Multi-Core Programming

For Academia

Student Workbook

Intel® Software College

Multi-Core Programming

For Academia

Student Workbook

Intel® Software College

Contents

Lab. 1: Intel® VTune™ Performance Analyzer 9

1: Finding Hotspots 11
- Collect Sampling Data for the Clockticks Event 11
- Review Questions 12

2: Sampling Over Time 13
- Create a Sampling Activity 13

3: Using Call Graph 14
- Collect call graph data 14

4: Using the Windows Command Line Interface 15
- Collect Samples based on Clockticks Event using Command Line Interface 15
- View the Profiling Data for gzip 15
- Pack the Data and View it in the GUI 15

Lab. 2: Intel® Math Kernel Library 17

1: Matrix Multiply Sample 19

2: Monte Carlo Calculation of Pi 20

Lab. 3: Programming with Windows* Threads 21

1: Starting with HelloThreads 23
- Build and Run HelloThreads Program 23
- Review Questions 26

2: Approximating Pi with Numerical Integration 27
- Build and Run the Serial Program 27
- Correct Errors and Validate Results 27
- Review Questions 28

3: Using Events 29
- Build and Run Original Threaded Program 29
- Modify Original Threaded Program to Use Events 29

4: Using Semaphores 30
- Build and Run Serial Program 30
- Build and Run Threaded Program 30

Lab. 4: Programming with OpenMP Threads 31

1: Starting with Hello Worlds 33
- Initial Compile 33
- Add OpenMP Directives 33
- OpenMP Compile 33

2: Computing Pi with Numerical Integration 35
- Initial Compile 35
- Add OpenMP Directives 35
- OpenMP Compile and Run 36

3: Monte Carlo Pi 37

Initial Compile 37
Add OpenMP Directives 37
OpenMP Compile 38

Review Questions 39

Lab. 5: Correcting Threading Errors with Intel® Thread Checker for Explicit Threads **41**

1A: Find Prospective Data Races 43
Build and Run Potential Serial Program 43
Build and Run Potential Threaded Program 43

1B: Resolve Data Races 45
Resolve the Problems 45

2: Identifying Deadlock 46
Build and Run the Program 46
Run Application within Intel® Thread Checker 48
Correct Errors and Validate Results 49
Review Questions 50

3: Testing Libraries for Thread Safety 51
Setting-up and Compilation for Thread Safety Testing 51
Modifying Binary Instrumentation Levels 52
Review Questions 53

Lab. 6: Tuning Threaded Code with Intel® Thread Profiler for Explicit Threads **55**

1A: Getting Started with Thread Profiler 57
Review Questions 58

1B: Analyzing an Application 59

2: Finding Load Balance Issues 61
Build and Run Potential Threaded Program 61
Evaluate Thread Profiler Results to Diagnose Problem 61
Fixing the Performance Issue 62

3: Finding Synchronization Contention Issues 64
Build & Run Threaded Numerical Integration Program 64
Evaluate Results to Diagnose Problem 64
Using the Timeline View for Analysis 67
Fixing the Performance Issue 68
Comparing Performance Runs 69

Review Questions 70

Lab. 7: Threaded Programming Methodology **71**

1: Compile and Run Serial Code 73
Build and Run Serial Program 73

2: Analyze Serial Code 74
Get Baseline Timing 74
Run VTune Performance Analyzer on Application 74

3: Run OpenMP Code 75

Build and Run Threaded Program 75

4: Use Thread Checker to Check for Threading Errors 76

Use Thread Checker for Analysis 76

5: Correct Threading Errors 77

Correct Errors and Validate Results 77

Build and Run Threaded Program 77

6: Find Threading Performance Issues 78

Instrument and Run Threaded Program 78

Use Thread Profiler for Analysis 78

7: Reduce the Number of printf Calls 79

Modify Code to Control Number of Progress Updates 79

Update Serial Code 80

8: Modify Critical Regions 81

Use InterlockedIncrement and Local Variables 81

9: Fix Load Imbalance 83

Use OpenMP schedule Clause 83

Lab. 8: Scalability of Threaded Applications 85

Workload-dependent Scaling - An Overview 87

Introduction to Sample Application 87

1a: Build the Application 89

1b: Characterize Multithreaded Implementation with Large Data Set 90

1c: Examine Thread Performance with Large Data Set 91

1d: Characterize Multithreaded Implementation with Small Data Set 92

2: Illustrate Effects of Load Balance in Multithreaded Implementation 94

3: Measuring Synchronization Object Overhead 95

4: Measuring Front-side Bus Saturation 96

5: Identifying False Sharing in Threaded Applications 98

Review Questions 100

Answers 100

Appendices 101

A: Setting-up Intel® Thread Checker 103

Setting Compile & Link Options 103

Creating a New Project and Thread Checker Activity 106

B: Setting-up Intel® Thread Profiler 109

Setting Compile & Link Options 109

Creating a New Project and Thread Checker Activity 113

Creating a New Activity by Modifying an Existing Activity 116

Additional Resources 119

Lab. 1: Intel® VTune™ Performance Analyzer

Time Required	One hour
Objectives	The sampling profiler in the Intel® VTune™ Performance Analyzer collects system-wide data, has very low overhead, and is easy to use. These labs will familiarize you with the different features and parameters of the profiler and help you interpret its results. At the successful completion of this lab, you will learn to: • sample-based profiling, VTune analyzer sampling user interface and call graph

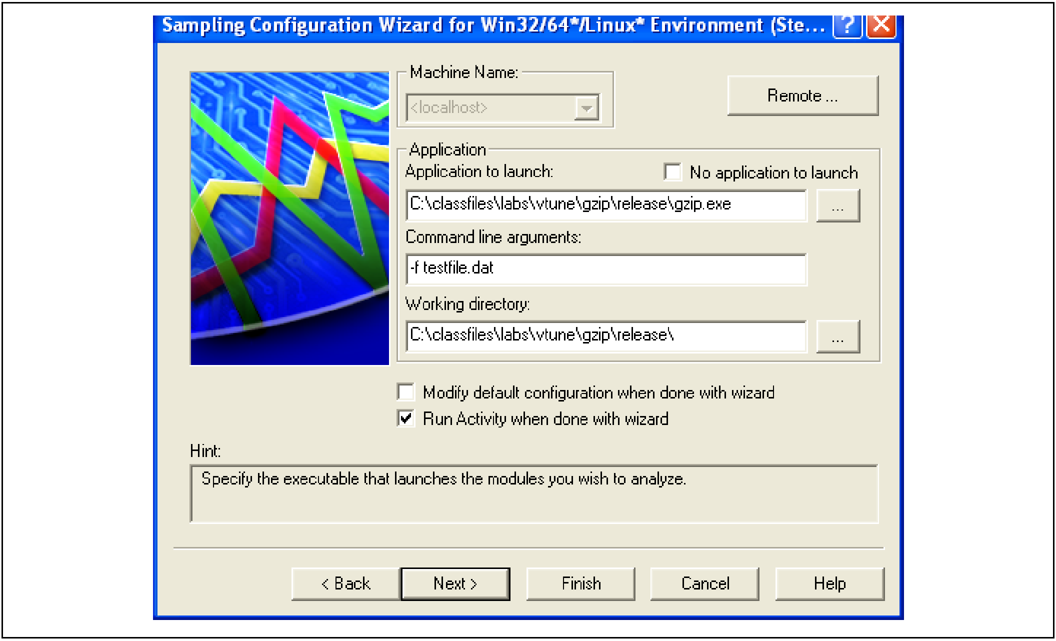
Activity 1: Finding Hotspots

In this activity, you will collect sampling data for the program gzip.exe and identify the function that consumes the most execution time.

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 on Category -> Analyzer Projects ->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 the full path of the application, for example:
   `c:\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 1.1. Sampling Configuration Wizard


Sampling Configuration Wizard for Win32/64*/Linux* Environment (Ste...
Remote ...
...
...
< Back
Next >
Finish
Cancel
Help

Machine Name: <localhost>

Application
Application to launch: ☐ No application to launch
C:\classfiles\labs\vtune\gzip\release\gzip.exe
Command line arguments:
-f testfile.dat
Working directory:
C:\classfiles\labs\vtune\gzip\release\

☐ Modify default configuration when done with wizard
☑ Run Activity when done with wizard

Hint:
Specify the executable that launches the modules you wish to analyze.

Review Questions

Question 1:	What function in gzip.exe takes the most time?
Question 2:	Which function in gzip.exe has the highest CPI?
Question 3:	Which line of source in gzip.exe has the most clocktick samples?
Question 4:	Is gzip.exe multithreaded?

Activity 2: Sampling Over Time

In this activity, you learn how to use the Sampling Over Time view. You analyze the performance of a tbd.

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:

```
c:\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 were 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.

Activity 3: Using Call Graph

In this activity, you will collect call graph data for the program gzip and identify the function that takes the most time and the functions that call it.

Collect call graph data

1. Create a new Activity by clicking on Activity->New Activity.
2. Double click Call Graph Wizard.
3. Navigate to the classroom directory and locate the gzip.exe application.
4. Click Next.
5. Enter the full pathname for the application, for example: c:\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.

Question 1: What function has the most time spent in it and what functions are calling it?

Activity 4: Using the Windows Command Line Interface

In this activity, you learn how to use the Windows* Command Line Interface to profile an application.

Collect Samples based on Clockticks Event using Command Line Interface

1. Open the command prompt by clicking Start->Programs->Accessories->Command Prompt.
2. Go to the following directory by typing:c:\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.
2. 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.
3. Start VTune Performance Analyzer (For example, by double-clicking the desktop icon, if present.)
4. Click Browse for Existing File.
5. Open: c:\classfiles\VTuneBasics\gzip\release\gzip.vxp
6. Click OK.
7. Click OK.
   The GUI displays the performance data.

Lab. 2: Intel® Math Kernel Library

Time Required	Twenty minutes
Objectives	In this lab, you will learn to use Intel® MKL subroutines and Intel® MKL's multithreading capability. At the successful completion of this lab, you will be able to: • Build images using functions linked from Intel MKL • Invoke Intel MKL's multithreading capability

Activity 1: 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. Note differences in timings among the different implementations.
3. 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
```

4. Observe the performance at different numbers of threads

Question 1: What happens then that the number exceeds the physical processors?

Activity 2: 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.

Question 1: 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().

Question 2: Why is this not a 1:1 substitution for rand()?

Question 3: What is the purpose of, and sensitivity to, blocksize?

Question 4: What are the parameters BRNG and VSL_BRNG_MCG31?

Question 5: Are they the best choices for this computation?

Question 6: Could this implementation be threaded?

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 "-up" 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.

Lab. 3: Programming with Windows* Threads

Time Required	Twenty-five minutes
Objectives	In this lab session, you will practice multi-threaded programming of applications using Win32* threading API. Sample programs will create threads, assign work to those threads, synchronize access to shared resources, and coordinate thread execution. After successfully completing this lab's activities, you will be able to: • Find and resolve a common data race involving parameter passing • Convert a serial application to a threaded version by encapsulating computations into a function that is executed by threads • Find simple data races in code and resolve these threading errors using critical sections as the synchronization mechanism

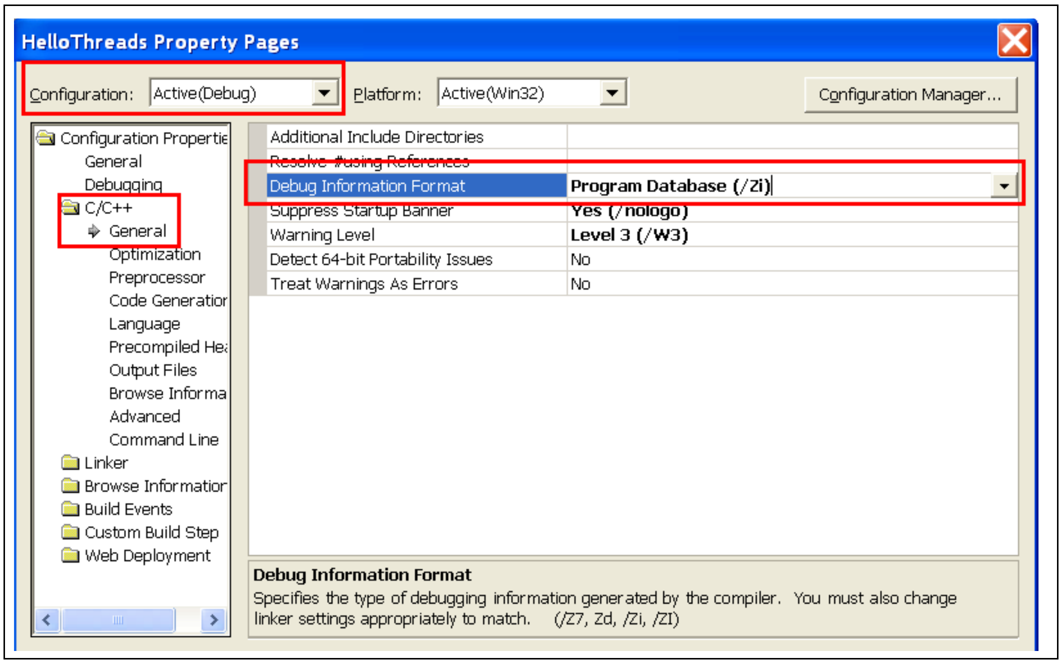
Activity 1: Starting with *HelloThreads*

Time Required	Ten minutes
Objective	• Find and resolve a common data race involving parameter passing

Build and Run HelloThreads Program

1. Close Microsoft Visual Studio, if it is started.
2. With Windows Explorer*, open the folder C:\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 3.1.

Figure 3.1. Project Setting – C/C++ Folder – Debug Options



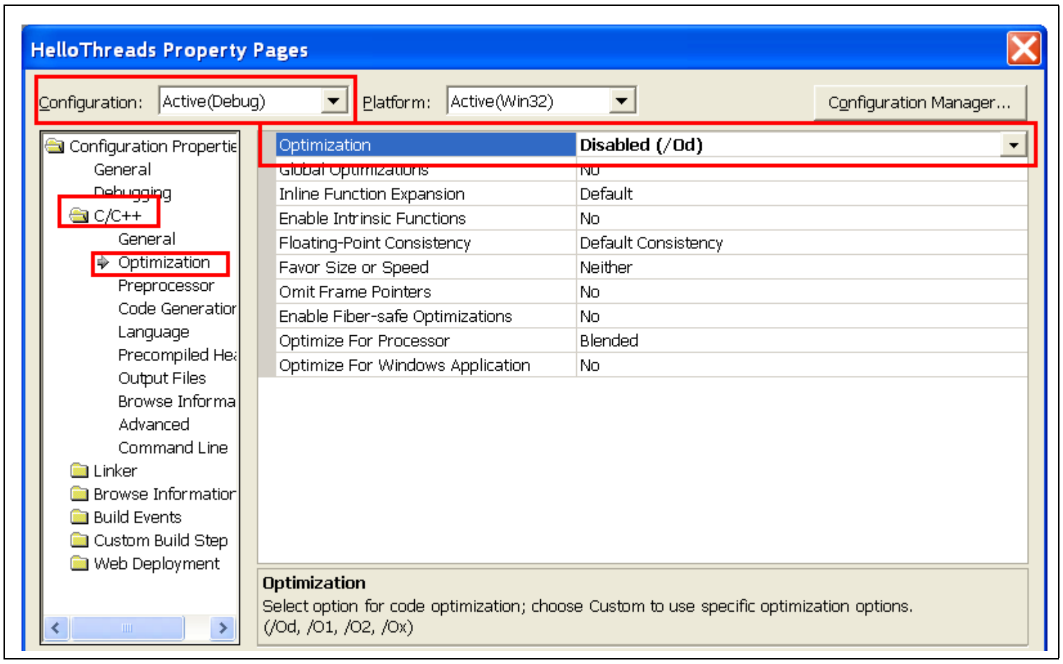
Configuration: Active(Debug)   Platform: Active(Win32)

Additional Include Directories	
~~Resolve #using References~~	
Debug Information Format	Program Database (/Zi)
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level 3 (/W3)
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No

Debug Information Format
Specifies the type of debugging information generated by the compiler. You must also change linker settings appropriately to match. (/Z7, Zd, /Zi, /ZI)

7. Make sure that Optimization is disabled, as shown in Figure 3.2.

Figure 3.2. Project Settings – C/C++ Folder – Optimization Options

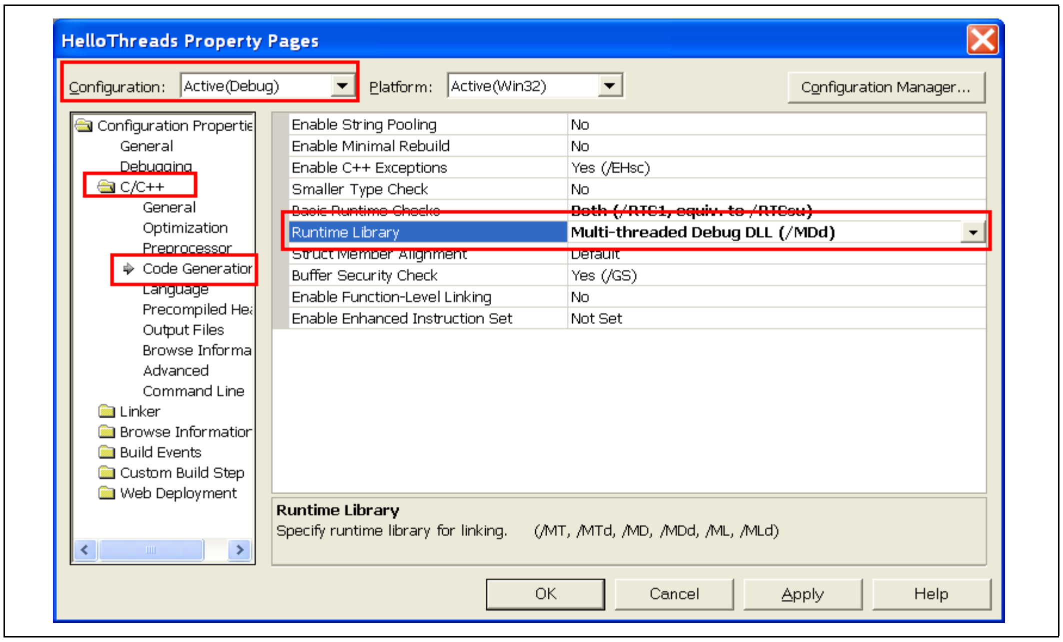


Optimization	Disabled (/Od)
Global Optimizations	No
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Floating-Point Consistency	Default Consistency
Favor Size or Speed	Neither
Omit Frame Pointers	No
Enable Fiber-safe Optimizations	No
Optimize For Processor	Blended
Optimize For Windows Application	No

Optimization
Select option for code optimization; choose Custom to use specific optimization options. (/Od, /O1, /O2, /Ox)

8. Make sure that thread-safe libraries are selected, as shown in Figure 3.3.

Figure 3.3. Project Settings – C/C++ Folder – Thread-safe Libraries Options

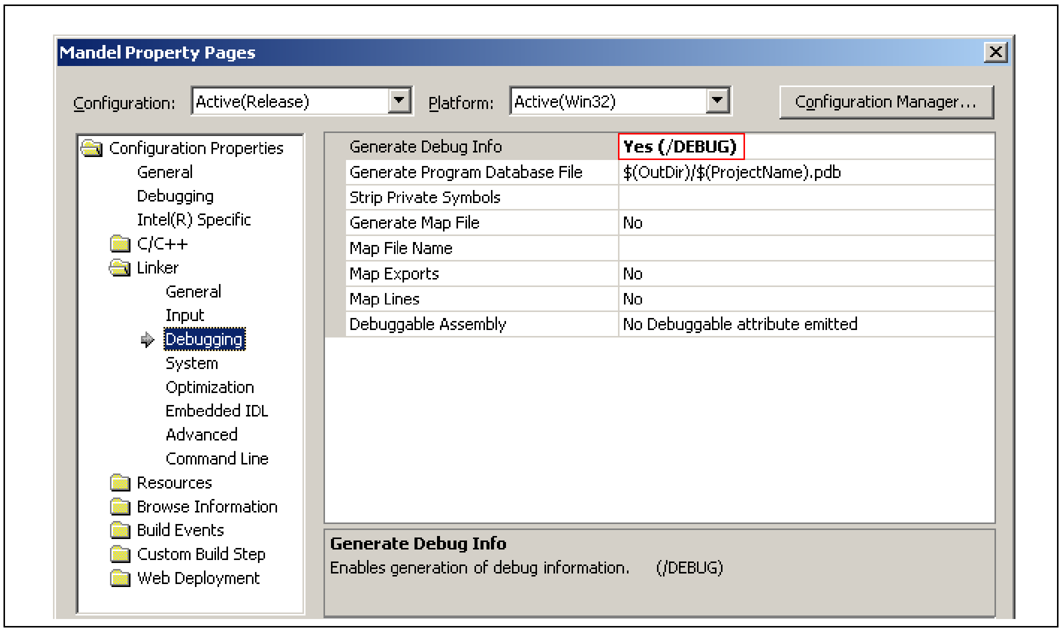


Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu)
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Buffer Security Check	Yes (/GS)
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd, /ML, /MLd)

9. Make sure that Debug symbols are preserved during the link phase, as shown in Figure 3.4.

Figure 3.4. Linker Settings – Debugging Folder



Mandel Property Pages

Configuration: Active(Release) Platform: Active(Win32) Configuration Manager...

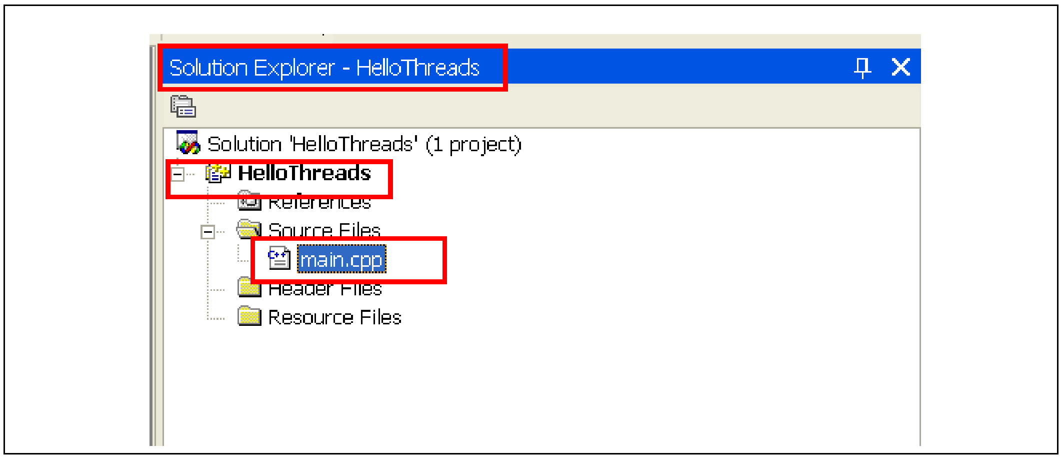
Configuration Properties
- General
- Debugging
- Intel(R) Specific
- C/C++
- Linker
  - General
  - Input
  - Debugging
  - System
  - Optimization
  - Embedded IDL
  - Advanced
  - Command Line
- Resources
- Browse Information
- Build Events
- Custom Build Step
- Web Deployment

Generate Debug Info	Yes (/DEBUG)
Generate Program Database File	$(OutDir)/$(ProjectName).pdb
Strip Private Symbols	
Generate Map File	No
Map File Name	
Map Exports	No
Map Lines	No
Debuggable Assembly	No Debuggable attribute emitted

Generate Debug Info
Enables generation of debug information. (/DEBUG)

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 3.5.

Figure 3.5. Solution Explorer – HelloThreads Project



Solution Explorer - HelloThreads

Solution 'HelloThreads' (1 project)
- HelloThreads
  - References
  - Source Files
    - main.cpp
  - Header Files
  - Resource Files

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).

Tip: 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

Question 7: The execution order of threads is unpredictable.

True False

Question 8: What build options are required for any threaded software development?

Activity 2: Approximating *Pi* with Numerical Integration

Time Required	Fifteen minutes
Objectives	• Thread the numerical integration application. • Find any data races in the code and resolve the threading errors using critical sections as the synchronization mechanism.

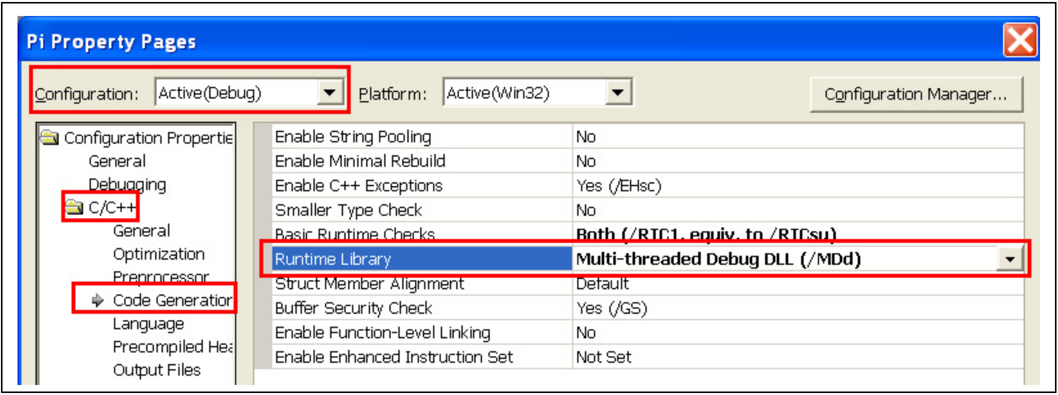
Build and Run the Serial Program

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 3.6.

Figure 3.6. Project Settings – C/C++ Folder – Thread-safe Libraries Options



Configuration: Active(Debug)
Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu)
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Buffer Security Check	Yes (/GS)
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

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).

Tip: 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.

Tip: A complete solution to the lab is provided in the file PiSolution.cpp, which is located in the following directory: C:\classfiles\Multi-Core\Windows\Win32 Threads\Pi\

Review Questions

Question 1: All threads should use the same CRITICAL_SECTION.

True False

Question 2: Threading errors in software can always be corrected by using only synchronization objects.

True False

Question 3: CRITICAL_SECTION objects should always be declared as global variables.

True False

Question 4: What build options are required for *any* threaded software development?

Activity 3: Using Events

Time Required	Fifteen minutes
Objective	Use Windows events to signal when computation threads have completed the assigned work.

The application computes an approximation of the natural logarithm of (1 + x), -1 < x <= 1, using the Mercator series. Compute threads are created in a suspended mode. These threads are released to compute the series elements that have been assigned after a "master" thread has been. The master thread waits on a thread count variable that is incremented by each compute thread as it finishes. Once all threads have completed computing the partial sums, the master thread does a final summation and terminates. Results are printed by the process thread after cleaning up all the objects and handles.

Build and Run Original Threaded Program

1. With Windows Explorer*, open the folder C:\classfiles\Win32 Threads\N_ThreadEvent.
2. Double-click on the N_ThreadEvent.sln icon.
3. Within Microsoft* Visual Studio, examine the source code until you understand how the threads are created and interact with the others.
4. Build the solution and run the application using the Start Without Debugging command from the Debug menu.

Modify Original Threaded Program to Use Events

1. Modify the threaded code to use events to signal the master thread rather than the count of a global variable. Some hints for how to do this are given below:
   a. Create an array of event handles, one event per thread. This can be done like the thread handle array is allocated. Be sure to initialize each event.
   b. Replace the spin-wait in the master thread with a wait on all the events being signaled from each compute thread.
   c. Replace the protected increment of the thread count variable with a signal of the event in the array that is indexed with the thread number.
   (Do you still need the critical section?)
2. Build the solution and run the application using the Start Without Debugging command from the Debug menu.

Activity 4: Using Semaphores

Time Required	Ten minutes
Objective	• Identify global data accessed by threads; resolve data races using binary semaphores

The application opens an input text file. Threads read in lines from the text file and count the total number of words in the line, as well as the number of words with an even number of letters and an odd number of letters. When done, the text file is closed and the final totals are printed.

Build and Run Serial Program

1. With Windows Explorer, open the folder C:\ classfiles\Win32 Threads\SemaphoreLab\.
2. Double-click on the SemphoreLab.sln icon. Within Microsoft* Visual Studio, you will find two projects. One is the serial version of the code and the second is the first attempt at threading that code with Win32 Threads.
3. Be sure the Serial project has been selected as the Startup Project. Build the solution and run the serial application using the Start Without Debugging command from the Debug menu.
4. Note the output generated:

Total Words: _______________
Total Even Words: ___________
Total Odd Words: ____________

Build and Run Threaded Program

1. Set the Threaded project as the Startup Project. Build the solution and run the threaded application using the Start Without Debugging command from the Debug menu. Do you get the same answers as above?.
2. Examine the source code. Identify the global variables that are being accessed by each thread.
3. Rewrite the threaded application to protect the use of these global variables. Wherever mutual exclusion is needed in your solution, use a binary semaphore. Be sure to declare the semaphores at the proper level and initialize them before use.
4. Build and run the threaded code until you are able to achieve the same totals as the serial version of the application.

Lab. 4: Programming with OpenMP Threads

Time Required	Thirty minutes
Objectives	In this lab session, 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. After successfully completing this lab's activities, you will be able to: • Use the most common OpenMP* C statements • Compile and run an OpenMP* program

Activity 1: Starting with *Hello Worlds*

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

Initial Compile

1. Within the Intel Compiler build environment, navigate to the folder OpenMP\HelloWorlds.
2. Compile serial code using the Intel compiler:

```
icl HelloWorlds.c
```

3. Run the program:

```
HelloWorlds.exe
```

Add OpenMP Directives

1. Using the editor of your choice (in Windows, the likely candidates will be Notepad or Visual Studio), open the source file and 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]...

}
```

Question 1: Do you need to include omp.h in the source file? Why or why not?

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.

Activity 2: Computing Pi with Numerical Integration

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

Initial Compile

1. Navigate to the folder OpenMP\Pi.
2. Open the solution file pi.sln (double-click on the file).
3. Build and run this project to ensure that this serial version runs correctly.
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. Update the Project settings to include the compiler flag /Qopenmp
2. Build and run the program.
3. Record the time:__________________.
4. In a command shell (as in Activity 1), run the program with different settings of OMP_NUM_THREADS, and record the time.
   Do the resulting scalings meet your expectations?

Activity 3: Monte Carlo *Pi*

In this activity, you convert a serial MKL-using version of a Monte Carlo Pi program to a parallel version, using OpenMP directives. The working environment is Visual Studio 2005.

Initial Compile

1. Navigate to the folder OpenMP\MonteCarloPi.
2. Open the solution file MonteCarloPi.sln (double-click on the file).
3. Build and run this project to ensure that this serial version runs correctly.

Note: You will want to be sure that the MKL include and library paths are set in the project and are correct for the system in use.

4. Record the serial time: __________________.

Add OpenMP Directives

1. Determine the section of code to make parallel and add the OpenMP parallel directive:
2. 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 helpful:

```
#pragma omp parallel private(varname,varname)\
reduction(+:varname,varname) \

shared(varname,varname)

{

...[Code to run in Parallel goes here]...

}
```

3. 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. Update the Project settings to include the compiler flag /Qopenmp
2. Build and run the program.
3. Record the time:___________________.
4. In a command shell (as in Activity 1), run the program with different settings of OMP_NUM_THREADS, and record the time.
   Do the resulting scalings meet your expectations?

Review Questions

Question 1: Name the pragma or directive that would split a loop into multiple threads.

Question 2: What website has the very readable OpenMP spec?

Lab. 5: Correcting Threading Errors with Intel® Thread Checker for Explicit Threads

Time Required	Fifty-five minutes
Objectives	In this lab session, you will learn how to debug Win32* threaded applications and resolve data race conditions. You will learn how to use Intel® Thread Checker, including how to use it with binary instrumentation when Intel® Compilers are not used. Intel Thread Checker is a powerful tool for finding threading errors. After successfully completing this lab's activities, you will be able to: • Find and resolve data races using simple threading techniques • Find actual and potential threading errors, and validate that the errors are fixed using Intel Thread Checker • Use Intel Thread Checker to determine if libraries are thread-safe • Use the advanced features of Intel Thread Checker, including instrumentation levels

Activity 1A: Find Prospective Data Races

Time Required	Ten minutes
Objective	• Find and run Intel Thread Checker to find any data races within a simple physics model code

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 C:\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 C:\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 (See Appendix A: “Setting-up Intel® Thread Checker” on page 103 for help):
    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?

Activity 1B: Resolve Data Races

Time Required	Ten minutes
Objective	• Resolve data races found in previous lab using simple threading techniques

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?

Activity 2: Identifying Deadlock

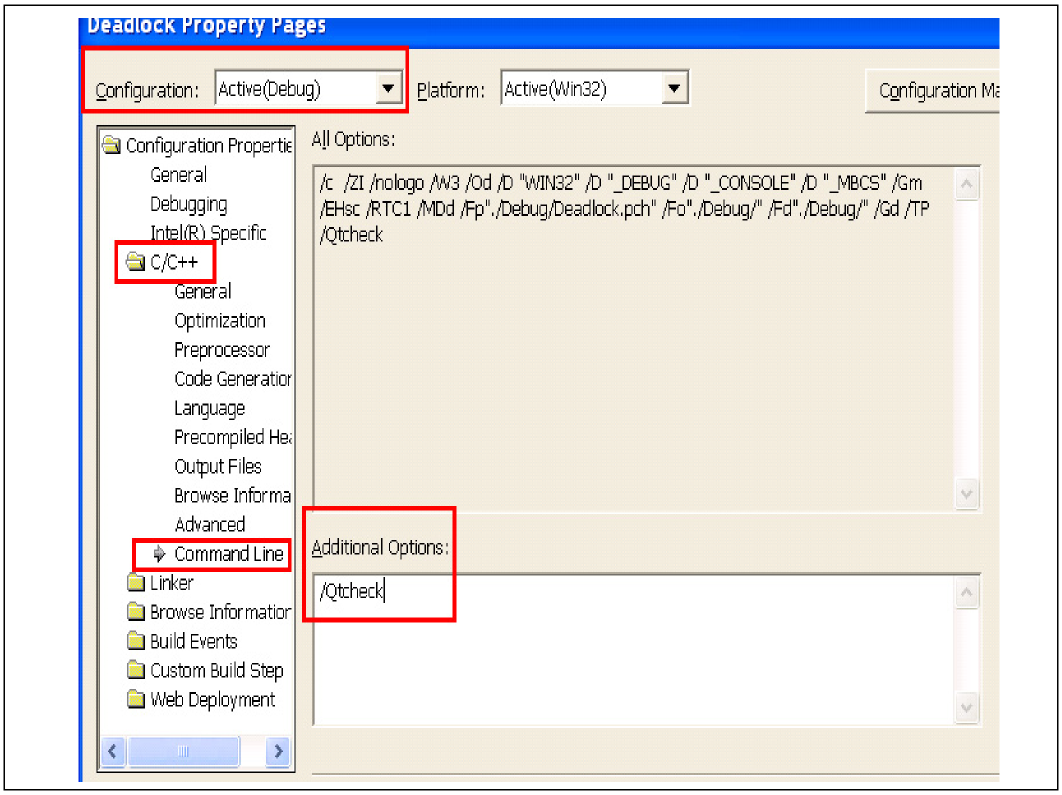
Time Required	Fifteen minutes
Objective	• Build software to use source code instrumentation for Intel Thread Checker • Find actual and potential threading deadlock errors, and validate that the errors are fixed using Intel Thread Checker

Build and Run the Program

1. With Windows Explorer, open the folder C:\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 (See Appendix A: “Setting-up Intel® Thread Checker” on page 103 for help).
    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 5.1.

Figure 5.1. Project Settings – C/C++ Folder – Compiler Options



```
/c  /ZI /nologo /W3 /Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_MBCS" /Gm
/EHsc /RTC1 /MDd /Fp"./Debug/Deadlock.pch" /Fo"./Debug/" /Fd"./Debug/" /Gd /TP
/Qtcheck
```

Additional Options:

```
/Qtcheck
```

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?

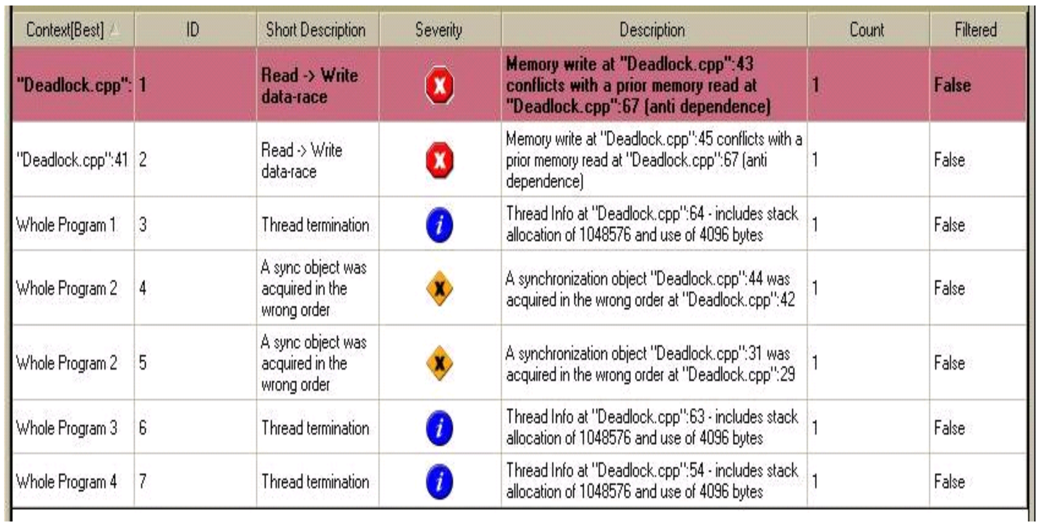
Run Application within Intel® Thread Checker

1. Start VTune Performance Analyzer and select **New Project**.(See Appendix A: “Setting-up Intel® Thread Checker” on page 103 for help.)
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 5.2 and Figure 5.3, because deadlock does not occur during every run.

Figure 5.2 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.

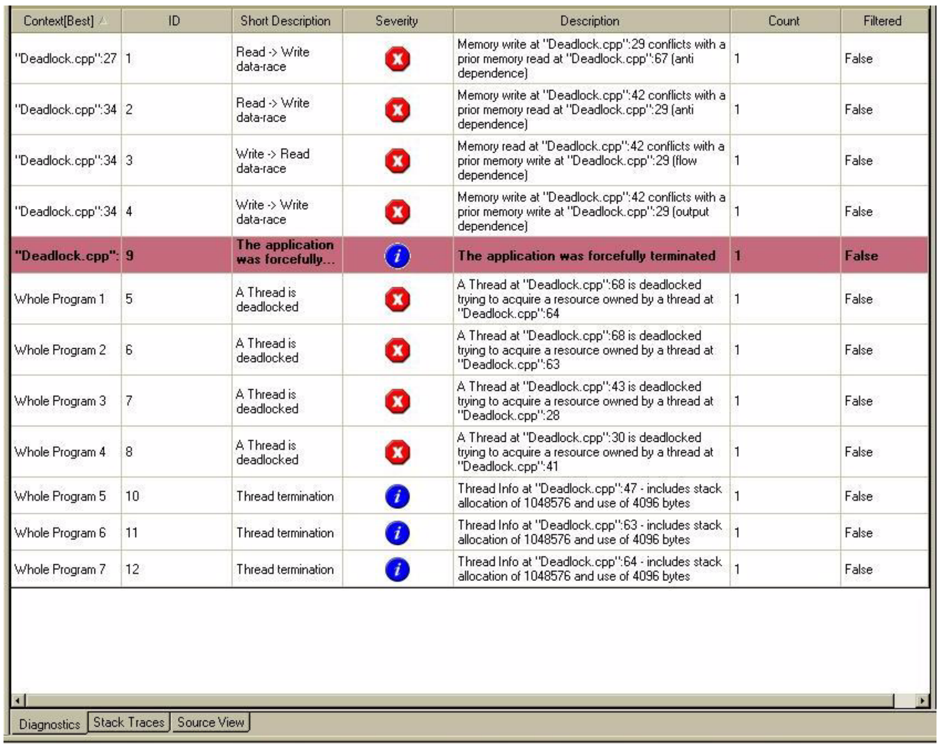
Figure 5.2. Diagnostic List When Deadlock Does Not Occur During a Run



Context[Best]	ID	Short Description	Severity	Description	Count	Filtered
"Deadlock.cpp":	**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	ℹ	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	ℹ	Thread Info at "Deadlock.cpp":63 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 4	7	Thread termination	ℹ	Thread Info at "Deadlock.cpp":54 - includes stack allocation of 1048576 and use of 4096 bytes	1	False

Figure 5.3 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.

Figure 5.3. 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...**	ℹ	**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	ℹ	Thread Info at "Deadlock.cpp":47 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 6	11	Thread termination	ℹ	Thread Info at "Deadlock.cpp":63 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Whole Program 7	12	Thread termination	ℹ	Thread Info at "Deadlock.cpp":64 - includes stack allocation of 1048576 and use of 4096 bytes	1	False

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.

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.

Tip: 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.

Tip: 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

Activity 3: Testing Libraries for Thread Safety

Time Required	Twenty minutes
Objective	• Use the advanced features of Intel Thread Checker, including instrumentation levels

Setting-up and Compilation for Thread Safety Testing

1. With Windows Explorer, open the folder C:\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?

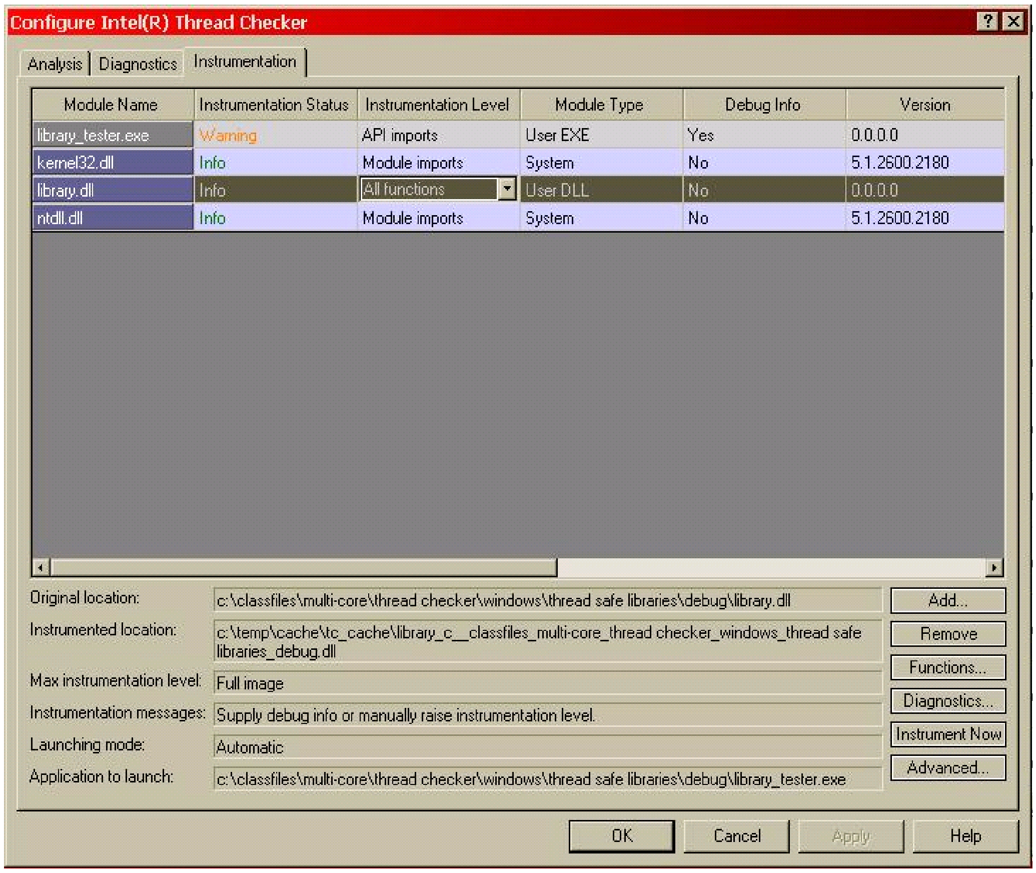
Modifying Binary Instrumentation Levels

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" dialog box(Figure 5.4).
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.

Figure 5.4. Setting binary instrumentation levels for DLLs and other modules



Module Name	Instrumentation Status	Instrumentation Level	Module Type	Debug Info	Version
library_tester.exe	Warning	API imports	User EXE	Yes	0.0.0.0
kernel32.dll	Info	Module imports	System	No	5.1.2600.2180
library.dll	Info	All functions	User DLL	No	0.0.0.0
ntdll.dll	Info	Module imports	System	No	5.1.2600.2180

Original location:	c:\classfiles\multi-core\thread checker\windows\thread safe libraries\debug\library.dll
Instrumented location:	c:\temp\cache\tc_cache\library_c__classfiles_multi-core_thread checker_windows_thread safe libraries_debug.dll
Max instrumentation level:	Full image
Instrumentation messages:	Supply debug info or manually raise instrumentation level.
Launching mode:	Automatic
Application to launch:	c:\classfiles\multi-core\thread checker\windows\thread safe libraries\debug\library_tester.exe

5. Click OK.
6. Re-run the Thread Checker analysis.
   Were any diagnostics found this time?

Review Questions

Question 1: Which combinations of functions are not thread safe?

Question 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?

Lab. 6: Tuning Threaded Code with Intel® Thread Profiler for Explicit Threads

Time Required	Forty-five minutes
Objectives	In this lab session, you will use Intel® Thread Profiler to detect performance issues in applications threaded with the Win32 threading API. After successfully completing this lab's activities, you will be able to: • Create new activities under the Thread Profiler • Use the Thread Profiler to determine system utilization • Navigate through the Thread Profiler features to understand program thread activity

Activity 1A: Getting Started with Thread Profiler

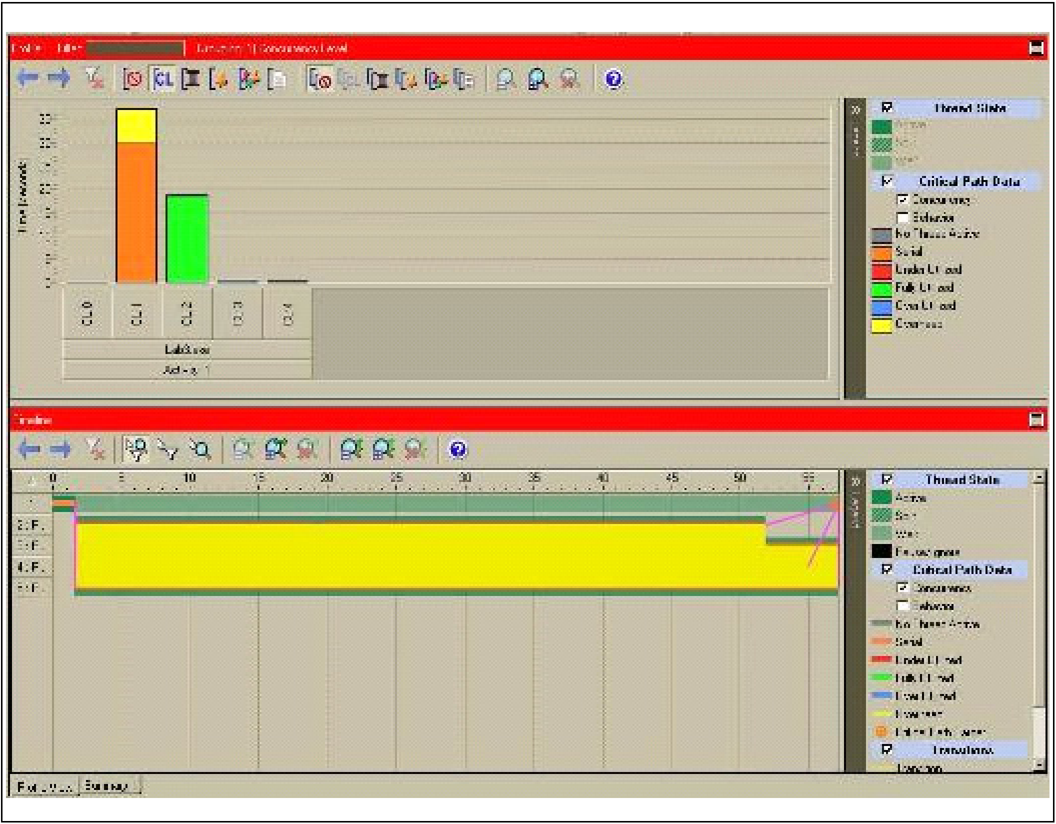
Time Required	Ten minutes
Objective	• Start up Intel Thread Profiler and examine the different analysis views offered by it

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 C:\classfiles\ThreadProfiler\potential lab 1\Potential Lab 1.sln.
2. Double-click the icon to open Microsoft Visual Studio and build the application. (See Appendix B: "Setting-up Intel® Thread Profiler" on page 109 for compilation and linker settings needed for the Thread Profiler analysis.)
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 split Profile View and Timeline View as shown in Figure 6.1.

Figure 6.1. Thread Profiler Default View Screen



6. Click on the expansion button in the upper right corner of the Profile View. You will see the Concurrency Level button in the toolbar has been automatically selected.

7. Click on the other grouping buttons (“Threads View” and “Object View”). Click on the bars and roll over portions of the windows to see what tooltips appear and what data is contained within them.

8. Click on the expansion button of 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.

Review Questions

Question 1: From what you've seen, how would you characterize the performance of this application?

Question 2: Are there any obvious performance issues?

Activity 1B: Analyzing an Application

Time Required	Ten minutes
Objective	• Examine the different analysis views offered by Intel Thread Profiler and determine if a performance issue is evident from the information presented

1. Return to the Profile View 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 and how each of those 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 Expand the "Timeline" 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?

Activity 2: Finding Load Balance Issues

Time Required	Fifteen minutes
Objective	• Use Intel Thread Profiler to find a load imbalance performance problem within a threaded application

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 C:\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

The Concurrency Level Profile View confirms that the majority of execution time is spent with only one thread active. If you switch on the Behavior categories for the Critical Path Data, a large percentage of time is spent in Impact time. This data means that a single thread is running and actively keeping other threads from executing, typically through some synchronization object.

1. Bring up the Object 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.

2. 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?

3. 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:

I. Because the computation for each pair of particles considered will be equivalent, modify the code to:
   i. find the number of such computations that will be done ((N**2)/2 - N)
   ii. divide this by the number of threads being used
   iii. compute groupings of particles that will yield the closest set of computations calculated in the previous step, and
   iv. statically set the bounds array entries to create these groupings to be assigned to threads.

II. 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.

Activity 3: Finding Synchronization Contention Issues

Time Required	Twenty minutes
Objective	• Use Intel Thread Profiler in order to find a synchronization contention performance issue within a threaded application. Demonstrate some of the filtering capabilities of the Thread Profiler.

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 C:\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

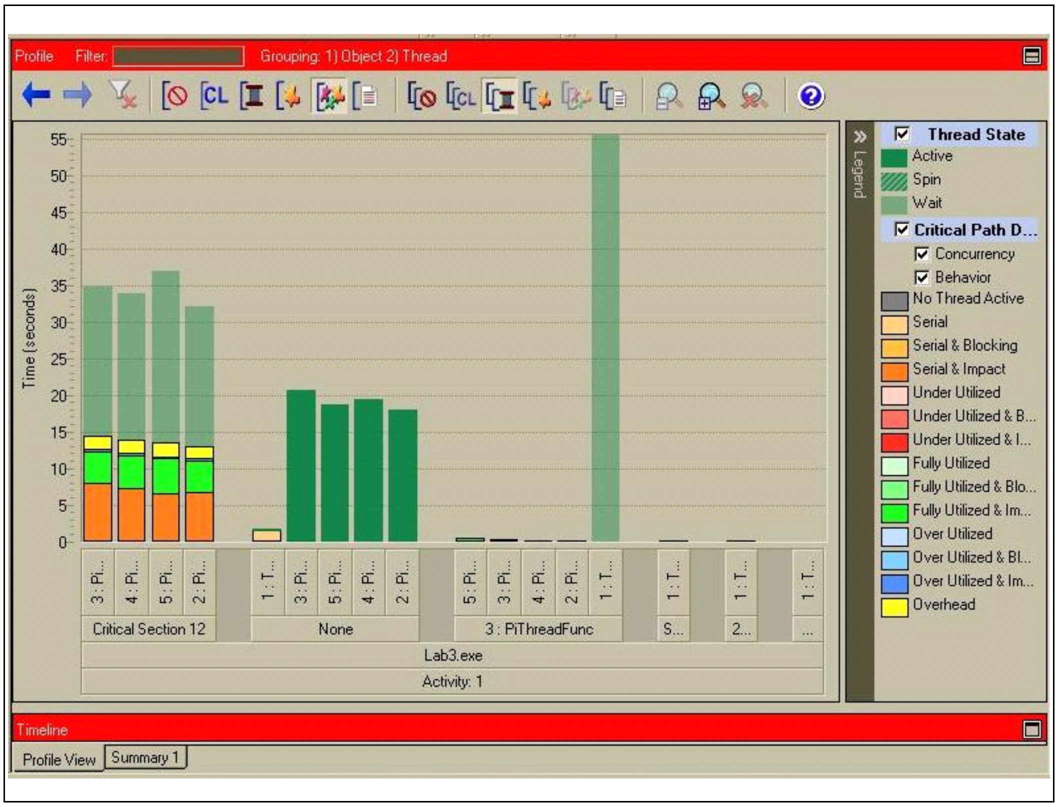
On completion of the run, you should examine the Concurrency Level view. There is a large portion of the execution along the critical path that was spent in "Serial impact time." If there is a way to eliminate or reduce this, the application will run faster. Also, there is quite a bit of time being spent 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 "Object View" by clicking on that 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 Object View by clicking on that button to choose Threads as the secondary grouping. This will bring up a view similar to the one shown in Figure 6.2.

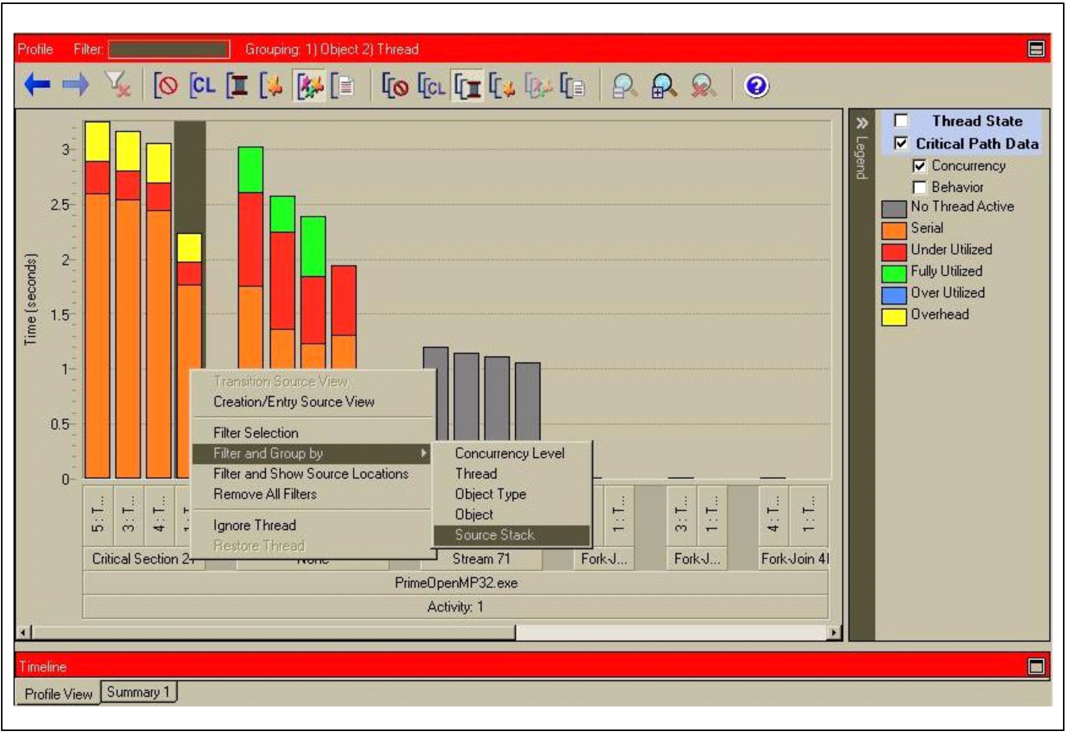
Figure 6.2. Second Level Grouping under Objects View



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.

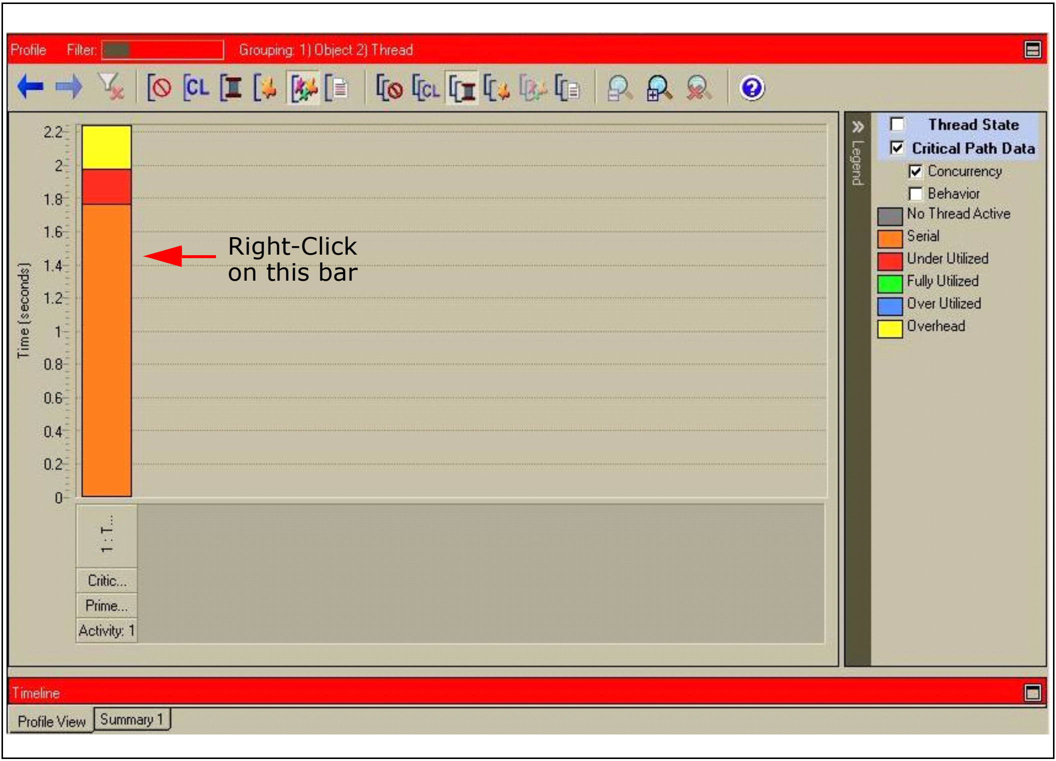
5. Select one of the Threads bar impacted by the critical section object to see the pop-up menu shown in Figure 6.3.

Figure 6.3. Filtering by Object and Grouping by Source Code Locations



6. Select "Filter and Group by" drop down menu and in the drop-down, select Source Stack and in the drop down, and then select All.
   This should bring up a screen similar to the one shown in Figure 6.4.

Figure 6.4. Filtered Profile View

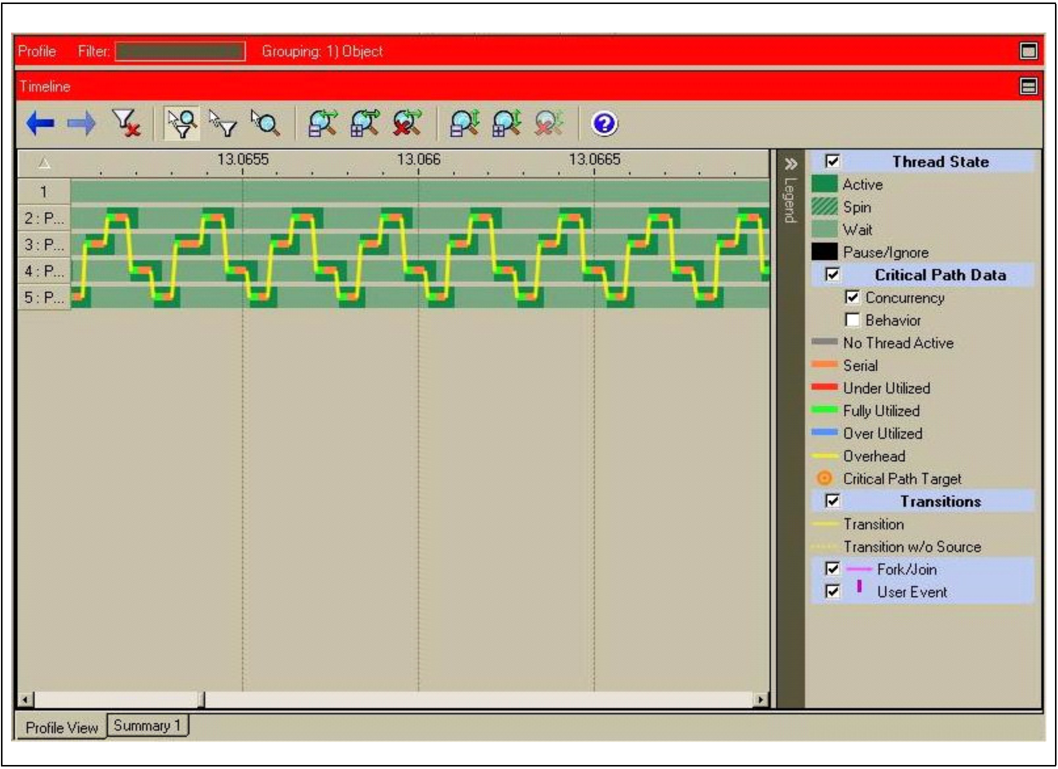


7. Right click on the bar graph and select "Transition Source View" in the selection. This above operation will take you to the source location where the object is used and shows the transition information.

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 expansion button to bring up the Timeline view. If you did not have "Transitions' set when the application was run, modify the collectors (See Section "Creating a New Project and Thread Checker Activity" on page 113). 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 6.5.

Figure 6.5. 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.

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 “Profiler View” tab and click on the Profile View expansion button.
2. Click on the button (shown at left margin) to remove all data grouping and show the summarized Critical Path data.
3. Drag the original Thread Profiler results of this application from the Tuning Browser to compare with the final results you achieved.
4. Highlight both bars (CTRL+click on bars not highlighted).
5. Examine the results displayed in the “Profile” and Timeline” Views.

   Is there a noticeable improvement from the original execution performance?

Review Questions

Question 1: Is oversubscription of processor resources by active threads a severe performance problem?

Yes No

Question 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?

Question 3: Was the Timeline view useful?

Question 4: Can applications instrumented for Timeline view be used in performance comparisons?

Yes No

Lab. 7: Threaded Programming Methodology

Time Required	Seventy-five minutes
Objectives	In this lab session, you will learn and practice the four steps of Threading Methodology. After successfully completing this lab's activities, you will be able to: • Apply threading on computational loops within C source code with OpenMP • Find actual and potential threading errors, and validate that the errors are fixed using Intel Thread Checker • Use Intel Thread Profiler to determine if threaded performance is achieving desired goals, and identify potential bottlenecks to performance

Activity 1: Compile and Run Serial Code

Time Required	Five minutes
Objective	• Check that serial code will compile and run; observe behavior of serial application

The application computes and saves prime numbers between the range of integers given on the command line. The algorithm takes a "brute force" approach and divides each potential prime by possible factors of that number. If one of those factors divides the number evenly, the number is not prime; if all the possible factors have been tried without any being found to divide the number evenly, the number is prime and saved within an array of primes. A running count of the number of primes found is maintained and printed upon completion of the search. A progress update of the percentage of numbers within the given range is printed as the code executes. The progress update value is changed for every 10% completed. The time taken to find the primes will also be printed.

Build and Run Serial Program

1. Locate and change to the PrimeSingle directory. You should find Visual Studio Solution and project files and a source file, PrimeSingle.cpp, in this directory.
2. Double-click on the solution icon and examine the source file.
3. Be sure the Release configuration is selected and build the executable binary PrimeSingle.exe.
4. After successfully compiling and linking, open a command window, change the directory to the Release folder holding the application, and run from the command line:

```
>> PrimeSingle.exe 1 2000
```

5. Try several different ranges of numbers as command line arguments to the application.

Activity 2: Analyze Serial Code

Time Required	Ten minutes
Objective	• Run serial code through VTune Performance Analyzer to locate portions of the application that use the most computation time. These will be the parts of the code that are the best candidates for threading to make a positive performance impact.

Get Baseline Timing

1. Run the serial application to find the primes between 1 and 5,000,000:

```
>> PrimeSingle.exe 1 5000000
```

What time is reported form the application? ________________________ seconds.

Run VTune Performance Analyzer on Application

1. Switch to the Debug configuration within Visual Studio.
2. Rebuild the application.
3. Start VTune Performance Analyzer and create a new Sampling activity with the debug version of PrimeSingle application to be launched for analysis. Use '1 1000000' as the command line arguments.
4. Run the analysis.

   Which function within the code took the most time for execution (most clockticks)? _____________________________

5. If possible, create a new activity for CallGraph analysis of the same application with the same command line.

   What is the average amount of time that each call of the above function takes? ________________usec

Activity 3: Run OpenMP Code

Time Required	Five minutes
Objective	• Build and run OpenMP threaded version of prime finding application

Build and Run Threaded Program

1. Locate and change to the PrimeOpenMP directory. You should find Visual Studio Solution and project files and a source file, PrimeOpenMP.cpp, in this directory.
2. Using the Debug configuration, build the source file into the executable binary PrimeOpenMP.exe.
3. After successfully compiling and linking, change directories in the command window to the PrimeOpenMP\Debug folder holding the threaded application, and run from the command line:

```
>> PrimeOpenMP.exe 1 5000000
```

   What execution time is reported from the threaded application?
   ________________________ seconds.

   Compared to the serial application run time, what is the speedup of the threaded version?

   [Speedup = (Serial Time) ÷ (Parallel Time)] _________________________

Activity 4: Use Thread Checker to Check for Threading Errors

Time Required	Fifteen minutes
Objective	• Use Intel Thread Checker to find the race condition that is causing the code to compute the wrong number of primes

Use Thread Checker for Analysis

1. Launch VTune Performance Analyzer on the Windows platform. Create a new Thread Checker activity.
2. Use the debug version of the PrimeOpenMP.exe application to launch. Use '1 200' as the command line arguments. Since the instrumentation from Thread Checker expands the run time of an application, you will not want to run a full test case. Enough numbers to run through the threads will be sufficient for Thread Checker to determine where any threading errors are within the code.
3. Run the application within Thread Checker to generate a diagnostic report.
4. Scan through some of the diagnostics for the run of the application that populate the diagnostic pane in the Thread Checker display. Double-click on one or more of the diagnostic lines to find the source code lines associated with the diagnostic. You can use the tabs at the bottom of the pane to navigate between the diagnostics list and source code.

   How many different lines of code have threading errors? ___________________

Activity 5: Correct Threading Errors

Time Required	Ten minutes
Objective	• Correct the errors identified from Thread Checker

Correct Errors and Validate Results

1. Within Visual Studio, edit your source code to correct the errors found by Thread Checker.

Tip: Insert critical regions around each use (both read and write) of the affected variables with #pragma omp critical.

2. When you have changed the source code, rebuild the application.
3. Re-run the application through Thread Checker.
4. Have all the errors been corrected? If not, go back to Step 1 and repeat until you have successfully handled all the error diagnostics reported by Thread Checker.

Build and Run Threaded Program

1. Once the threading errors have been removed from the program, re-run the application within the command window:

```
>> PrimeOpenMP.exe 1 5000000
```

   What execution time is reported from the threaded application? ________________________ seconds.

   Compared to the serial application run time, what is the speedup of the corrected threaded version?

   [Speedup = (Serial Time) ÷ (Parallel Time)] _________________________

Activity 6: Find Threading Performance Issues

Time Required	Ten minutes
Objective	• Use Intel Thread Profiler to find any problems that might be hampering the parallel performance

The prime finding application needs to be rebuilt with the performance measurement source code instrumentation flag in order to identify potential performance bottlenecks resulting from how threads interact with each other. This will allow either OpenMP or explicit threads analysis can be used within Intel Thread Profiler.

Instrument and Run Threaded Program

1. Open the PrimeOpenMP project properties dialog box. Within the C/C++ folder, on the Command Line, change the /Qopenmp flag to /Qopenmp_profile. This is the source code instrumentation flag.
2. Rebuild the application with the new compile flag.
3. Create a new Thread Profiler activity. Be sure to choose the "Threaded (Windows* API or POSIX Threads)" radio button in the Threading Type selector. Run the executable with a full range of numbers (1 to 1000000 should be sufficient). A "production-sized" data set is used since this would be the data size normally used. Performance problems may not be evident if the data set used for analysis is smaller than normal.

Use Thread Profiler for Analysis

1. Once the analysis is available, click through the tabs to examine the different views.

   From the Timeline view, can you get to the transition source that is showing some thread inactivity?

Activity 7: Reduce the Number of printf Calls

Time Required	Five minutes
Objective	• Correct the problem of implicit synchronization issue from calling printf more times than necessary

Modify Code to Control Number of Progress Updates

1. Edit the PrimeOpenMP.cpp source code to only print the minimum required progress update output statements and, consequently, remove the implicit synchronization from unnecessary use of thread-safe library calls.

```
void ShowProgress( int val, int range )
{
    int percentDone;
    static int lastPercentDone = 0;
#pragma omp critical
{
       gProgress++;
       percentDone = (int)((float)gProgress/
(float)range*200.0f+0.5f);
}
    if( percentDone % 10 == 0 && lastPercentDone <
percentDone / 10){
        printf("\b\b\b\b%3d%%", percentDone);
        lastPercentDone++;
    }
}
```

2. Compile the source file into the executable binary PrimeOpenMP.exe. (Be sure the /Qopenmp_profile flag has been changed back to just /Qopenmp for compilation.) Within the command window, run the executable with an appropriate range of numbers to test.

```
>> ./PrimeOpenMP 1 5000000
```

What execution time is reported from the load balanced application?
______________________ seconds.

Compared to the serial application run time, what is the speedup of the current threaded version?

[Speedup = (Serial Time) ÷ (Parallel Time)] _________________________

Update Serial Code

1. Modify the serial version of the code to use the same ShowProgress function that the OpenMP code uses. Recompile and run the serial application.

   What execution time is reported from the improved serial application?
   _____________________ seconds.

   Compared to the updated serial run time, what is the speedup of the current threaded version?

   [Speedup = (New Serial Time) ÷ (Parallel Time)] _________________________

Activity 8: Modify Critical Regions

Time Required	Ten minutes
Objective	• Modify the explicit critical regions to reduce contention and limit the amount of time threads spend in each region

Use *InterlockedIncrement* and Local Variables

1. For each critical region within the PrimeOpenMP.cpp source file, modify the code to replace the OpenMP critical pragmas by using the Windows InterlockedIncrement function.

   First modification:

```
#pragma omp parallel for

     for( int i = start; i <= end; i+= 2 ){

        if( TestForPrime(i) ) {

globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;

        }

        ShowProgress(i, range);

    }
```

   Second modification:

```
   long percentDone, localProgress;

   static int lastPercentDone = 0;

       localProgress =
InterlockedIncrement(&gProgress);

       percentDone = (int)((float)localProgress/
(float)range*200.0f+0.5f);
```

2. Rebuild the source file into the executable binary PrimeOpenMP.exe. Run the application within the command window with an appropriate range of numbers to test.

```
>> ./PrimeOpenMP 1 5000000
```

What execution time is reported from the threaded application?
____________________ seconds.

Compared to the (new) serial application run time, what is the speedup of the threaded application?

[Speedup = (Serial Time) ÷ (Parallel Time)] _______________________

Activity 9: Fix Load Imbalance

Time Required	Five minutes
Objective	• Correct the load balance issue between threads by distributing loop iterations in a better fashion

Use OpenMP schedule Clause

1. Within Visual Studio, edit your source code to remedy the load imbalance problem identified by Thread Profiler. To do this, add a schedule clause to the OpenMP parallel region pragma.

```
#pragma omp parallel for schedule(static, 8)

    for( int i = start; i <= end; i += 2 )

    {

       if( TestForPrime(i) )

globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;


        ShowProgress(i, range);

    }
```

2. Compile the source file into the executable binary PrimeOpenMP.exe and run the executable with an appropriate range of numbers to test.

```
>> PrimeOpenMP.exe 1 5000000
```

   What execution time is reported from the load balanced application? ______________________ seconds.

3. Build the Release configuration of the threaded application and run.

   What is the exection time? ______________________ seconds.

   What execution time is reported from the Release build application? ______________________ seconds.

Compared to the serial application run time, what is the final speedup of the threaded version?

[Speedup = (Serial Time) ÷ (Parallel Time)] ________________________

Lab. 8: Scalability of Threaded Applications

Workload-dependent Scaling - An Overview

Activity 1, comprising four parts, discusses an example of workload dependent scaling with a multi-threaded application. Scaling of multithreaded applications may vary based on workloads. Workloads with balanced data set division among the threads and with larger data set may scale well. On the other hand, workloads with imbalanced division or smaller data sets may show poor scaling. Studying workload statistics is one of the important areas to consider when multi-threading an application.

While root causing poor scaling of a multithreaded application, it is recommended that different sizes of workloads be run to determine whether scaling performance changes with a different workload type/size. Another recommended step is to use Intel® Thread Profiler in identifying serial vs. parallel time.

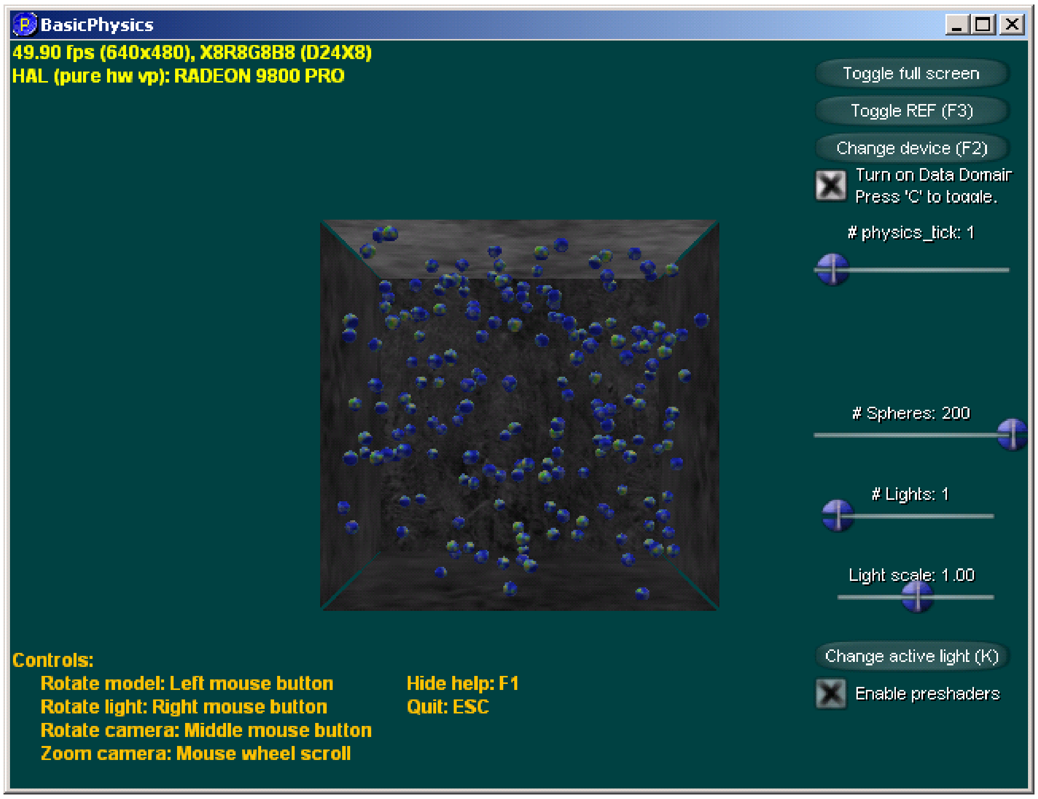
In this four-part activity, you will examine a sample application which shows a difference in multithreaded performance improvement due to the difference in size of workloads.

Introduction to Sample Application

The sample application used here is a DirectX application based on the BasicHLSL sample included in DX SDK.

This example demonstrates a number of spheres in a box with an implementation of collision detection mechanisms for sphere-sphere and sphere-plane based on Newtonian physics. The GUI of the BasicPhysics demo (see Figure 8.1) provides sliders to add or remove spheres from the scene and change the number of times the physics computations are done before actually rendering.

The GUI displays the frame rate at the top left corner. Besides the GUI, there is a checkbox to turn on threading for data decomposition. When unchecked, all computation and rendering is done by a single thread; when checked, the rendering is done on the main thread and the position computations for the sphere objects are divided among two worker threads.



BasicPhysics

49.90 fps (640x480), X8R8G8B8 (D24X8)
HAL (pure hw vp): RADEON 9800 PRO

Controls:
Rotate model: Left mouse button
Rotate light: Right mouse button
Rotate camera: Middle mouse button
Zoom camera: Mouse wheel scroll
Hide help: F1
Quit: ESC

[✓ Turn on Data Domain] Press 'C' to toggle.
# physics_tick: 1
Spheres: 200
Lights: 1
Light scale: 1.00
[✓ Enable preshaders]

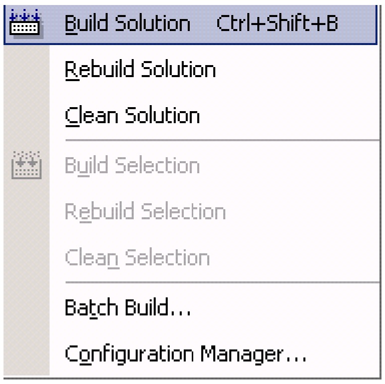
Figure 8.1. BasicPhysics Application's User Interface

While multi-threading an application, it is important to study characteristics of typical workloads. If the workload size is too small, multi-threading may cause an extra overhead. Another factor is to maintain a good balance of work among all threads.

Activity 1a: Build the Application

1. Locate the 'BasicPhysics.sln' solution file in the Scalability\Workload Scaling Lab directory.
2. Double-click the icon to launch Visual Studio .NET 2005 with this solution loaded.
3. Use Ctrl+Shift+B or the Build menu to build the application after the .sln file is opened.

Figure 8.2. The Build Menu



4. Run the application using Ctrl + F5.

Note: The application may take some time to launch, as it loads and initializes meshes for all spheres.

Activity 1b: Characterize Multithreaded Implementation with Large Data Set

1. Set 'physics_tick' = 10 and '# Spheres' = 200 by moving sliders to right.
2. Uncheck 'Turn On Data Domain Decomposition' box. Once it has stabilized, note the frame rate (in space below) displayed at the top left corner of the window.
3. Check 'Turn On Data Domain Decomposition' box and note frame rate displayed at the top left corner of the window.
4. Compute Multi-Threaded scaling of the application with the large data set.

ST Frame Rate with 200 spheres and physics_tick 10: ______________FPS

MT Frame Rate with 200 spheres and physics_tick 10: _____________ FPS

MT Speedup: ___________

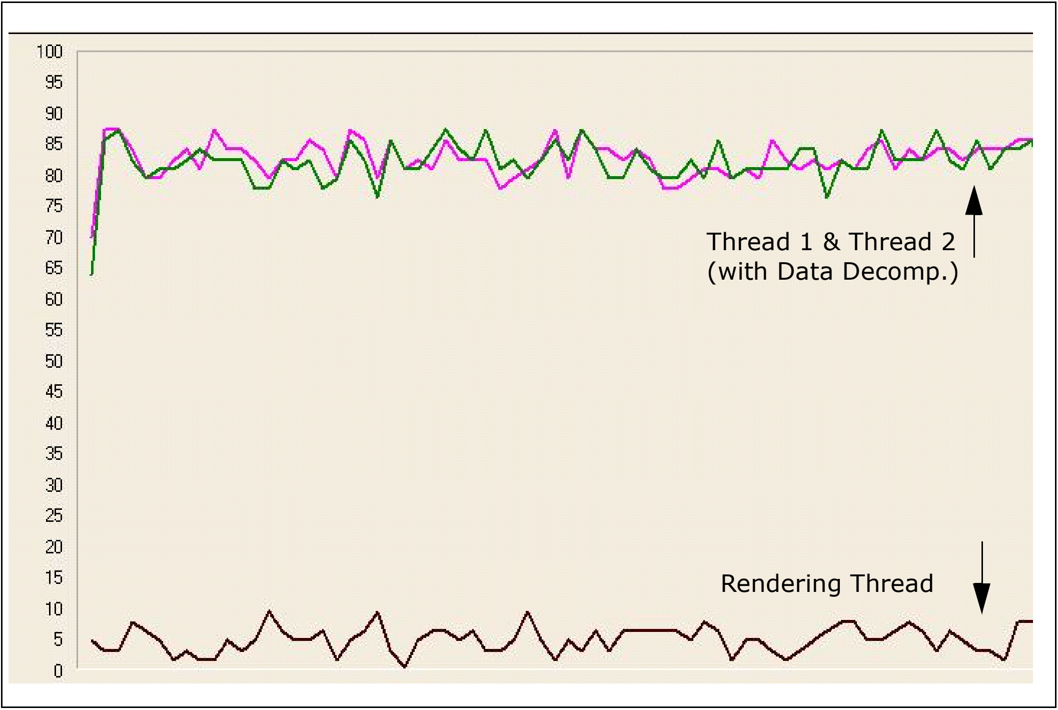
Activity 1c: Examine Thread Performance with Large Data Set

This activity should be run while the multithreaded version of the BasicPhysics application is running with 200 spheres.

1. From a command line window, launch the PerfMon performance monitor while the BasicPhysics application is running.
2. Click on the "Add" button (or use Ctrl-I).
3. Within the "Add Counters" window, select "Threads" from the Performance Object pull-down menu.
4. Be sure the "% Processor Time" counter is selected and add each of the BasicPhysics threads from the Instances pane. There should be three threads from BasicPhysics.
5. Click the Close button on the "Add Counters" window.
6. Remove the other counters that were active when PerfMon was started in order to focus on the BasicPhysics threads.

   After some execution time has passed, you should see something similar to the graphing pane excerpt shown in Figure 8.3.

Figure 8.3. Threads time with 200 spheres and physics_tick 10



Activity 1d: Characterize Multithreaded Implementation with Small Data Set

1. While still running the BasicPhysics application, bring the GUI window to the front.
2. Set 'physics_tick' = 5 and '# Spheres' = 100 by moving sliders to right.
3. Uncheck 'Turn On Data Domain Decomposition' box. Once it has stabilized, note the frame rate (in space below) displayed at the top left corner of the window.
4. Check 'Turn On Data Domain Decomposition' box and note frame rate displayed at the top left corner of the window.
5. Compute Multi-Threaded scaling of the application for 100 spheres.

ST Frame Rate with 100 spheres and physics_tick 5: ______________FPS

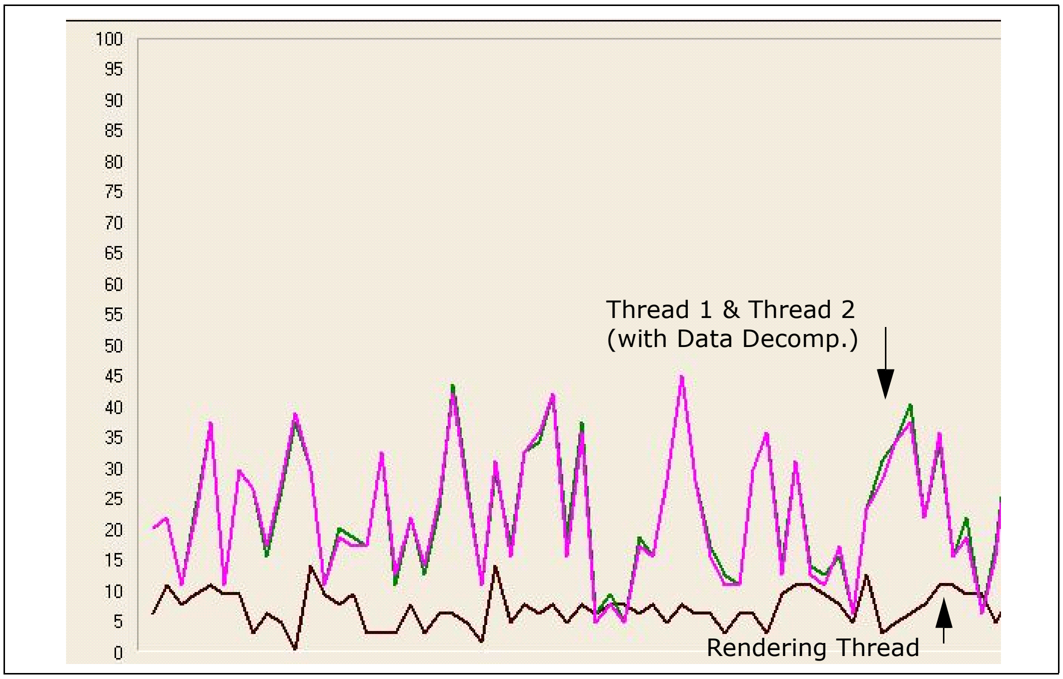
MT Frame Rate with 100 spheres and physics_tick 5: _____________ FPS

MT Speedup: ___________

6. While running the BasicPhysics application in multithreaded mode, bring the PerfMon window to the front.

   You should see something similar to the graphing pane excerpt shown in Figure 8.4.

Figure 8.4. Threads time with 100 spheres and physics_tick 5



.

Comparing your calculated speedup results from Activities 1c and 1d, it should be clear that better scaling is possible when time spent in the parallel (threaded) execution is much larger than time spent in serial section. PerfMon results in Figure 2 (workload of 200 spheres, physics_tick=10) and Figure 3 (workload of 100 spheres, physics_tick=5) show the relationship of parallel to serial time during execution.

Activity 2: Illustrate Effects of Load Balance in Multithreaded Implementation

1. While still running the BasicPhysics application, bring the GUI window to the front.
2. Set 'physics_tick' = 10 and '# Spheres' = 200 by moving sliders to right. Also, ensure that the 'Turn On Data Domain Decomposition' box is checked.
3. The "Balance pct." slider should have been pre-set to 50. This divides the position computations evenly between the two worker threads. Move the slider to the left to a position close to 33. This will assign twice as many objects to the second thread than are assigned to the first thread.
4. Note the frame rate below:

MT Frame Rate with 200 spheres, physics_tick 10, and Balance pct. 30:
________________FPS

Question 1: Is there any difference in the graphs of the two worker threads within the PerfMon graphing pane?

5. Move the "Balance pct." Slider to the left to a position close to 10.
   Computation for nine times as many objects will be assigned to the second thread in this case.
6. Note the frame rate below:

MT Frame Rate with 200 spheres, physics_tick 10, and Balance pct. 10:
________________FPS

Question 2: How do the graphs of the two worker threads differ under this load assignment as shown by PerfMon?

Question 3: How do the threaded frame rates of the imbalanced work loads compare to the serial and balanced multithreaded workloads?

Activity 3: Measuring Synchronization Object Overhead

Objectives	• Measure the relative speed of different synchronization objects in an uncontended fashion

1. Locate the Scalability/SyncObjectBenchmark folder and double-click on the Visual Studio solution icon. Examine the code to understand what is being timed in the application. Rebuild the project.
2. From the Debug menu, run the executable by choosing the "Start Without Debugging" command.
3. Fill in the relevant entries in the table provided below, from the output generated.

File	Time	Relative Speed
InterlockedIncrement		**1.0X**
Critical Section		
Critical Section with Spincount		
Mutex		
Semaphore		

Question 1: Is there any difference in the relative speeds if the number of loops, each synchronization object is tested on, is changed? Why or why not?

Question 2: Does a different spin count change the relative speed of the use of the Critical Section object?

Activity 4: Measuring Front-side Bus Saturation

Objectives	• Observe impact of FSB saturation on scalability using Stream benchmark • Learn use of appropriate VTune performance event to monitor bus utilization

This lab is designed for a dual socket multi-core platform with dual independent buses. You will build and run the Stream benchmark application on multiple cores. Stream will report execution time and a "Copy Function Rate" in MB/s. We will compare this latter measure with VTune event data that is used to compute bus utilization.

1. Locate the Scalability/Front-side Bus folder and double-click on the Visual Studio solution icon.
2. Re-build ScalabilityLab-Bus project.
   (The binary is made available at the Release directory.)
3. Launch Windows Task Manager and bring up the "Performance" tab.
4. Open a command window and change directory to the Scalability/Front-side Bus/ Release directory.
5. Run "scalabilitybus0.bat" at the command window.
   This runs Stream on Core 0 (Socket 0).
6. Note the "Copy Function" Rate and Run Time in the table provided below.
7. Run "scalabilitybus0123.bat" at the command window.
   This runs 4 Stream processes (Cores 0, 1, 2, 3 used).
8. Note the "Copy Function" Rate and Run Time in the table provided below.
9. If your system has 8 cores available, you can run the "scalabilitybus0246.bat" and the "scalabilitybus0-7.bat" command files. Note the "Copy Function" Rate and run time for each in the table provided below.

File	Copy Function Rate	Run Time
scalabilitybus0.bat		
scalabilitybus0123.bat		
scalabilitybus0246.bat		
scalabilitybus0-7.bat		

10. Start VTune and set up a new activity to track the "Bus Clockticks" (CPU_CLK_UNHALTED.BUS) and ""Bus Data Ready from the Processor" (BUS_DRDY_CLOCKS.THIS_AGENT) events and execute the "scalabilitybus0.bat" command file.

11. Compute bus bandwidth for the core and %bus utilization. Note the results in the table below.
12. Drill down to source to note high clock tick areas.
13. Run the VTune activity with "scalabilitybus0123.bat" (as well as "scalabilitybus0246.bat" and "scalabilitybus0-7.bat", if possible).
14. Compute total bus bandwidth and %bus utilization for each run. Note the results in the table provided below.

File	Total Bus Bandwidth	% bus utilization
scalabilitybus0.bat		
scalabilitybus0123.bat		
scalabilitybus0246.bat		
scalabilitybus0-7.bat		

Activity 5: Identifying False Sharing in Threaded Applications

Objectives	• Observe impact of false sharing on scalability • Learn use of appropriate VTune performance event to compare and contrast false sharing vs. no false sharing

This lab is designed for a dual socket multi-core platform with dual independent buses. You will build and run the Stream benchmark application on multiple cores. Stream will report execution time and a "Copy Function Rate" in MB/s. We will compare this latter measure with VTune event data that is used to compute bus utilization.

1. Locate the Scalability/False sharing folder and double-click on the Visual Studio solution icon. Re-build ScalabilityLab-FS project. The binary will be available in Release directory.
2. Open a command window and change directory to the Scalability/False sharing/Release directory.
3. Run "scalabilityfs-1t-fs.bat".
   This runs code with false sharing using 1 thread fixed to Core0 (Socket 0).
4. Note run time reported at table provided below.

5. Run "scalabilityfs-4t-fs.bat".
   This runs code with false sharing using 4 threads with one thread fixed to each core.
6. Note run time reported at table provided below.
7. Use the single core run time to compute speedup.

8. Run "scalabilityfs-4t-nofs.bat".
   This runs code with no false sharing using 4 threads with one thread fixed to each core.
9. Note run time reported at table provided below.
10. Use the single core run time to compute speedup.

File	Run Time	Speedup
scalabilityfs-1t-fs.bat		**1.0X**
scalabilityfs-4t-fs.bat		
scalabilityfs-4t-nofs.bat		

11. Launch the VTune Performance Analyzer. Create an activity to include the following events:

- Clockticks .... CPU_CLK_UNHALTED.CORE
- Instructions Retired .... INST_RETIRED.ANY
- Memory Order Machine Clear .... MACHINE_NUKES.MEM_ORDER
- 2nd Level Cache Read Misses .... MEM_LOAD_RETIRED.L2_MISS
- Bus Data Ready From the Processor .... BUS_DRDY_CLOCKS.THIS_AGENT

12. Run following batch files within VTune using the events listed above. Use a new event for each run. Fill in the table provided below from the total event counts for each run. Drill down to source view in each and compare locations of high numbers of event activities.
    a. scalabilityfs-1t-fs.bat
    b. scalabilityfs-4t-fs.bat
    c. scalabilityfs-4t-nofs.bat

	Sum of events on all 4 cores		
	1T-FS	**4T-FS**	**4T-NOFS**
Clockticks			
Instructions Retired			
Memory Order Machine Clear			
2nd Level Cache Read Misses			
Bus Data Ready From the Processor			

Review Questions

Question 1: How does the scaling in Activity 1b and 1d compare?

Question 2: How is multithreaded scaling affected by workload size?

Question 3: How would you identify the amount of serial and parallel time for a workload?

Question 4: What are some of the initial steps in root causing poor MT scaling with a particular workload?

Answers

1. Activity 1 demonstrates higher potential for scaling, as workload size is larger, hence more time spent in parallel sections.
2. In case of smaller size workloads, effective work done by threads may be less as compared to larger data set. Also if the data set is too small, overhead for maintaining threads may potentially be higher than gain achieved by multi-threading.
3. Run Perfmon/Intel® Thread Profiler monitoring %processor time, %user time and elapsed time. This will give an estimate of serial vs. parallel time for that workload. (Parallel time: When multiple threads are active concurrently. Serial time: When only 1 thread is active at a given time).
4. Collect Intel® Thread Profiler/Perfmon data to understand serial vs. parallel time spent in workload.
   Characterize various workloads with same application to examine if MT scaling changes with workloads.
   Analyze MT implementation to identify any bottlenecks (tools like Intel® Thread Profiler may be helpful).

Appendices

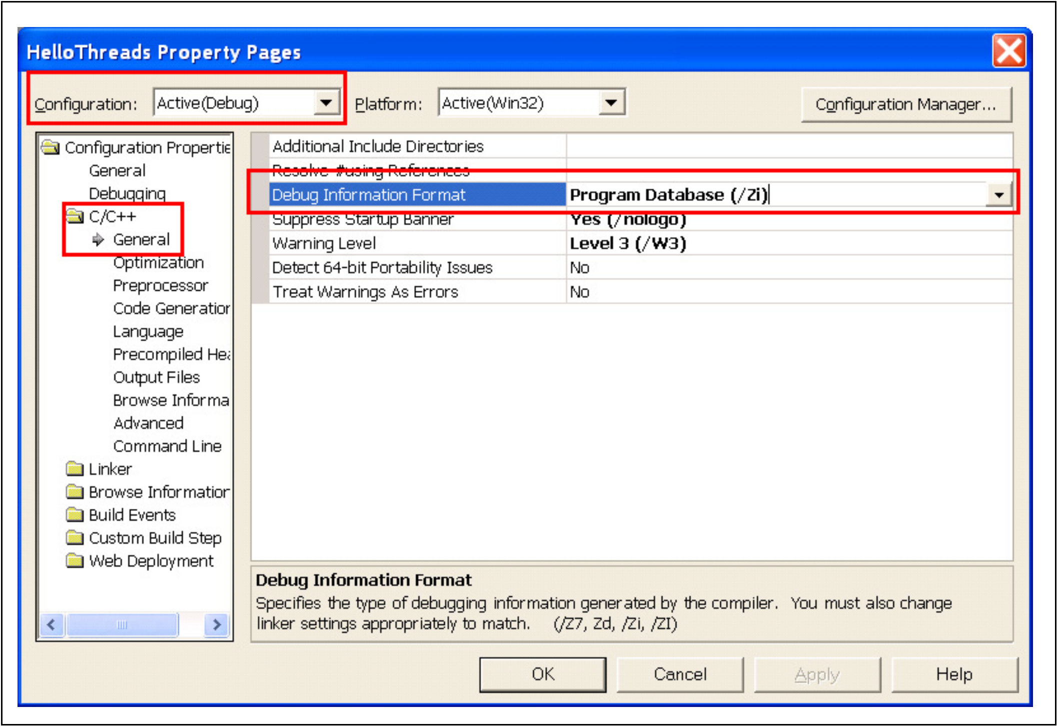
Appendix A: Setting-up Intel® Thread Checker

Setting Compile & Link Options

In order to compile applications for use with Thread Checker, certain debug features, optimization levels, and system library choices need to be made. This section contains the steps that detail the setting that should be used and how to check and set them within a Microsoft Visual Studio project configuration.

1. Ensure that the Debug options are selected, as shown in Figure A-1.

Figure A-1. Project Setting – C/C++ Folder – Debug Options


HelloThreads Property Pages

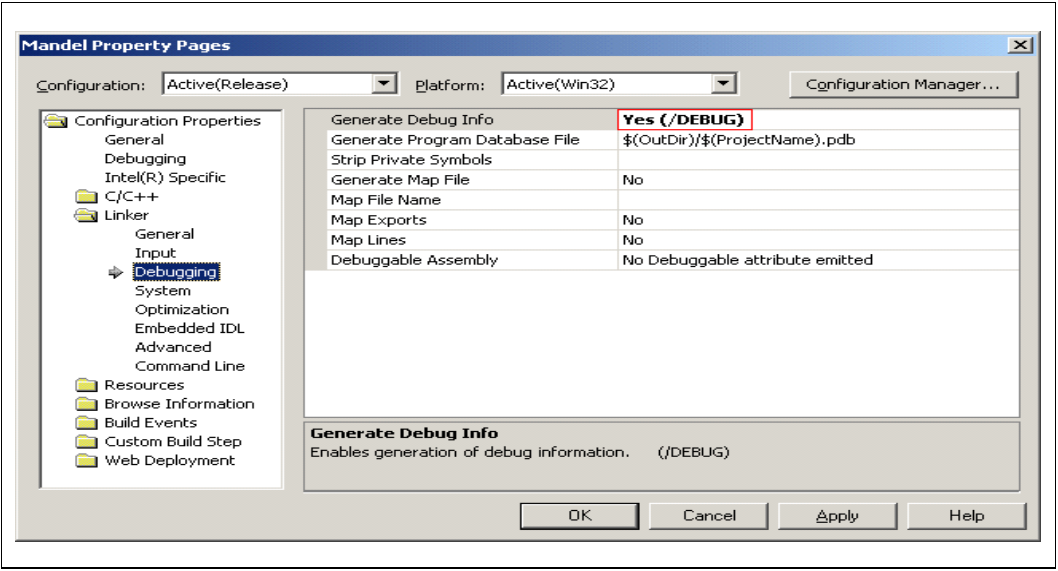
Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

Additional Include Directories	
~~Resolve #using References~~	
Debug Information Format	Program Database (/Zi)
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level 3 (/W3)
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No

Debug Information Format
Specifies the type of debugging information generated by the compiler. You must also change linker settings appropriately to match. (/Z7, Zd, /Zi, /ZI)

2. Ensure that the Debug symbols are preserved during the link phase, as shown in Figure A-2.

Figure A-2. Linker Settings – Debugging Folder



Mandel Property Pages

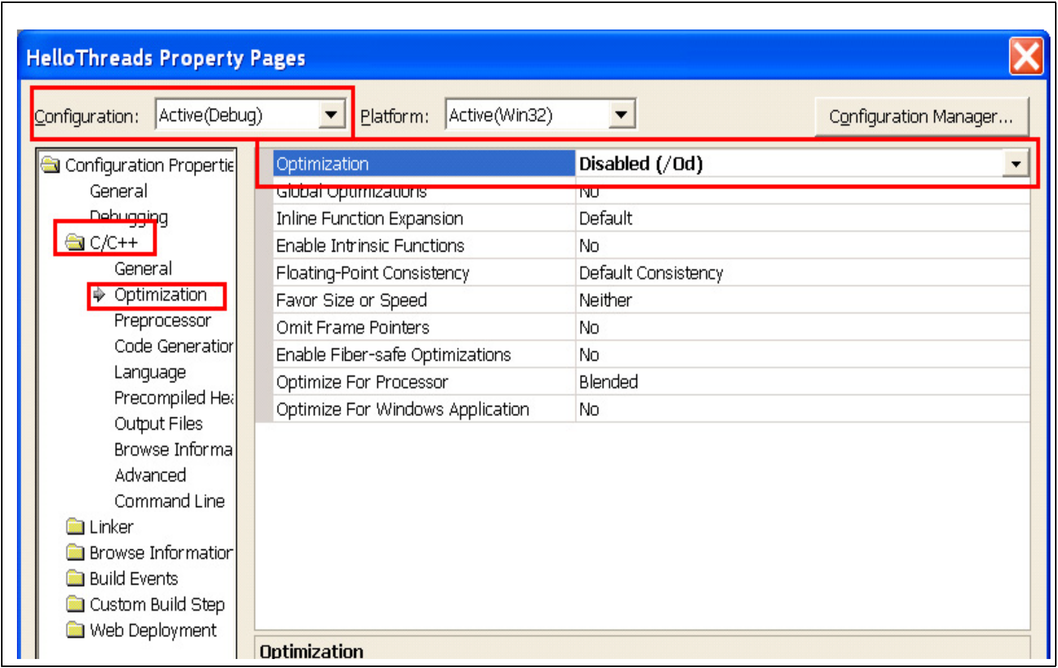
Configuration: Active(Release) Platform: Active(Win32) Configuration Manager...

Generate Debug Info	Yes (/DEBUG)
Generate Program Database File	$(OutDir)/$(ProjectName).pdb
Strip Private Symbols	
Generate Map File	No
Map File Name	
Map Exports	No
Map Lines	No
Debuggable Assembly	No Debuggable attribute emitted

Generate Debug Info
Enables generation of debug information. (/DEBUG)

3. Ensure that the Optimization is disabled, as shown in Figure A-3.

Figure A-3. Project Settings – C/C++ Folder – Optimization Options



HelloThreads Property Pages

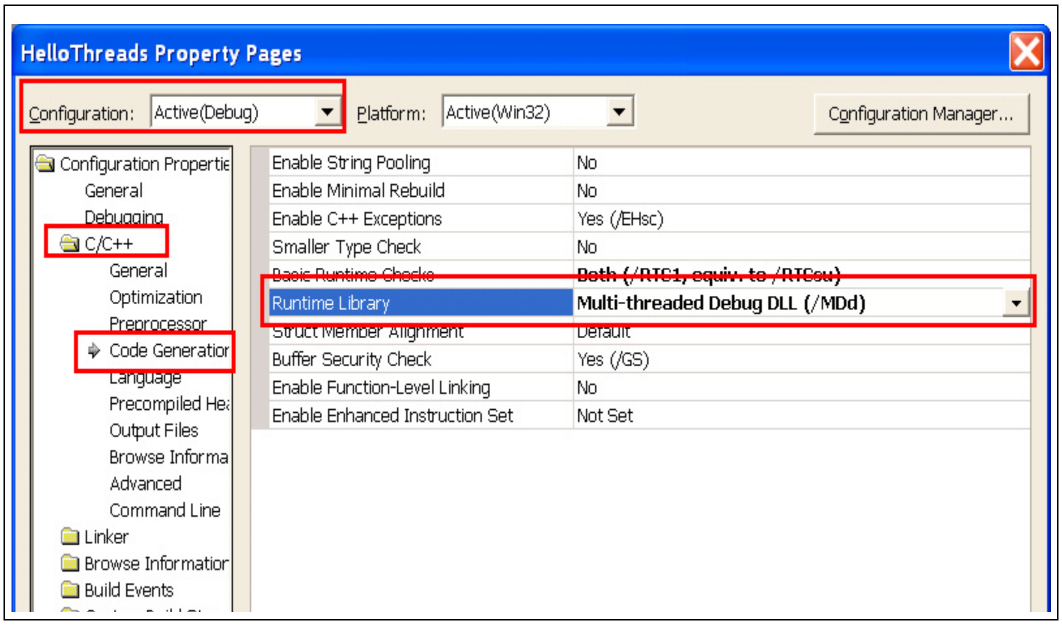
Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

Optimization	Disabled (/Od)
Global Optimizations	No
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Floating-Point Consistency	Default Consistency
Favor Size or Speed	Neither
Omit Frame Pointers	No
Enable Fiber-safe Optimizations	No
Optimize For Processor	Blended
Optimize For Windows Application	No

Optimization

4. Ensure that the thread-safe libraries are selected, as shown in Figure A-4.

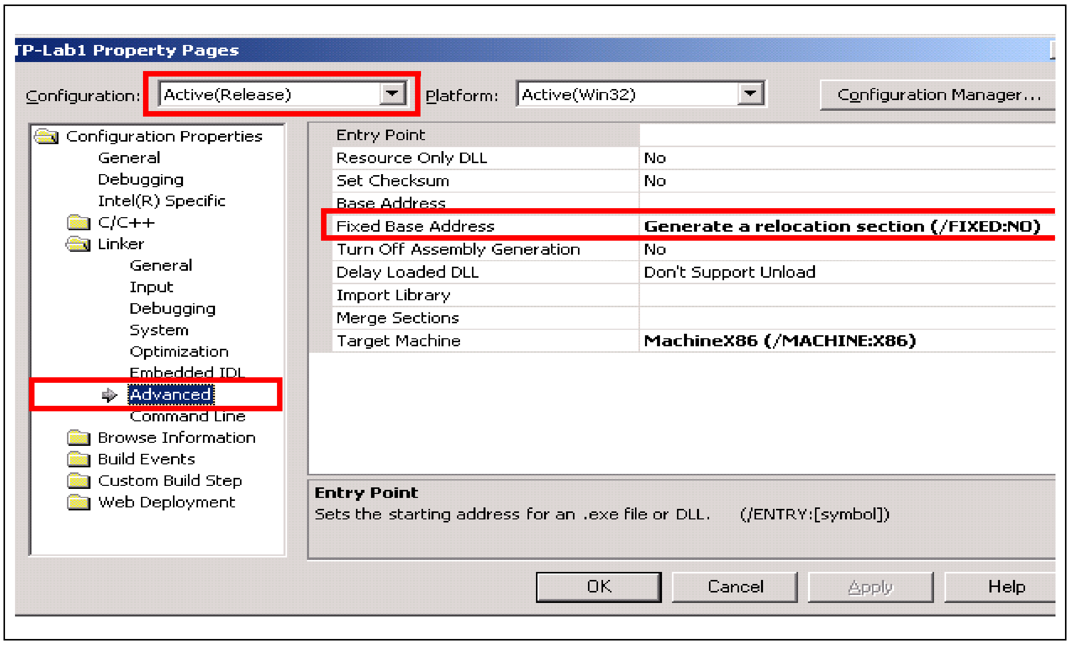
Figure A-4. Project Settings – C/C++ Folder – Thread-safe Libraries Options



Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu)
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Buffer Security Check	Yes (/GS)
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

5. Ensure that the application is built with the /fixed:no option from the Linker->Advanced attribute as shown in Figure A-4.

Figure A-5. Linker Settings – Advanced Attributes



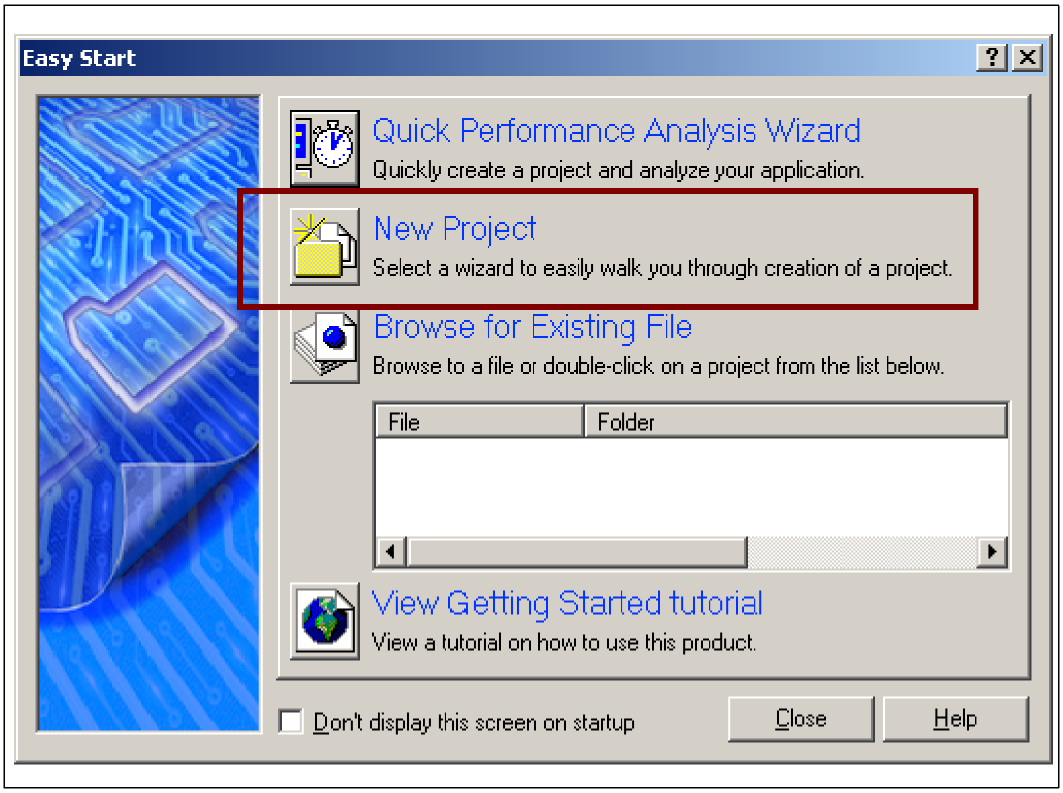
Entry Point	
Resource Only DLL	No
Set Checksum	No
Base Address	
Fixed Base Address	Generate a relocation section (/FIXED:NO)
Turn Off Assembly Generation	No
Delay Loaded DLL	Don't Support Unload
Import Library	
Merge Sections	
Target Machine	MachineX86 (/MACHINE:X86)

Entry Point
Sets the starting address for an .exe file or DLL. (/ENTRY:[symbol])

Creating a New Project and Thread Checker Activity

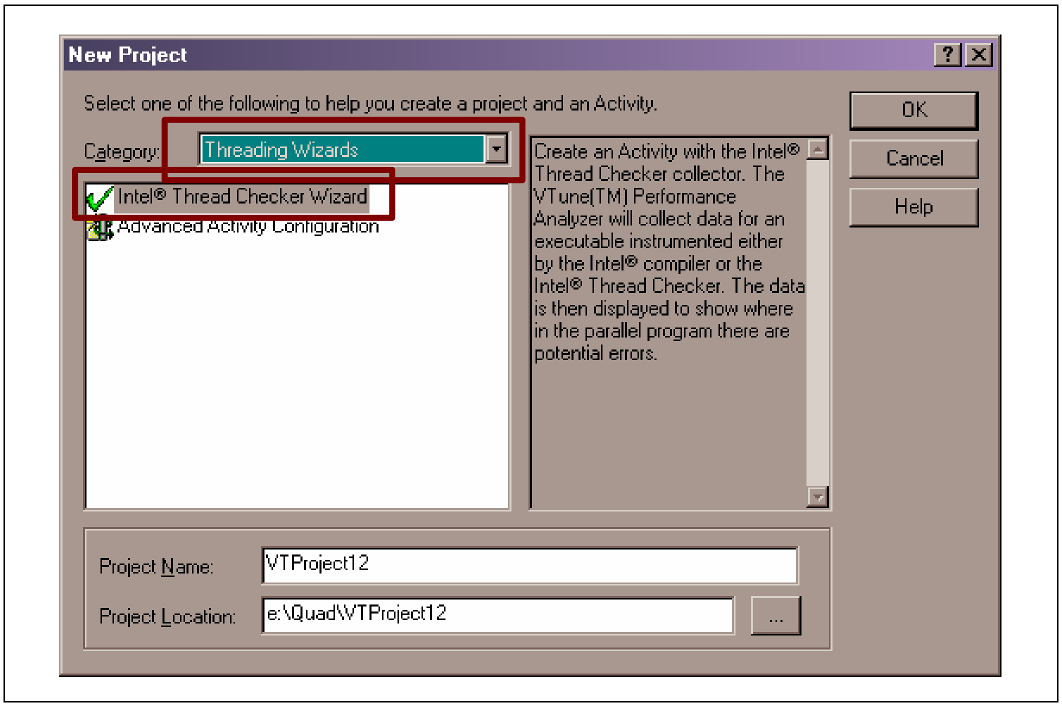
1. Start VTune Performance Analyzer and select New Project, as shown in Figure A-6.

Figure A-6. VTune™ Analyzer Easy Start Menu


Easy Start
Quick Performance Analysis Wizard
Quickly create a project and analyze your application.
New Project
Select a wizard to easily walk you through creation of a project.
Browse for Existing File
Browse to a file or double-click on a project from the list below.
File
Folder
View Getting Started tutorial
View a tutorial on how to use this product.
Don't display this screen on startup
Close
Help

2. From the Category Threading Wizards, select the Intel® Thread Checker Wizard, as shown in Figure A-7.

Figure A-7. Intel® Thread Checker New Project Menu



New Project

Select one of the following to help you create a project and an Activity.

Category: Threading Wizards

Intel® Thread Checker Wizard
Advanced Activity Configuration

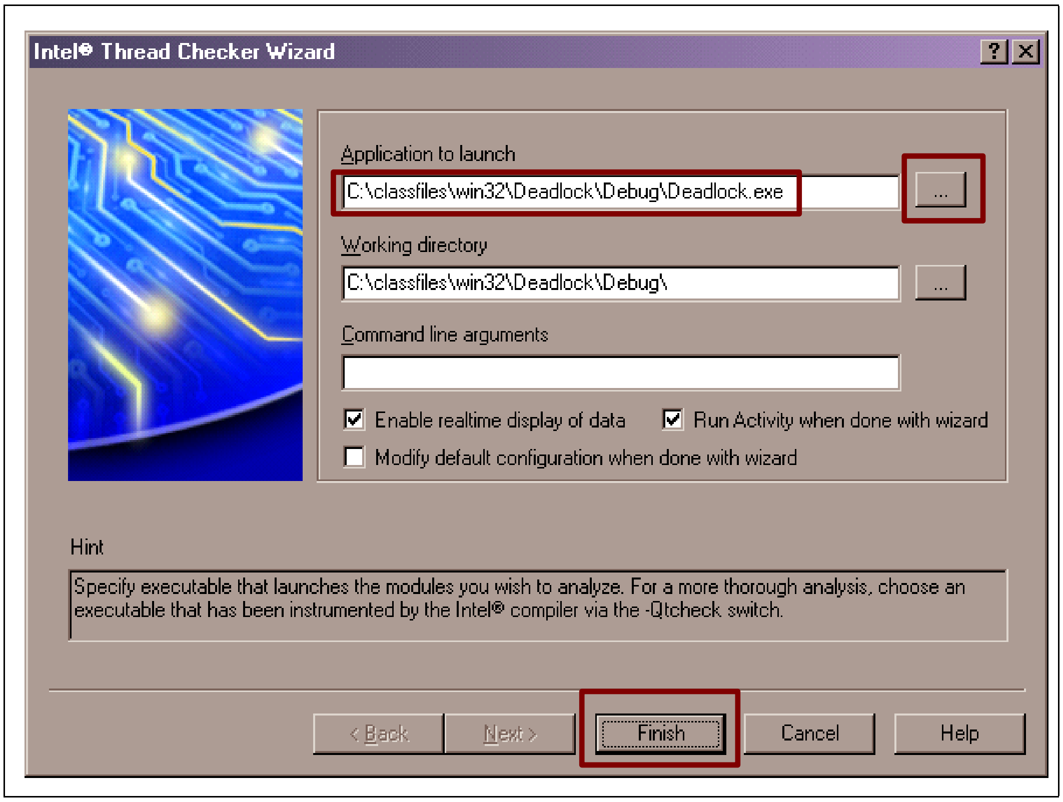
Create an Activity with the Intel® Thread Checker collector. The VTune(TM) Performance Analyzer will collect data for an executable instrumented either by the Intel® compiler or the Intel® Thread Checker. The data is then displayed to show where in the parallel program there are potential errors.

Project Name: VTProject12

Project Location: e:\Quad\VTProject12

3. Set up the application to be run by using the browse "..." button (highlighted in red in Figure A-8.) and then click Finish to start Intel Thread Checker.

Figure A-8. Intel® Thread Checker Wizard



Intel® Thread Checker Wizard

Application to launch: C:\classfiles\win32\Deadlock\Debug\Deadlock.exe

Working directory: C:\classfiles\win32\Deadlock\Debug\

Command line arguments:

[✓] Enable realtime display of data
[✓] Run Activity when done with wizard
[ ] Modify default configuration when done with wizard

Hint

Specify executable that launches the modules you wish to analyze. For a more thorough analysis, choose an executable that has been instrumented by the Intel® compiler via the -Qtcheck switch.

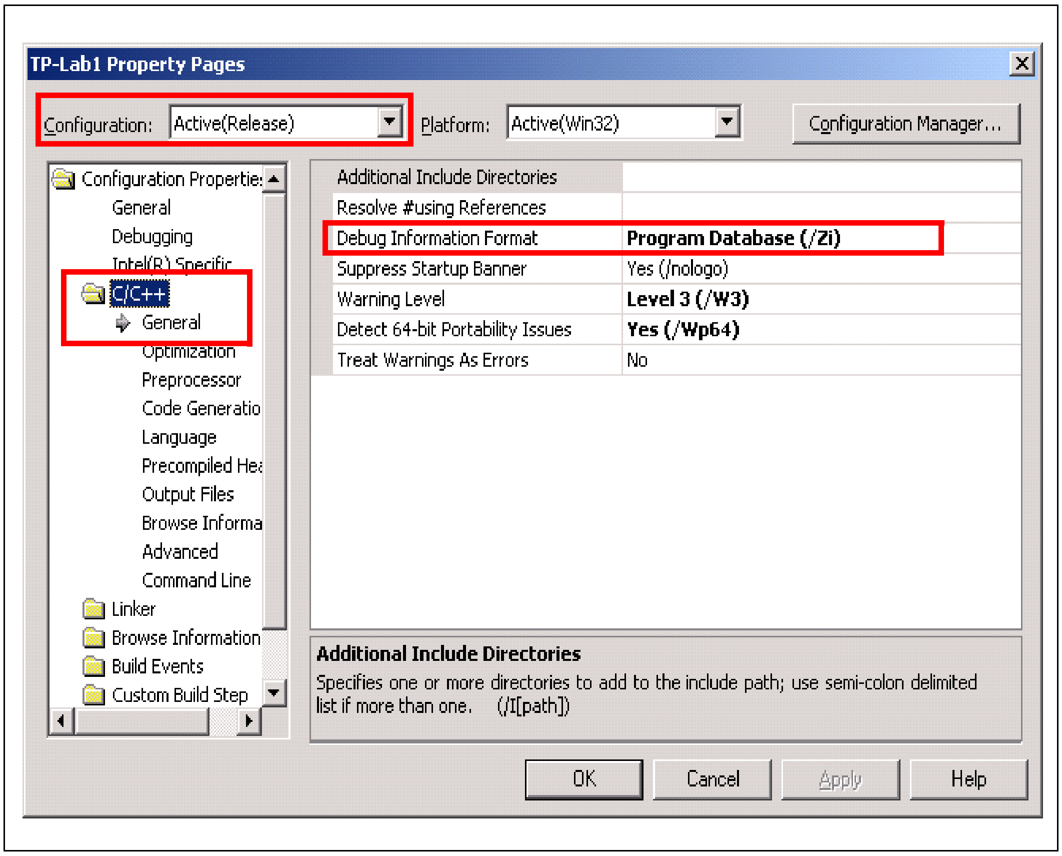
Appendix B: Setting-up Intel® Thread Profiler

Setting Compile & Link Options

In order to compile applications for use with Intel® Thread Profiler, certain debug features and system library choices need to be made. The steps in this section detail the setting that should be used and how to check and set them within a Microsoft Visual Studio project configuration.

1. From the Build menu, select Configuration Manager… and then select the Release build.
2. Make sure that Debug format is specified as shown in Figure B-1.

Figure B-1. Specifying Debug Format



TP-Lab1 Property Pages

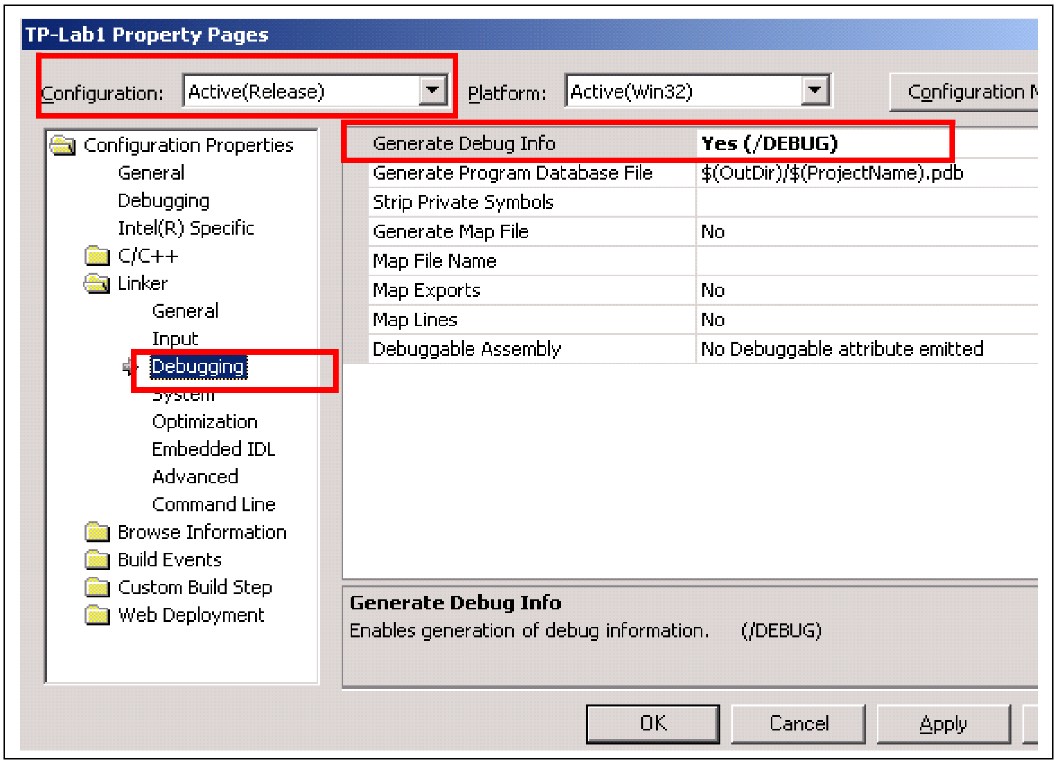
Configuration: Active(Release)
Platform: Active(Win32)

Additional Include Directories	
Resolve #using References	
Debug Information Format	**Program Database (/Zi)**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	**Yes (/Wp64)**
Treat Warnings As Errors	No

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

3. Make sure that the debug symbols are generated for the application as shown in Figure B-2.

Figure B-2. Generating Debug Symbols

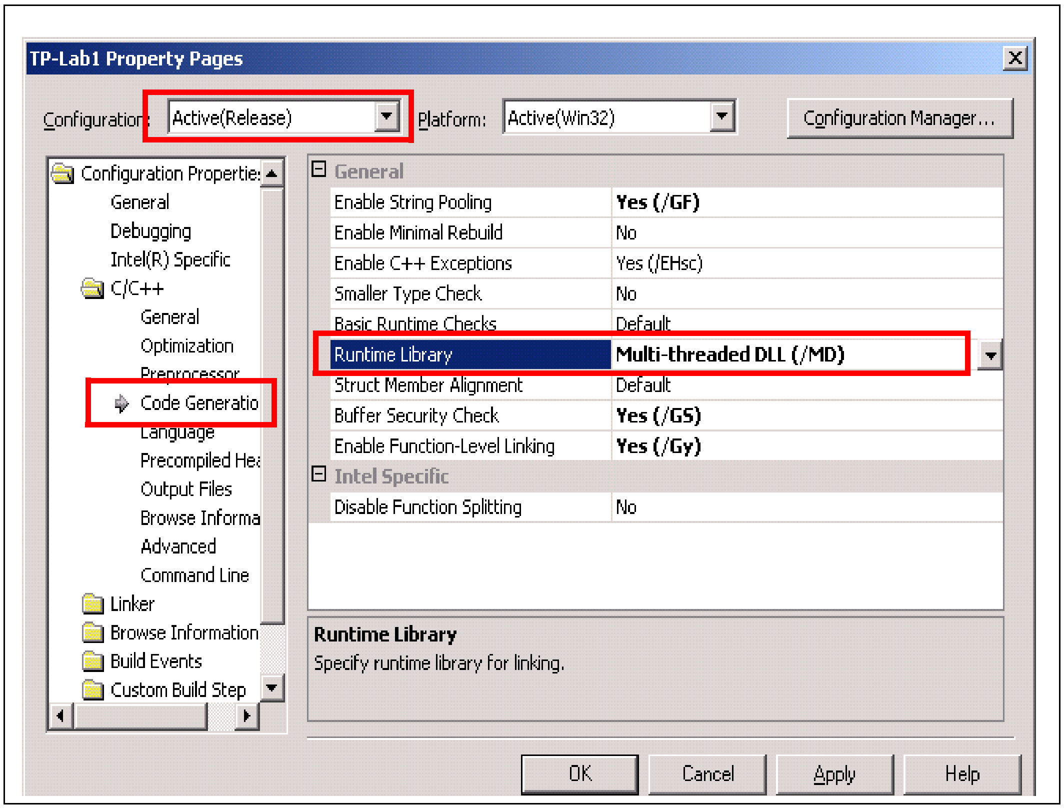


Generate Debug Info	**Yes (/DEBUG)**
Generate Program Database File	$(OutDir)/$(ProjectName).pdb
Strip Private Symbols	
Generate Map File	No
Map File Name	
Map Exports	No
Map Lines	No
Debuggable Assembly	No Debuggable attribute emitted

Generate Debug Info
Enables generation of debug information. (/DEBUG)

4. Make sure that thread safe libraries have been selected as shown in Figure B-3.

Figure B-3. Selecting Thread Safe Libraries



General

Enable String Pooling	**Yes (/GF)**
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	**Multi-threaded DLL (/MD)**
Struct Member Alignment	Default
Buffer Security Check	**Yes (/GS)**
Enable Function-Level Linking	**Yes (/Gy)**

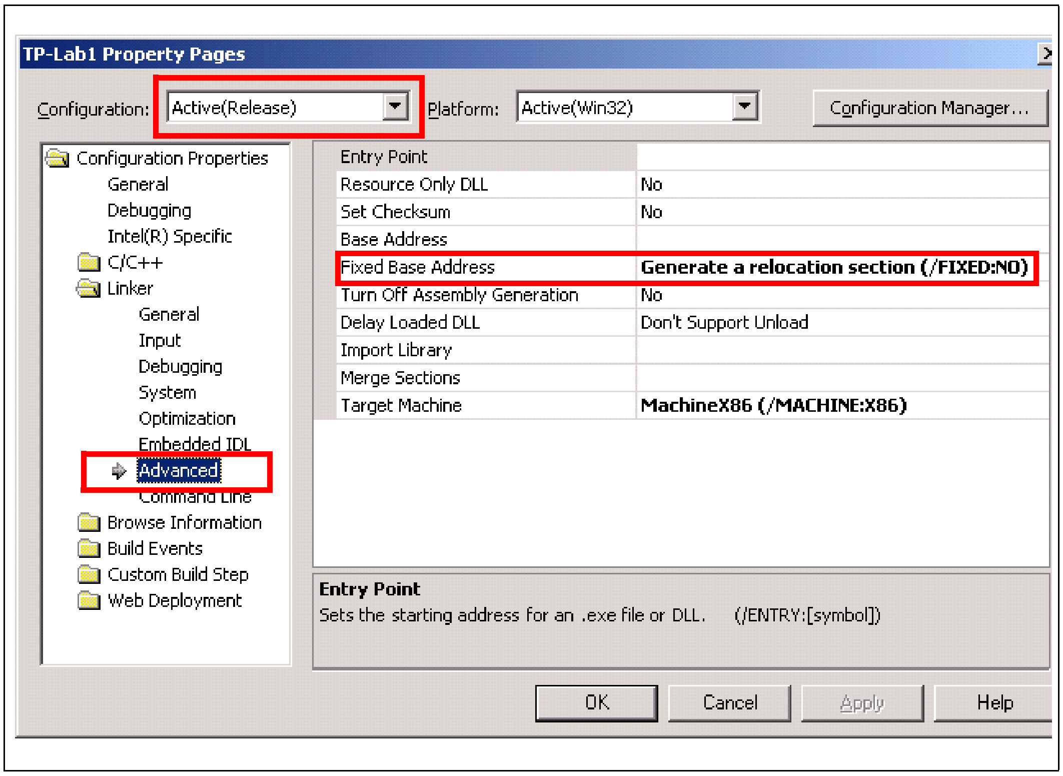
Intel Specific

Disable Function Splitting	No

Runtime Library
Specify runtime library for linking.

5. Make sure that the application is built with the **/fixed:no** option from the Linker->Advanced attribute as shown in Figure B-4.

Figure B-4. Specifying Relocatable Binary



Configuration: Active(Release)
Platform: Active(Win32)

Entry Point	
Resource Only DLL	No
Set Checksum	No
Base Address	
Fixed Base Address	**Generate a relocation section (/FIXED:NO)**
Turn Off Assembly Generation	No
Delay Loaded DLL	Don't Support Unload
Import Library	
Merge Sections	
Target Machine	**MachineX86 (/MACHINE:X86)**

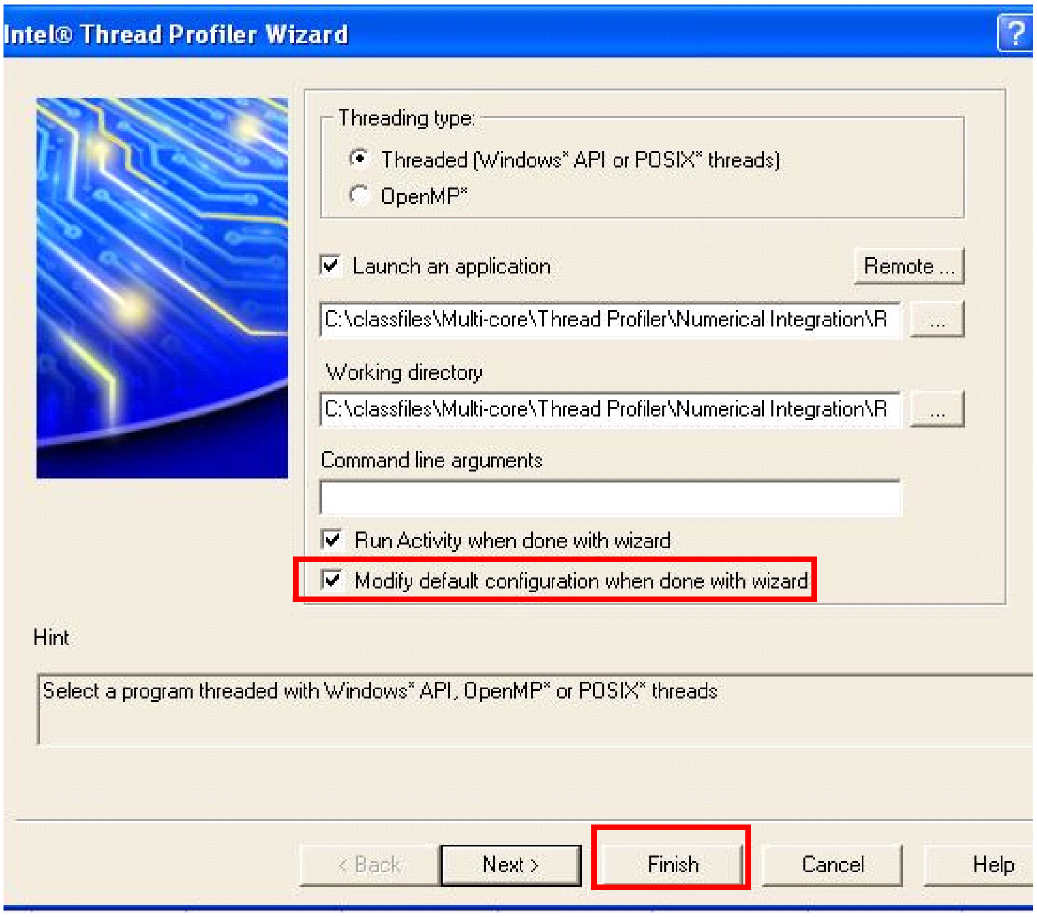
Entry Point
Sets the starting address for an .exe file or DLL. (/ENTRY:[symbol])

6. From the Build menu, select Build Solution to build. The application will now be ready to be run under the Thread Profiler.

Creating a New Project and Thread Checker Activity

1. Start VTune™ Performance Analyzer and select New Project.
2. From the Category Threading Wizards, select the Intel® Thread Profiler Wizard. (Figure B-5)
3. Within the Thread Profiler Wizard, be sure to click on the "Threaded (Windows* API or POSIX* threads)" radio button is selected.
4. Choose an application to be analyzed and enter any command line arguments required.

Figure B-5. Thread Profiler Wizard


Intel® Thread Profiler Wizard

Threading type:
[✓ Threaded (Windows* API or POSIX* threads)]
OpenMP*

[✓ Launch an application]
Remote ...

C:\classfiles\Multi-core\Thread Profiler\Numerical Integration\R

Working directory
C:\classfiles\Multi-core\Thread Profiler\Numerical Integration\R

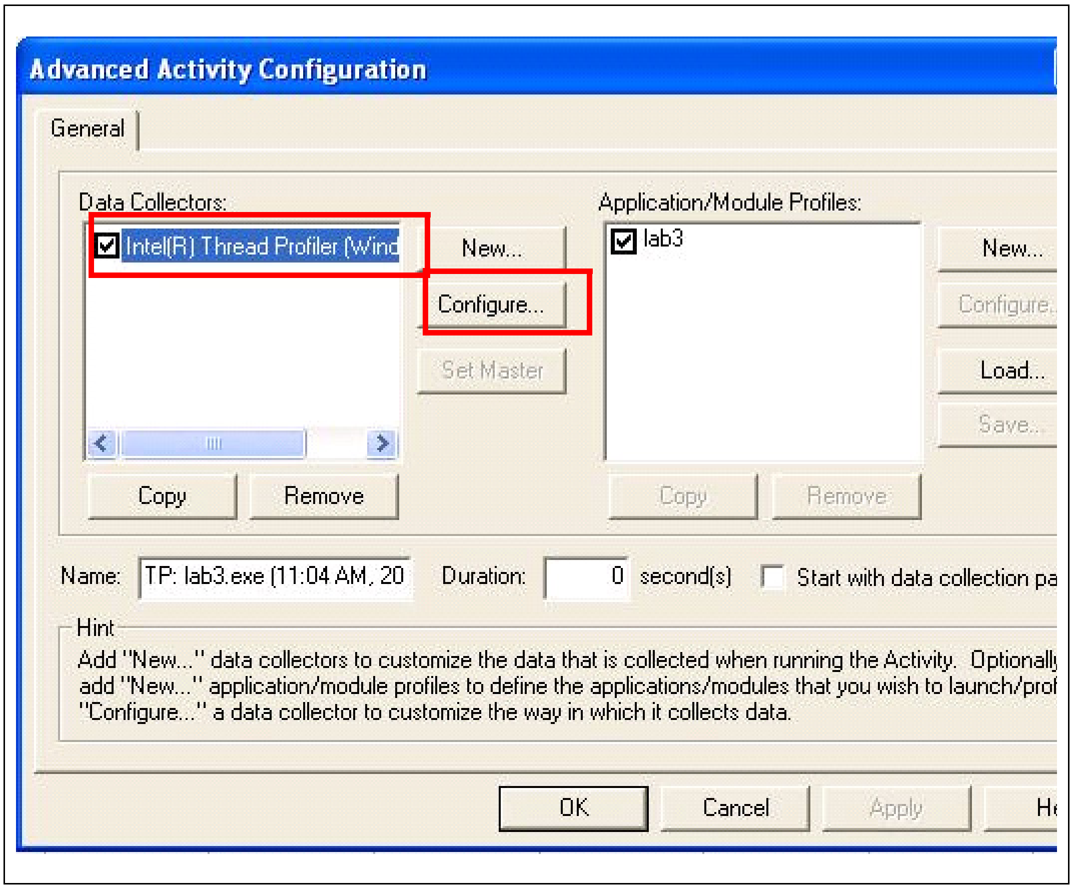
Command line arguments

[✓ Run Activity when done with wizard]
[✓ Modify default configuration when done with wizard]

Hint
Select a program threaded with Windows* API, OpenMP* or POSIX* threads

5. Select “Thread Profiler” and click on “Configure” (Figure B-6). In the resulting pop-up dialog, select the “Miscellaneous” Tab.

Figure B-6. Advanced Activity Configuration

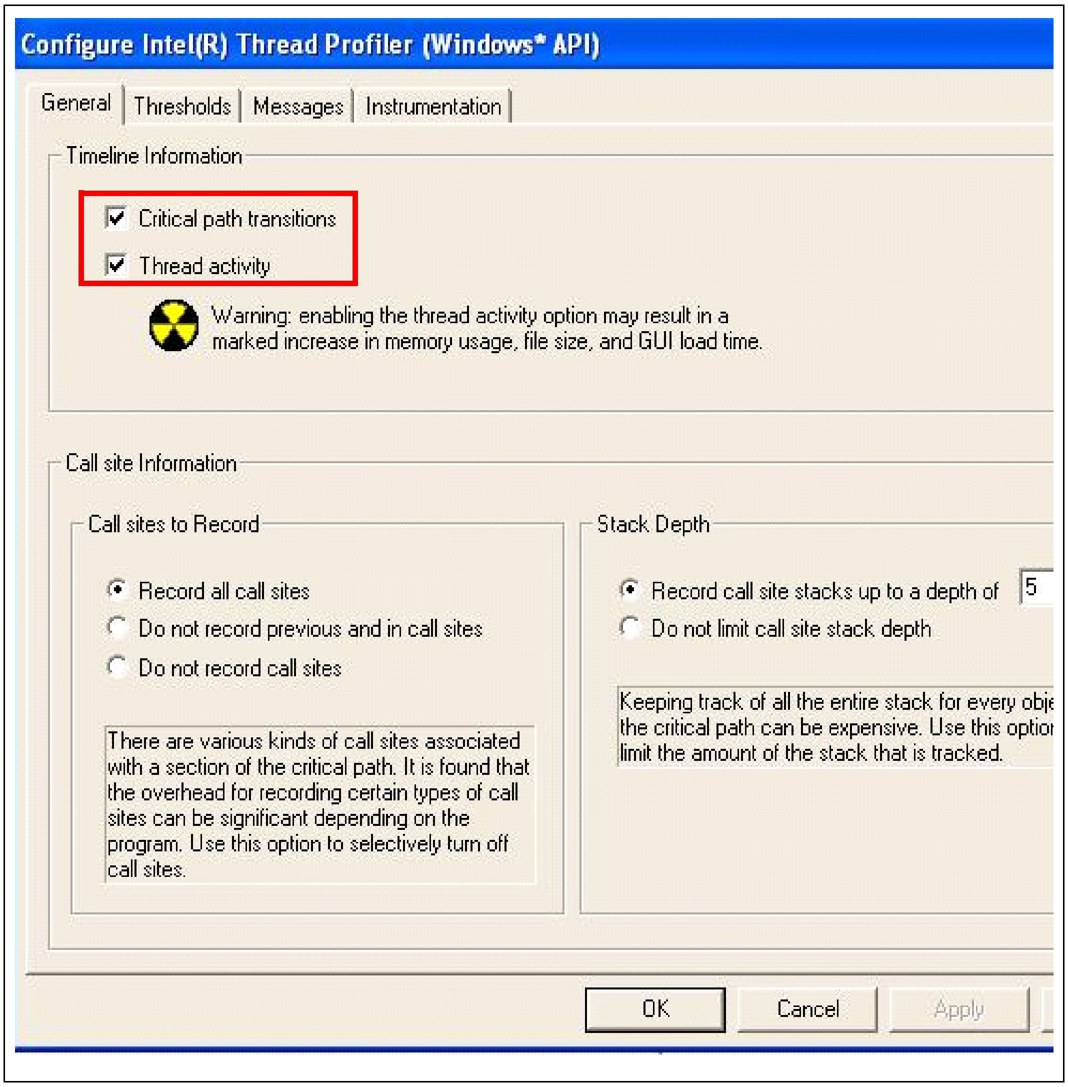


Name: TP: lab3.exe (11:04 AM, 20 Duration: 0 second(s) Start with data collection pa

Hint
Add "New..." data collectors to customize the data that is collected when running the Activity. Optionally add "New..." application/module profiles to define the applications/modules that you wish to launch/prof "Configure..." a data collector to customize the way in which it collects data.

6. Select the "Thread Activity" and "Transitions" check boxes (Figure B-7). This selection will issue a warning that the Thread Profiler may take longer to run. Accept this warning by clicking on "Yes".

Figure B-7. Specifying Timeline Information Collection Level



Configure Intel(R) Thread Profiler (Windows* API)

General | Thresholds | Messages | Instrumentation

Timeline Information

☑ Critical path transitions

☑ Thread activity

Warning: enabling the thread activity option may result in a marked increase in memory usage, file size, and GUI load time.

Call site Information

Call sites to Record

● Record all call sites

○ Do not record previous and in call sites

○ Do not record call sites

There are various kinds of call sites associated with a section of the critical path. It is found that the overhead for recording certain types of call sites can be significant depending on the program. Use this option to selectively turn off call sites.

Stack Depth

● Record call site stacks up to a depth of 5

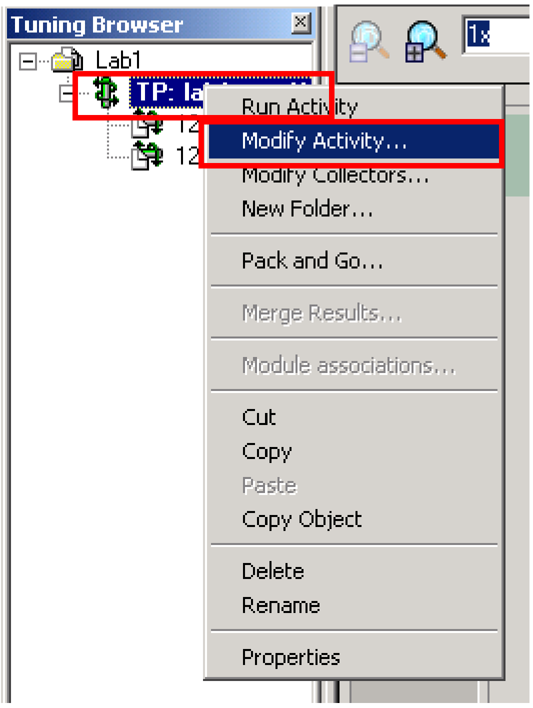
○ Do not limit call site stack depth

Keeping track of all the entire stack for every obje
the critical path can be expensive. Use this optio
limit the amount of the stack that is tracked.

Creating a New Activity by Modifying an Existing Activity

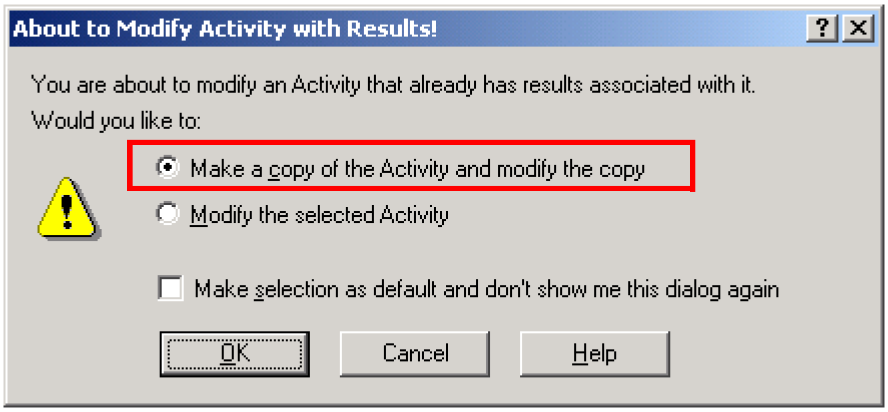
1. Right click on the current activity in the "Tuning Browser" window and select "Modify Activity". (Figure B-8)

Figure B-8. Modifying an Existing Activity



2. The About to Modify Activity pop-up dialog appears as shown in Figure B-9. Ensure that the "Make a copy of the Activity and modify the copy" radio-button is selected.

Figure B-9. About to Modify Activity dialog



About to Modify Activity with Results!

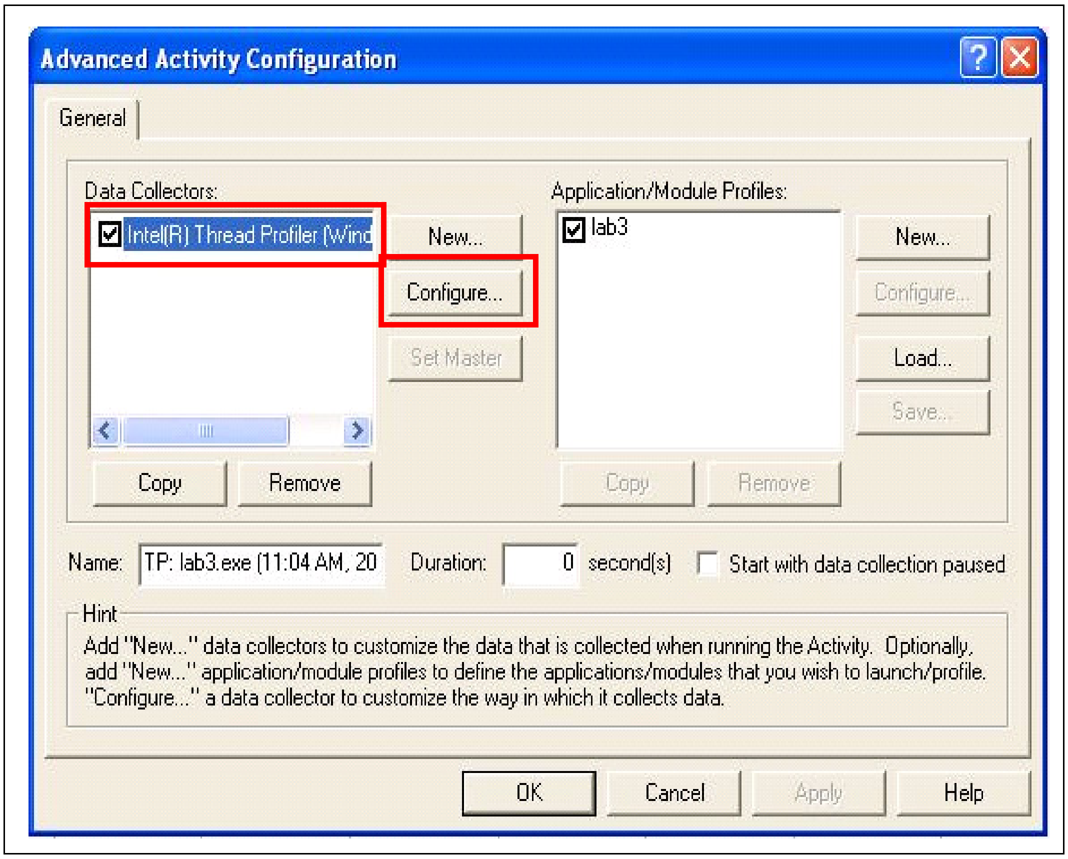
You are about to modify an Activity that already has results associated with it.
Would you like to:

[✓ Make a copy of the Activity and modify the copy]
Modify the selected Activity

Make selection as default and don't show me this dialog again

3. Select "Thread Profiler" and click on "Configure" (Figure B-10). In the resulting pop-up dialog, select the "Miscellaneous" Tab.

Figure B-10. Advanced Activity Configuration



Advanced Activity Configuration

General

Data Collectors:

☑ Intel(R) Thread Profiler (Wind

Application/Module Profiles:

☑ lab3

Name: TP: lab3.exe (11:04 AM, 20

Duration: 0 second(s)

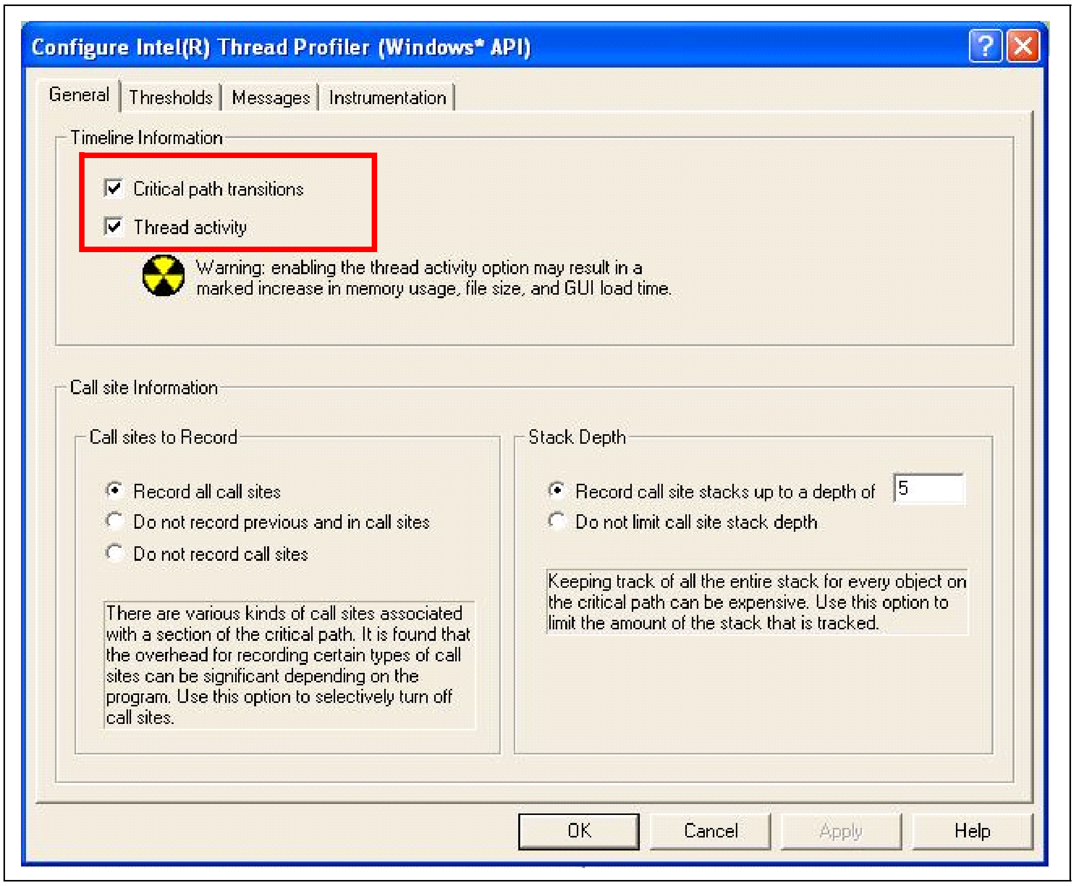
☐ Start with data collection paused

Hint

Add "New..." data collectors to customize the data that is collected when running the Activity. Optionally, add "New..." application/module profiles to define the applications/modules that you wish to launch/profile. "Configure..." a data collector to customize the way in which it collects data.

4. Select the "Thread Activity" and "Transitions" check boxes (Figure B-11). This selection will issue a warning that the Thread Profiler may take longer to run. Accept this warning by clicking on "Yes".

Figure B-11. Specifying Timeline Information Collection Level



Configure Intel(R) Thread Profiler (Windows* API)

General | Thresholds | Messages | Instrumentation

Timeline Information

- [✓] Critical path transitions
- [✓] Thread activity

Warning: enabling the thread activity option may result in a marked increase in memory usage, file size, and GUI load time.

Call site Information

Call sites to Record

- (●) Record all call sites
- ( ) Do not record previous and in call sites
- ( ) Do not record call sites

There are various kinds of call sites associated with a section of the critical path. It is found that the overhead for recording certain types of call sites can be significant depending on the program. Use this option to selectively turn off call sites.

Stack Depth

- (●) Record call site stacks up to a depth of 5
- ( ) Do not limit call site stack depth

Keeping track of all the entire stack for every object on the critical path can be expensive. Use this option to limit the amount of the stack that is tracked.

Additional Resources

Introduction to Parallel Programming

Starting from foundation principles, this course introduces concepts and approaches common to all implementations of parallel programming for shared-memory systems.

Topics include Recognizing parallelism opportunities, dealing with sequential constructs, using threads to implement data and functional parallelism, discovering dependencies and ensuring mutual exclusion, analyzing and improving threaded performance, and choosing an appropriate threading model for implementation.

- For more information on Intel Software College, visit www.intel.com/software/college .

- For more information on software development products, services, tools, training and expert advice, visit www.intel.com/software .

- Put your knowledge to the ultimate test solving coding problems in competitions for multi-threading on multi-core microprocessors and win cash prizes! Intel Multi-Threading Competition Series at www.topcoder.com/intel .

- For more information about the latest technologies for computer product developers and IT professionals, look up Intel Press books at http://www.intel.com/intelpress/ .

- Maximize application performance using Intel® Software Development Products: www.intel.com/software/products/ .

www.intel.com/software/college