tool. You will also see what those views reveal about the execution of threads within an application.

1. Find the Microsoft Visual Studio solution file **D:\classfiles\ThreadProfiler\potential lab 1\Potential Lab 1.sln**.
2. Double-click the icon to open Microsoft Visual Studio and build the application.
3. Launch VTune Performance Analyzer and create a new Thread Profiler activity that will run the executable generated from the previous step.
4. Run the application from within the Thread Profiler.
5. On completion of the run, you should see the Critical Path view as shown in [Figure 7.1](#).
6. Double click on the critical path bar to go to the profile view. The “Concurrency Level”  view should be the default. If this is not the case, click on the “Concurrency Level” button in the toolbar.
7. Click on the other grouping buttons (“Threads View”  and “Objects View” ). Click on the bars and roll over portions of the windows to see what pop windows appear and what data is contained within them.
8. Click on the Timeline Tab to see the “Timeline View” of the performance data. Explore the widow displayed and click on parts of the display to see what information might be available from this view.


Questions

From what you've seen, how would you characterize the performance of this application?
 Are there any obvious performance issues?

Figure 7.1. Thread Profiler Critical Path View Screen





Analyzing an Application

1. Return to the Profile tab and bring up the “Concurrency Level” view .

Approximately, what percentage of time was spent in serial (one thread only) and under-subscribed execution on the platform? -----

What percentage of time (approximately) was spent in full parallel execution? -----

What is this view telling you? -----

2. Select the “Threads View” by clicking on that button .
This view shows all the threads in the applications that were active in this critical path.
How many total threads were used during the execution of this application?
Can you tell if there is some performance problem inherent in the application?
Yes No
If so, what problem is it? If not, why not?
3. Select “Objects View”  by clicking on that button.
Are there any synchronization objects that account for a significant portion of the critical path time spent in serial or under-subscribed impact time? If so, which one(s)?
4. Select the “Timeline” tab to bring up the Timeline View.
What is the most striking feature that you notice from this data? Can you tell if there is some performance problem inherent in the application from this view?
Yes No
If so, what would you suggest be done to correct this problem?

Finding Load Balance Issues

The application used for this lab is a version of the potential energy physics simulation that employs a pool of threads, rather than creating and terminating a new set of threads for each time-step. Threads are controlled by a pair of events that signal threads to start and also signal the main thread when threaded execution has completed. Even though the loop iterations over the particles has been divided equally between threads, there is still a load balance issue with this code.

Build and Run Potential Threaded Program

1. With Windows Explorer*, open the folder **D:\classfiles\ThreadProfiler\Potential Lab 2**.
2. Double-click the **Potential Lab 2.sln** icon to start Microsoft Visual Studio.
3. Notice in the source code, the definition and initialization of the two events (**bSignal** and **eSignal**). Also, notice the use of the *done* variable within the **tPoolComputePot** function. This controls the termination of the threads when the simulation has completed.
4. Select the **Release** build, ensure the debug and linker options are set as needed, and build the application.
5. Launch VTune Performance Analyzer and create a new Thread Profiler activity that will run the executable generated from the previous step.
6. Run the application from within the Thread Profiler.

Evaluate Thread Profiler Results to Diagnose Problem

1. Look at the Critical Path view, and note that a much larger percentage of time is spent in “Serial impact time.” The Concurrency Level profile view confirms that the majority of execution time is spent with only one thread active. This data means that a single thread is running and actively keeping other threads from executing, typically through some synchronization object.
2. Bring up the Objects profile view.
Can you tell if there is one object that has been involved with the Serial Impact Time? If so, which one?

Is this object protecting access to data?

If so, can the application and data access patterns be modified to reduce the amount of time spent by threads “holding” this synchronization object?

Note: Since the synchronization object (eSignal) is used to signal when the threaded portion of the computations are completed, and not to protect data, the main thread is the only one “impacted.” This impact is necessary to ensure the correct execution of the application. Thus, modifications to the data will not affect this situation.

3. If the problem is not with data-protecting synchronization, the cause may be within the threads themselves. Bring up the Threads profile view. Can you tell if one or more threads were involved in the “Serial impact time” from the critical path? If so, which ones?

4. Compare the Lifetime and Active time of each of the `tpoolComuptePot` (worker) threads.

Thread 2 Lifetime: _____	Thread 2 Active time: _____
Thread 3 Lifetime: _____	Thread 3 Active time: _____

What does this tell you about the application’s threaded execution?

Should we be concerned with the Serial cruise time within Thread 1?

Fixing the Performance Issue

Note: The difference in active times of the worker threads indicates that there is a load imbalance between the amounts of computation assigned to these threads. You will need to reconfigure how work is assigned to each thread, in order to achieve a more balanced amount of work between the worker threads.

1. Bring up the source code within the Microsoft Visual Studio window.
2. Notice that the first loop within the main routine statically divides up the particle iterations based on the number of worker threads that will be created within the thread pool.

In the `computePot` routine, each thread uses the stored boundaries indexed by the thread’s assigned identification number (`tid`), to fix the start and end range of particles to be used. However, the inner loop within this routine uses the outer index within the exit condition. Thus, the larger the particle number used in the outer loop, more iterations of the inner loop will be executed. This is done so that each pair of particles contributes only once to the potential energy calculation.

There are two obvious ways to fix this load imbalance:

A. Because the computation for each pair of particles considered will be equivalent, modify the code to:

1. find the number of such computations that will be done $((N^2)/2 - N)$
2. divide this by the number of threads being used
3. compute groupings of particles that will yield the closest set of computations calculated in the previous step, and
4. statically set the bounds array entries to create these groupings to be assigned to threads.

B. Use a more dynamic assignment of particles to threads. For example, rather than assigning consecutive groups of particles, as in the original version, have each thread, starting with the particle indexed by the thread id `tid`, compute all particles whose particle number differ by the number of threads. For example, when using two threads, one thread handles the even-numbered particles while the other thread handles the odd-numbered particles.

The first scheme involves considerable amount of code modifications, but will still be scalable for different numbers of threads and/or number of particles, and still achieve a good load balance. The

second scheme will achieve similar results and scalability, but requires much less code modifications.

3. Modify the physics simulation code to achieve a better load balance between the worker threads. You can use one of the solutions outlined above, or use one of your own design.

4. Re-run your modified code through the Thread Profiler to see if you have achieved the results you desired.

Note: After making such changes, you would normally run the modified application through the Thread Checker to ensure that no new threading errors had been introduced. If you have achieved sufficient load balance from the modified code and have the time, you should test the application with the Thread Checker.

[Finding Synchronization Contention Issues](#)

The application computes an approximation to the value of the constant Pi using the “midpoint (rectangle) rule” of numerical integration. The worker threads that are created compute the area of rectangles using the function value at selected points as the height of a rectangle. The results of each height computation are stored into a global sum. Access to this global variable is (correctly) protected by a CRITICAL_SECTION object. Even though the number of rectangle-area computations has been divided equally between threads, there is still a performance issue with this code.

Build & Run Threaded Numerical Integration Program

1. With Windows Explorer*, open the folder

D:\classfiles\ThreadProfiler\Numerical Integration\.

2. Double-click the **TP_Lab3.sln** icon to start Microsoft Visual Studio


3. Select the **Release** build, ensure the debug and linker options are set as needed, and build the application.

4. Launch VTune Performance Analyzer and create a new Thread Profiler activity that will run the executable generated from the previous step.


5. Run the application from within the Thread Profiler.


Evaluate Results to Diagnose Problem

On completion of the run, you should see the Critical Path view. There is a large portion of the execution along the critical path that was spent in “Serial impact time.” If there is a way to eliminate or reduce this, the application will run faster. Also, there is quite a bit of time being spent in threading overhead.

1. Double click on the critical path bar to go to the profile view. If the Concurrency Level view does not open as the default, click on the “Concurrency Level”  button in the toolbar.

What is this view telling you? _____

2. Select the “Threads View”  by clicking on that button. This view shows all the threads in the applications and how those were active in this critical path. Is there a load imbalance between the worker threads?

3. Select “Objects View”  by clicking on button. This view will show that the application is impacted by one Critical Section object where all of the accumulated impact is present.

4. Select a second level grouping in the Objects view by clicking on that button and choosing


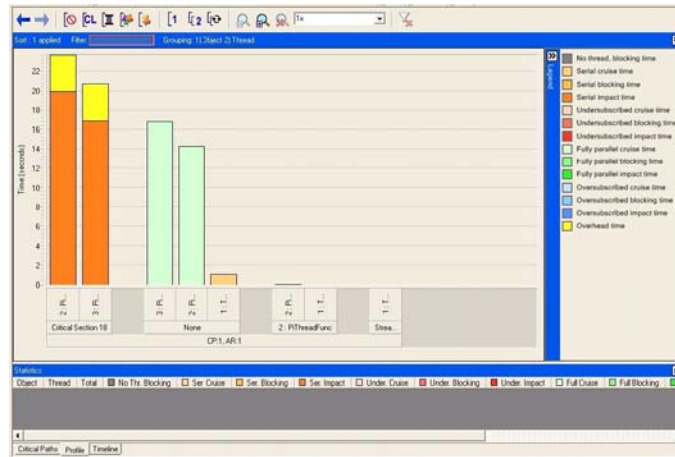
“Thread”  to bring up the view shown in [Figure 7.2](#). Both worker threads are impacted by the same Critical Section Object. The next few steps show how to get from the impacting object to the source line.

Figure 7.2. Second Level Grouping under Objects View

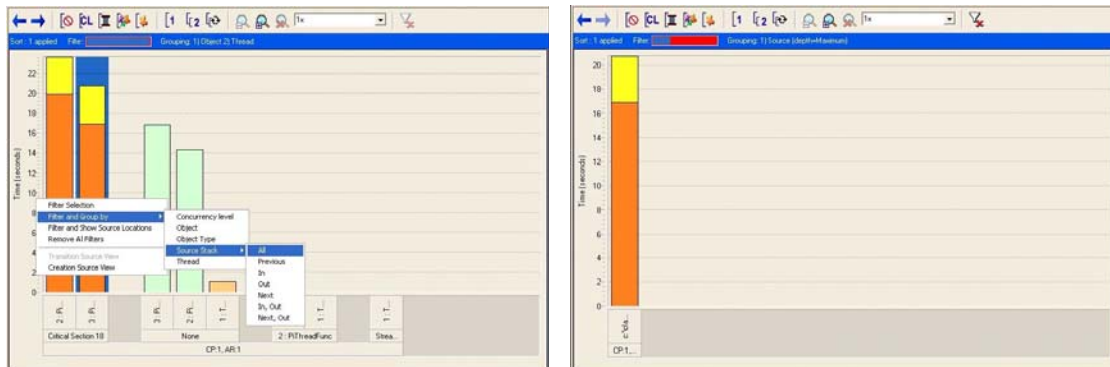


Both worker threads are impacted by the same Critical Section Object. The next few steps show how to get from the impacting object to the source line.

1. Select one of the Threads bar impacted by the critical section object to see the pop-up menu shown in [Figure 7.3](#).
 2. Select “Filter and Group by” drop down menu and in the drop-down, select “Source All”.
- This brings up a screen shown in [Figure 7.4](#).

Figure 7.3. Filtering by Object and Grouping by Source Code Locations

Figure 7.4. Filtered Profile View

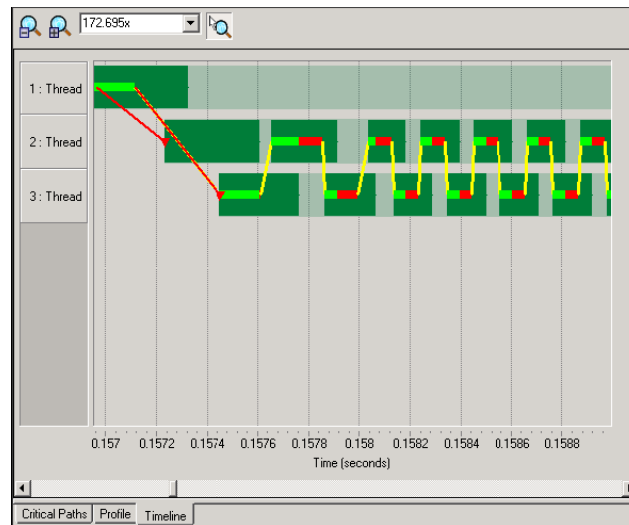


3. Right click on the bar graph and select “Transition Source View” in the selection. This above operation will take you to the source location where the object is used and shows the transition information.

Using the Timeline View for Analysis

1. Return to the Profile view. Remove the filtering by clicking on the <funnelX> icon. (Though it is not necessary, you may also want to go back to a simple display by clicking the <Obar> icon.)
2. Click on the Timeline tab to bring up the Timeline view. If you did not have “Transitions” set when the application was run, modify the collectors. Re-run the application in the Thread Profiler; click on the Timeline tab after results are displayed.
3. Zoom (click and drag) into a portion near the middle of the timeline that has multiple threads running. Continue zooming in until you have a view similar to the one shown in [Figure 7.5](#).

Figure 7.5. Timeline View (Zoomed)



What is this view telling you? _____

4. Hold the pointer over one of the red impact bars. The tooltip box details which synchronization object was involved and identifies the threads involved.
5. Hold the pointer on one of the transition lines. The tooltip popup details the synchronization object, the threads involved and the source lines with the transition of the critical path from one thread to another.
6. Right click on a transition line. Choose “Transition Source View” from the menu popup to open a Transition Source window.

This will show you the source code location where the critical path transitioned from one thread to another (Out).

Fixing the Performance Issue

The repeated access and contention on the `CRITICAL_SECTION` object causes a large part of the run to be executed in serial. If more worker threads were used, the problem would be worse, as more threads would sit idle waiting to acquire the synchronization object.

1. Modify the numerical integration code to achieve better performance.

To fix the contention problem, declare a variable in the `PiThreadFunc` function to collect the partial sums of the rectangle areas within each thread. A copy of this variable will be local to each thread and not require synchronization. Once a thread has completed calculating all the assigned rectangles, the thread should update the global sum with the local partial sum. This update should be protected. Thus, the protected global update is done once for every thread, rather than once for every rectangle computation.

2. Re-run the updated application through the Thread Profiler to ensure that performance has improved.

Comparing Performance Runs

1. Click on the “Critical Paths” tab.
2. Drag the original Thread Profiler results of this application from the Tuning Browser to compare with the final results you achieved.
3. Highlight both bars (CTRL+click on bars not highlighted).
4. Examine the results displayed in the “Profile” and “Timeline” tabs.

Is there a noticeable improvement from the original execution performance?

Review Questions

1. Is oversubscription of processor resources by active threads a severe performance problem?
Yes No
2. Combining views within the Profile View can give you more information than what is discernible from single views. What can you tell by combining the Objects and Threads views in the Thread Profiler?
3. Was the Timeline view useful?
4. Can applications instrumented for Timeline view be used in performance comparisons?
Yes No