

Multi-Core Programming

On Linux

Student Workbook

Intel® Software College



Multi-Core Programming

On Linux

Student Workbook

Intel® Software College



The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation ("Intel") and its contributors ("Contributors") on, as of the date of publication. Intel and the Contributors make no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT, IS PROVIDED "AS IS." NEITHER INTEL, NOR THE CONTRIBUTORS MAKE ANY REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL, THE CONTRIBUTORS, OR THIRD PARTIES. INTEL, AND ITS CONTRIBUTORS EXPRESSLY DISCLAIM ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT THIS DOCUMENT IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL, AND ITS CONTRIBUTORS DISCLAIM ALL LIABILITY THEREFOR.

INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIM ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel, its contributors and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL, OR ITS CONTRIBUTORS HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL, OR ANY CONTRIBUTOR HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel and Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2007, Intel Corporation. All Rights Reserved.

Contents

Lab. 1: Programming with POSIX* Threads	13
1: Starting with HelloThreads	15
Review Questions	15
2: Approximating Pi with Numerical Integration	16
Build and Run the Serial Program	16
Thread the Serial Code	16
Review Questions	17
3: Using Condition Variables	18
Build and Run Original Threaded Program	18
Modify Original Threaded Program to Use Condition Variables	18
4: Using Semaphores	19
Build and Run the Serial Program	19
Build & Run Threaded Program	19
Lab. 2: Programming with OpenMP Threads	21
1: Starting with Hello Worlds	23
Initial Compile	23
Add OpenMP Directives	23
OpenMP Compile	23
2: Computing Pi with Numerical Integration	24
Add OpenMP Directives	24
OpenMP Compile and Run	25
3: Monte Carlo Pi	26
Setup	26
Initial Compile	26
Add OpenMP Directives	26
OpenMP Compile	27
Extra Activities	27
Review Questions	29
Lab. 3: Threaded Programming Methodology	31
1: Compile and Run Serial Code	33
Build and Run Serial Program	33
2: Analyze Serial Code	34
Get Baseline Timing	34
Run VTune Performance Analyzer on Application	34
3: Run OpenMP Code	35
Build and Run Threaded Program	35
4: Use Thread Checker to Check for Threading Errors	36
Instrument and Run Threaded Program	36
Use Thread Checker for Analysis	36
5: Correct Threading Errors	38
Correct Errors and Validate Results	38

	Build and Run Threaded Program	38
6:	Find Threading Performance Issues	39
	Instrument and Run Threaded Program	39
	Use Thread Profiler for Analysis	39
7:	Fix Load Imbalance	41
	Use OpenMP schedule Clause	41
8:	Reduce the Number of printf Calls	42
	Modify Code to Control Number of Progress Updates	42
9:	Modify Critical Regions	44
	Update Serial Code	44
	Name Critical Regions to Reduce Contention	44
	Use Local Variables to Hold Updated Globals	44

Lab. 4: Correcting Threading Errors with Intel® Thread Checker for POSIX Threads..... 47

1A:	Find Prospective Data Races	49
	Build and Run Potential Serial Program	49
	Build and Run Potential Threaded Program	49
1B:	Resolve Data Races	51
	Resolve the Problems	51
	Review Questions	51
2:	Identifying Deadlock	52
	Build and Run the Program	52
	Enable Deadlock to Occur	53
	Fix the Deadlock Problem	53
	Review Question	53
3:	Testing Libraries for Thread Safety	54
	Setting-up and Compilation for Thread Safety Testing	54
	Modifying Binary Instrumentation Levels	55
	Review Questions	56

Lab. 5: Tuning Threaded Code with Intel® Thread Profiler for POSIX* Threads..... 57

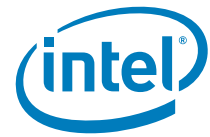
1A:	Getting Started with Thread Profiler	59
	Build and Run Potential Threaded Program	59
	Review Questions	60
1B:	Analyzing an Application	61
2:	Finding Load Balance Issues	62
	Build and Run Potential Threaded Program	62
	Evaluate Thread Profiler Results to Diagnose Problem	63
	Fixing the Performance Issue	64
3:	Finding Synchronization Contention Issues	65
	Build & Run Threaded Numerical Integration Program	65
	Evaluate Results to Diagnose Problem	66
	Using the Timeline View for Analysis	68
	Fixing the Performance Issue	69
	Comparing Performance Runs	70

Review Questions	71
Appendices	73
A: Setting-up Intel® Thread Checker	75
Setting Compile & Link Options	75
Creating a New Project and Thread Checker Activity.....	78
B: Setting-up Intel® Thread Profiler	81
Setting Compile & Link Options	81
Creating a New Project and Thread Checker Activity.....	85
Creating a New Activity by Modifying an Existing Activity	88
C: Creating a New Project and Thread Checker RDC Activity	91
.....	95
Additional Resources	97

Figures

4.1 VTune™ Performance Analyzer's First Use Wizard Welcome Screen	40
4.2 VTune™ Performance Analyzer Help Screen	42
4.3 VTune™ Performance Analyzer Help Screen	43
4.4 VTune™ Performance Analyzer Sampling Configuration Wizard Screen	44
4.5 VTune™ Performance Analyzer Configure Sampling Screen	46
4.6 VTune™ Performance Analyzer Configure Sampling Screen	49
4.7 VTune™ Performance Analyzer Configure Sampling Screen	50
10.1 Thread Profiler Default View Screen	112
10.2 Second Level Grouping under Objects View.....	118
10.3 Filtering by Object and Grouping by Source Code Locations.....	119
10.4 Filtered Profile View	120
10.5 Timeline View (Zoomed)	121
A-1 Project Setting – C/C++ Folder – Debug Options	127
A-2 Linker Settings – Debugging Folder.....	128
A-3 Project Settings – C/C++ Folder – Optimization Options	128
A-4 Project Settings – C/C++ Folder – Thread-safe Libraries Options	129
A-5 Linker Settings – Advanced Attributes	129
A-6 VTune™ Analyzer Easy Start Menu	130
A-7 Intel® Thread Checker New Project Menu.....	131
A-8 Intel® Thread Checker Wizard	132
B-1 Specifying Debug Format	133
B-2 Generating Debug Symbols	134
B-3 Selecting Thread Safe Libraries	135
B-4 Specifying Relocatable Binary	136
B-5 Thread Profiler Wizard.....	137
B-6 Advanced Activity Configuration	138
B-7 Specifying Timeline Information Collection Level.....	139
B-8 Modifying an Existing Activity.....	140
B-9 About to Modify Activity dialog	140
B-10 Advanced Activity Configuration	141
B-11 Specifying Timeline Information Collection Level.....	142
C-1 VTune™ Analyzer Easy Start Menu	143
C-2 Intel® Thread Checker New Project Menu.....	144
C-3 Intel® Thread Checker Wizard	145
C-4 Remote Host Configuration dialog.....	146

C-5	Intel® Thread Checker Wizard dialog box	147
-----	---	-----



Lab. 1: Programming with POSIX* Threads

Time Required	Twenty-five minutes
Objectives	<p>In this lab session, you will practice multi-threaded programming of applications using the POSIX* threading API. Sample programs will create threads, assign work to those threads, synchronize access to shared resources, and coordinate thread execution.</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none">• 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 mutexes, condition variables, and semaphores as synchronization mechanisms

Activity 1: Starting with *HelloThreads*

Time Required	Ten minutes
Objective	<ul style="list-style-type: none">Modify threaded application to print assigned thread number. Find and resolve data races using simple threading techniques, if needed

1. Move to the Hello directory:

```
$ cd ~/classfiles/Pthreads/Hello
```

2. Examine the hello.c source file. This code will create four threads, each of which will print "Hello Thread".
3. Build the application using the associated Makefile.

```
$ make
```

4. Run this version of the application.

```
$ ./hello
```

5. Modify the thread function to report the thread creation sequence (For example, "Hello Thread 0", "Hello Thread 1", "Hello Thread 2", and so on.

Tip:

The loop iterator on the `pthread_create()` loop can be used to give each thread a unique number. The function argument parameter can be used to ensure each thread created gets a value.

6. Build and execute your program.

In what order do the threads execute?
Do the results look correct?
Why or why not?

Review Questions

Question 1: The execution order of threads is unpredictable.

True False

Question 2: What build options are required for any software threaded with Pthreads?

Activity 2: Approximating π with Numerical Integration

Time Required	Fifteen minutes
Objectives	<ul style="list-style-type: none"> • Thread the numerical integration application. • Find any data races in the code and resolve the threading errors using mutexes as the synchronization mechanism.

Build and Run the Serial Program

1. Move to the directory:

```
$ cd ~/classfiles/Pthreads/Pi
```

2. Build the numerical integration application using the provided Makefile.

```
$ make
```

3. Run the application.

```
$ ./pi
```

Is the computed value of π (3.1415...) printed correct?
Why, or why not?

Thread the Serial Code

1. Thread the serial code to compute π 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 evenly amongst the threads such that each iteration is computed by only one thread.
2. Use a mutex 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 using local variables.

3. When you have changed the source code, build and run the threaded application.
4. Keep correcting your source code until you see the correct value of π being printed. The correct value of π is 3.14159.

Review Questions

Question 1: All threads should use the same mutex object.

True False

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

True False

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

True False

Activity 3: Using Condition Variables

Time Required	Fifteen minutes
Objectives	<ul style="list-style-type: none"> Use Windows events to signal when computation threads have completed the assigned work.

The application computes an approximation of the natural logarithm of $(1 + x)$, $-1 < x \leq 1$, 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 created. 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. Change to the condition variable lab directory:

```
$ cd ~/classfiles/Pthreads/CondVarLab
```

2. Build the Mercator series application using the provided Makefile.

```
$ make
```

3. Run the application.

```
$ ./CondVar 1.0
```

4. Note the output generated.

Modify Original Threaded Program to Use Condition Variables

1. Modify the threaded code to use condition variables to replace the spin-waits for the following events:
 - a. wake up the worker threads after the master thread has started
 - b. wake up the master thread after all worker threads have completed computation
2. Build the solution and run the application. Does your new version of the application run correctly?

Activity 4: Using Semaphores

Time Required	Ten minutes
Objectives	<ul style="list-style-type: none"> 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 the Serial Program

1. Change to the semaphore lab directory.

```
$ cd ~/classfiles/Pthreads/SemaphoreLab
```

2. Build the word counting application using the provided Makefile.

```
$ make serial
```

3. Run the application.

```
$ ./serial
```

4. Note the output generated:

Total Words: _____
Total Even Words: _____
Total Odd Words: _____

Build & Run Threaded Program

1. Build the threaded word counting application using the provided Makefile.

```
$ make threaded
```

2. Run the application.

```
$ ./threaded
```

Do you get the same answers as above?

3. Examine the source code. Identify the global variables that are being accessed by each thread.

4. 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.
5. Correct, build and run the threaded code until you are able to achieve the same totals as the serial version of the application.



Lab. 2: Programming with OpenMP Threads

Time Required	Thirty minutes
Objectives	<p>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.</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none">• Use the most common OpenMP* C pragmas• 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. Navigate to the HelloThreads directory:

```
$ cd ~/classfiles/OpenMP/HelloThreads
```

2. Compile serial code using the Intel compiler:

```
$ make
```

3. Run the program in multithreaded environment:

```
$ ./hello
```

Add OpenMP Directives

1. Add an OpenMP parallel directive to include both printf statements in the parallel region:

```
#pragma omp parallel
{
    ...[Code to run in Parallel goes here]...
}
```

OpenMP Compile

1. Compile in OpenMP mode using the Intel compiler.
2. Add a -openmp flag to the compiler variables.
3. Run the program:

```
$ make
```

```
$ ./hello
```

4. Edit the source file to put the second printf statement outside the parallel region.
5. Compile and re-run the application.
Did you get the results that you expected?

Activity 2: Computing Pi with Numerical Integration

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

1. Move to the Pi directory

```
$ cd ~/classfiles/OpenMP/Pi
```

2. Compile and execute the serial version of the Pi program.

```
$ make clean
```

```
$ make
```

```
$ ./pi
```

3. 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)
```

```
{  
    ... [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 region goes here] ...  
}
```

OpenMP Compile and Run

1. Update the Makefile to use -openmp as a compile flag.

```
CF= -openmp
```

2. Build and run the threaded code until you are able to achieve the same totals as the serial version of the application.

```
$ make clean
```

```
$ make
```

```
$ ./pi
```

If you do not get the correct answer, fix the problems and run again until you arrive at the correct answer.

3. Record time: _____.

Activity 3: Monte Carlo *Pi*

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

Setup

1. Move to the Monte Carlo Pi directory:

```
$ cd ~/OpenMP/MonteCarloPi
```

Initial Compile

1. Compile serial code using the Intel compiler (This version is the one that uses the Vector Statistics Library from MKL.):

```
$ make -f Makefile.VSL
```

2. Run the program in a multithreaded environment:

```
$ ./pimonte_VSL
```

3. Record Pi: _____
4. Record the serial time: _____

Add OpenMP Directives

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

```
#pragma omp parallel
{
    // Code to run in Parallel goes here...
}
```

2. Find the loop to make parallel and add the following:

```
#pragma omp for
for(...) {
    // Code within loop goes here...
}
```


3. Examine all variables and determine which ones need to be specially declared. There are hints within the source code with regards to some variables and arrays that need to be private to each thread. The following may be handy:

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

shared(varname,varname)

{

// Code to run in Parallel goes here ...

}
```

4. Depending on your implementation you may need the following. For any remaining shared variables add appropriate locks, if you update that variable.

```
#pragma omp critical [(name)]

{

Code in Critical region goes here...

}

Fortran Syntax:
*$OMP CRITICAL [(name)]

....[Code in Critical section goes here]...

*$OMP END CRITICAL [(name)]
```

OpenMP Compile

1. Edit the file Makefile.VSL so that it will compile an OpenMP version of the code. Build this version.
2. Run the program in a multithreaded environment:

```
$ ./pi
```

3. Record Pi: _____
4. What is the speedup? _____

Extra Activities

1. Optimize the program with compiler switches:

```
$ make "CF=-switch -switch2"
```

1. Replace vsl routines with C runtime rand, random, or rand48 functions, and try to get parallel speedup.
2. Play with the BLOCK_SIZE settings.

Review Questions

Question 1: Name the pragma or directive that would split a loop into multiple threads.
What website has the very readable OpenMP spec?

Lab. 3: Threaded Programming Methodology

Time Required	Seventy-five minutes
Objectives	<p>In this lab session, you will learn and practice the four steps of the Threading Methodology.</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none">• Apply threading on computational loops within C source code using 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	<ul style="list-style-type: none"> 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. On the Linux platform, locate and change to the PrimeSingle directory. You should find a Makefile and a source file, PrimeSingle.c, in this directory.
2. Compile the source file into the executable binary PrimeSingle using the make command.

```
$ make
```

3. After successfully compiling and linking, to run the executable and find the primes between 1 and 2000, run from the command line:

```
$ ./PrimeSingle 1 2000
```

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

Activity 2: Analyze Serial Code

Time Required	Ten minutes
Objective	<ul style="list-style-type: none"> 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

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

```
$ ./PrimeSingle 1 1000000
```

What time is reported from the application? _____ seconds.

Run VTune Performance Analyzer on Application

- Modify the Makefile to compile the application with debug symbols by adding the -g switch to the CFLAGS variable.

```
CFLAGS= -g
```

- Recompile the application.

```
$ make
```

- Start VTune Performance Analyzer and create a new Sampling activity with PrimeSingle application to be launched for analysis. Use '1 1000000' as the command line arguments.
- Run the analysis.

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

- 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?
_____ μ sec

Activity 3: Run OpenMP Code

Time Required	Five minutes
Objective	<ul style="list-style-type: none">Build and run OpenMP threaded version of prime finding application

Build and Run Threaded Program

1. On the Linux platform, locate and change to the PrimeOpenMP directory. You should find a Makefile and a source file, PrimeOpenMP.c, in this directory.
2. Compile the source file into the executable binary PrimeOpenMP using the make command.

```
$ make
```

3. After successfully compiling and linking, to run the executable and find the primes between 1 and 1000000, give the following command:

```
$ ./PrimeOpenMP 1 1000000
```

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	<ul style="list-style-type: none"> Use Intel Thread Checker to find the race condition that is causing the code to compute the wrong number of primes

The prime finding application needs to be rebuilt with the source code instrumentation flag used in the compile phase. Once this is done, running the application will generate a diagnostics file that can be analyzed with the Intel Thread Checker on the Windows platform.

Instrument and Run Threaded Program

1. Within the PrimeOpenMP directory, edit the Makefile to include -tcheck and -g flags in order to compile with Thread Checker source code instrumentation.

```
CFLAGS= -openmp -g -tcheck
```

2. Rebuild the application with the new compile flags.

```
$ make clean
```

```
$ make
```

3. Run the executable with a small range of numbers, for example, 1 to 1000. A small data set size is used since instrumentation increases execution time and memory use by applications. Also, code with threading errors need only be executed once to be recognized and recorded, thus, long runs are unnecessary and discouraged when using Thread Checker.

```
$ ./PrimeOpenMP 1 1000
```

4. List the files in the directory to ensure that the threadchecker.thr file has been generated from the instrumented run.
5. Once execution has terminated successfully, you should copy the source, object, and threadchecker.thr files from the PrimeOpenMP directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker). Depending upon the lab set-up, you may run VMware to open a Windows virtual machine and access files directly from the Linux system or use FTP or some other file transfer system.

Use Thread Checker for Analysis

1. Launch VTune Performance Analyzer on the Windows platform. You can click the Close button on the Easy Start dialog box since you will not need to create a new activity.

Lab. 3: Threaded Programming Methodology

2. From the File menu, select the Open File command.
3. In the dialog box that opens, locate the threadchecker.thr file from the application run. Select the file and click the Open button.
4. Browse to the location of the binary and source code files when asked for their locations within the Map Binary File and Map Source File pop-up dialog boxes, as needed.
5. 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	<ul style="list-style-type: none"> Correct the errors identified from Thread Checker

Correct Errors and Validate Results

1. On the Linux platform, 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 instrumented version of the application.
3. From your Windows system, open the `threadchecker.thr` file within Thread Checker. You may need to copy this file from Linux to your Windows platform.
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, remove the `-tcheck` flag from the `CFLAGS` variable in the Makefile.
2. Compile the source file into the executable binary `PrimeOpenMP` using the `make` command.

```
$ make clean
```

```
$ make
```

3. After successfully compiling and linking, to run the executable and find the primes between 1 and 1000000, give the following command:

```
$ ./PrimeOpenMP 1 1000000
```

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	<ul style="list-style-type: none"> 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.

Instrument and Run Threaded Program

1. Within the PrimeOpenMP directory, edit the Makefile to use the `-openmp_profile` and `-g` flags in order to compile with Thread Profiler source code instrumentation.

```
CFLAGS= -openmp_profile -g
```

2. Rebuild the application with the new compile and link flags.

```
$ make clean
```

```
$ make
```

3. Run the executable with a full range of numbers. 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.

```
$ ./PrimeOpenMP 1 1000000
```

4. Ensure that the `guide.gvs` file has been generated from the instrumented run.

Use Thread Profiler for Analysis

1. Launch (if not already running) VTune Performance Analyzer on the Windows platform. You can click the Close button on the Easy Start dialog box since you will not need to create a new activity
2. From the File menu, select the Open File... command.
3. In the dialog box that opens, locate the `guide.gvs` file. (You may need to have copied this file from Linux to your Windows platform.) Select the file and click the Open button.
4. Browse to the location of the binary and source code files on the Windows platform when asked for their local locations within the Map Binary File and Map Source File pop-up dialog boxes, as needed.
5. Click through the tabs to examine the different views.
From the Summary view, what percentage of time is taken by Imbalance?

From the Threads view, which thread had the most time spent in Imbalance?

From the Summary view, what percentage of time is taken by Locks and Synchronized combined? _____

Activity 7: Fix Load Imbalance

Time Required	Five minutes
Objective	<ul style="list-style-type: none"> Correct the load balance issue between threads by distributing loop iterations in a better fashion

Use OpenMP schedule Clause

- On the Linux platform, 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) )
#pragma omp critical
            globalPrimes[gPrimesFound++] = i;

        ShowProgress(i, range);
    }
```

- Once the source modification is done, change the `-openmp_profile` flag back to `-openmp` in the `CFLAGS` variable in the `Makefile`.
- Compile the source file into the executable binary `PrimeOpenMP` using the `make` command and run the executable with an appropriate range of numbers to test.

```
$ make
```

```
$ ./PrimeOpenMP 1 1000000
```

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

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

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

Activity 8: Reduce the Number of printf Calls

Time Required	Five minutes
Objective	<ul style="list-style-type: none"> Correct the problem of implicit synchronization issue from calling printf more times than necessary

Modify Code to Control Number of Progress Updates

- On the Linux platform, edit your 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++;

    }
}
```

- Compile the source file into the executable binary PrimeOpenMP using the make command and run the executable with an appropriate range of numbers to test.

```
$ make
```


Lab. 3: Threaded Programming Methodology

```
$ ./PrimeOpenMP 1 1000000
```

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)] _____

Activity 9: Modify Critical Regions

Time Required	Ten minutes
Objective	<ul style="list-style-type: none"> Modify the explicit critical regions to reduce contention and limit the amount of time threads spend in each region

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)] _____

Name Critical Regions to Reduce Contention

1. Modify the PrimeOpenMP.c source file to add names to the OpenMP critical regions. Since there is no overlap in variables that each region is protecting, a different name for each region will allow one thread to be executing within each region in parallel. Names for critical regions follow the same rules and restrictions of variable names.
2. Compile the source file into the executable binary PrimeOpenMP using the make command and run the executable with an appropriate range of numbers to test.

```
$ make
```

```
$ ./PrimeOpenMP 1 1000000
```

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

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

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

Use Local Variables to Hold Updated Globals

1. For each critical region within the PrimeOpenMP.c source file, modify the code to protect the update of the global variable and copy the value of the global to a

Lab. 3: Threaded Programming Methodology

variable local to the thread. Then use the local in place of the global for the index of the found primes array and computing the percentage of numbers done.

First modification:

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) ) {
            #pragma omp critical (nameOne)
                lPrimesFound = gPrimesFound++;
            globalPrimes[lPrimesFound] = i;
        }
        ShowProgress(i, range);
    }
```

Second modification:

```
#pragma omp critical (otherName)
    lProgress = gProgress++;
    percentDone = (int)(lProgress/range *200.0f+0.5f)
```

Compile the source file into the executable binary PrimeOpenMP using the make command and run the executable with an appropriate range of numbers to test.

```
$ make
```

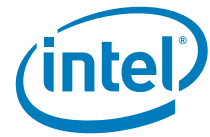
```
$ ./PrimeOpenMP 1 1000000
```

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

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

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

Does this meet with expectations?



Lab. 4: Correcting Threading Errors with Intel® Thread Checker for POSIX Threads

Time Required	Fifty-five minutes
Objectives	<p>In this lab session, you will learn how to debug Pthreads* applications and resolve data race conditions. You will learn how to use Intel® Thread Checker, including how to use it with binary instrumentation and Remote Data Collection. Intel Thread Checker is a powerful tool for finding threading errors.</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Build software to use source code instrumentation for Intel Thread Checker• 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. Change to the serial code directory:

```
$ cd ~/classfiles/ThreadChecker/potential_serial
```

2. Examine the `potential_serial.c` source file. This code runs a simulation of 1000 particles (NPARTS) for 301 time-steps (NITER), computing the potential energy of the system. This potential energy is printed at every tenth time-step.
3. Build the serial application using the associated Makefile:

```
$ make
```

4. Run this version of the application:

```
$ ./potential
```

Build and Run Potential Threaded Program

1. Change to the threaded code directory:

```
$ cd ~/classfiles/ThreadChecker/potential_thred
```

2. Examine the `potential_thred.c` source file. Notice that this code is not only modified to perform the computations in parallel with threads, but also that the number of particles (10) and iterations (11) have been reduced from the serial case. This is done to reduce the execution time of the whole run, because the Thread Checker increases overall execution time.
3. Build the serial application using the associated Makefile:

```
$ make
```

This will use the `-tcheck` flag in order to perform source code instrumentation on the application.

4. Run the application on the Linux platform:

```
$ ./potential_thred
```

When this is complete, you will find that a Thread Checker results file, `threadchecker.thr` is created.

5. Once execution has terminated successfully, you should copy the source, object, and `threadchecker.thr` files from the directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker). Depending upon the lab set-up, you may run VMware to open a Windows virtual machine and access files directly from the Linux system, or use FTP or some other file transfer system.
6. On your Windows platform, start up the VTune Performance Analyzer. You may wish to create a new project, but you will not need to create a new activity because the application has already been executed. Under the File menu, select the Open File... command (Ctrl+O) and read in the `threadchecker.thr` file from the directory on the Linux side (from the Linux platform in Step 4).
7. In the next pop-up dialog boxes that display, browse to the application's binary and source code files.
The diagnostics for the run of the application will now populate the diagnostic pane in the Thread Checker display.
8. Scan through some of the diagnostics displayed. Double-click on one of the diagnostic lines to check 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.

Question 1: Can you see why there is a conflict on those lines of code?

Activity 1B: Resolve Data Races

Time Required	Ten minutes
Objective	<ul style="list-style-type: none">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-examine the corrected application execution 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.
4. Be sure to remove the -tcheck flag from the compilation in order to get unhindered execution.

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

Review Questions

Question 1: What build options are required for source instrumentation?

Question 2: What build options are required for binary instrumentation?

Question 3: Using a larger data set (workload) causes Intel Thread Checker to find more information (errors, warnings, and so on).

True False

Question 4: Threading errors in software can always be corrected by using only synchronization objects.

True False

Activity 2: Identifying Deadlock

Time Required	Fifteen minutes
Objective	<ul style="list-style-type: none"> 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. Change to the deadlock lab directory:

```
$ cd ~/classfiles/ThreadChecker/deadlock
```

2. Examine the `deadlock.c` source file. This code contains an incorrect locking hierarchy between two threaded functions. There is a call to the sleep function that will pause one thread for 5 seconds to help ensure there is no deadlock that would freeze the application execution. Thread Checker will still detect a problem.

3. Build the application using the associated Makefile.

```
$ make
```

4. Run the application on the Linux platform:

```
$ ./deadlock
```

When this is complete, you will find a Thread Checker results file, `threadchecker.thr`, has been created.

5. Once execution has terminated successfully, copy the source, object, and `threadchecker.thr` files from the directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker).
6. On your Windows machine, start up VTune Performance Analyzer. You may wish to create a new project, but you will not need to create a new activity because the application has already been executed. Under the File menu, select the Open File... command (Ctrl+O) and read in the `threadchecker.thr` file.
7. Browse to locate the application's binary and source code files as requested from Map Source File pop-up dialog box.
8. In the diagnostics list, notice the icon shown on the left. This diagnostic has recognized that two threads have accessed synchronization objects, and one thread has accessed the synchronization objects in an order different from that of another thread. This is a caution that deadlock may occur.



Question 1: Why does this set of diagnostics include data contention errors? Are these valid diagnostics?

Enable Deadlock to Occur

1. On the Linux system, use the makefile to remove the application binary and threadchecker.thr file

```
$ make clean
```

2. Edit the `deadlock.c` file. Move the `sleep(5)` call down one line, after the first `pthread_mutex_lock` call. This will increase the chances that an actual deadlock occurs.
3. Build and run the application. Wait 10 seconds or so for the application to terminate on its own. If the application has deadlocked, terminate execution by issuing a Ctrl+C.
4. Ensure that the relevant files are available to your Windows platform. From within Thread Checker on the Windows machine, open the new `threadchecker.thr` file.
How does this set of diagnostics differ from the previous run of the application?
5. Right click on one of the "A Thread is deadlocked" diagnostics. Select the "Diagnostic Help" option to get more information about thread deadlock and how Thread Checker presents information on this error.
6. Double-click on one of the "A Thread is deadlocked" diagnostics to examine the source lines that are involved with the deadlock situation.

Fix the Deadlock Problem

1. Return to Linux system to correct the threading problems identified by Thread Checker within the program.

Tip: Notice the order that the `pthread_mutex_t` objects `mtx0` and `mtx1` are used in each thread function `work0()` and `work1()`.

2. When you have modified the source code, rebuild the application and run it again. 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, `deadlock_solution.c` in the ThreadChecker/solutions directory.

Review Question

Question 1: 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	<ul style="list-style-type: none"> • Use Intel Thread Checker to determine if libraries are thread-safe • Use the advanced features of Intel Thread Checker, including instrumentation levels

Setting-up and Compilation for Thread Safety Testing

1. In your home directory, create a lib and include directory.
2. Change to the thread safety lab directory:

```
$ cd ~/classfiles/ThreadChecker/thread_safe_libraries
```

There is a related directory for this lab that holds the library source files and builds the shared object library that will be tested for thread safety.

3. Build the application using the associated Makefile.

```
$ make
```

This will also build the library and associated include file. These will be copied over to the ~/lib and ~/include directories, respectively

4. Run the application on the Linux platform.

```
$ ./libtester
```

This is a test code that calls the three library functions under consideration. The code is used to demonstrate how to call the functions.

OpenMP will be used to create a quick framework under which combinations of calls to the three library routines can be used to generate threaded calls to the library routines involved. Intel Thread Checker will be used to determine if there are any data conflicts or other problems.

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 add the `-openmp` and `-tcheck` flags to the compile and link flags in the Makefile.
7. Rebuild the application.

Note:

You will need to start an Intel Threading Tools server (ittserver) on the Linux platform in order to perform a remote data collection (RDC). This will perform a binary instrumentation of the application and the "third-party" library.

8. To launch the ittserver, open a window on the Linux machine and type the command:

```
/opt/intel/itt/bin/ittserver -d /tmp -clear-all
```

9. On the Windows machine, set up a Thread Checker activity to the Linux machine (RDC) and the libtester executable.
10. Run the application through the Thread Checker activity on the Windows machine. When this is complete, the Thread Checker diagnostics pane should be updated with any diagnostics found in the execution of the run.
11. If you have any error diagnostics, double-click on them to view the source code. Browse to locate the application's binary and source code files from Map Binary File and Map Source File pop-up dialog boxes as needed. If you are initially presented with the assembly language of the library calls, find the program stack frame in the call stack in order to determine which library calls were involved.

Were any diagnostics found?

Modifying Binary Instrumentation Levels

Even though the dynamic library 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-shared libraries 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 database 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 displays, select the "Modify the selected Activity" radio button, and click OK.
3. Click on the Instrumentation tab in the "Configure Intel Thread Checker" box.
4. Left-click on the "Instrumentation Level" entry for the "library.so" 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 library at the chosen level.

5. Click OK.
6. Re-run the Thread Checker analysis.

Were any diagnostics found this time?

Review Questions

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



Lab. 5: Tuning Threaded Code with Intel® Thread Profiler for POSIX* Threads

Time Required	Forty-five minutes
Objectives	<p>In this lab session, you will use Intel® Thread Profiler to detect performance issues in POSIX* threaded applications.</p> <p>After successfully completing this lab's activities, you will be able to:</p> <ul style="list-style-type: none">• 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	<ul style="list-style-type: none">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.

Build and Run Potential Threaded Program

1. Change to the code directory for the first Thread Profiler lab

```
$ cd ~/classfiles/ThreadProfiler/potential_lab1
```

2. Examine the `potential1.c` source file. This code will run a simulation of 1000 particles (NPARTS) for 21 time steps (NITER) computing the potential energy of the system. That potential energy will be printed every tenth time-step. The number of time-steps is reduced not only in order to have the instrumented code run quickly, but also as to not overrun the Thread Profiler with results.
3. Build the serial application using the associated Makefile.

```
$ make
```

This will use the `-tpprofile` flag in order to perform source code instrumentation on the application.

4. Run the application on the Linux platform.

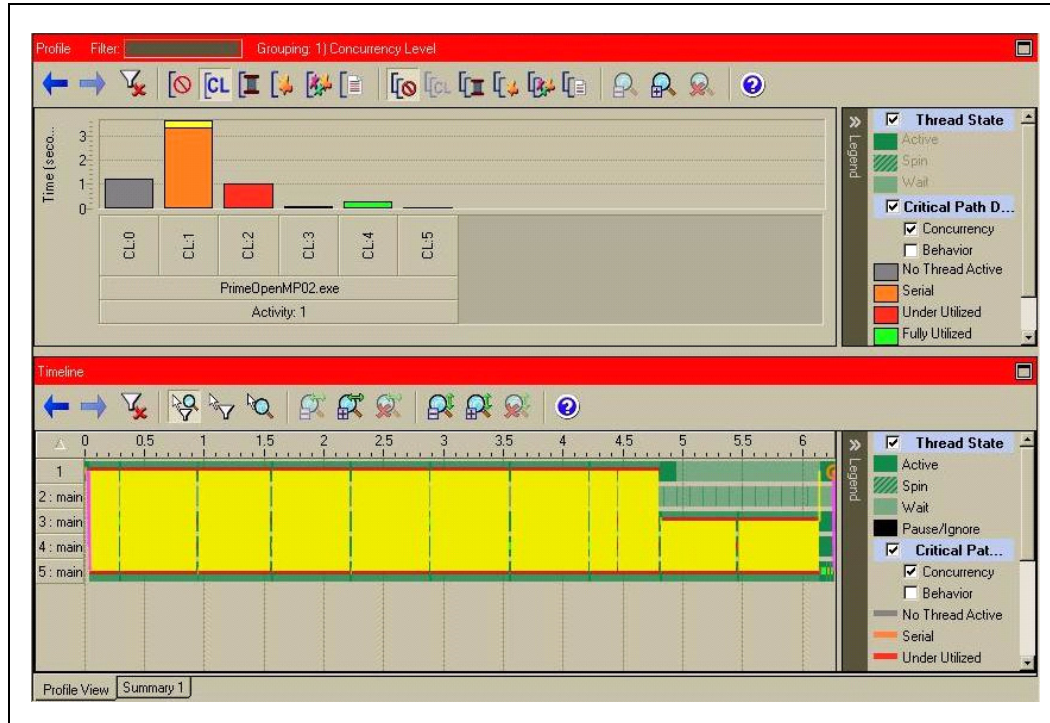
```
$ ./potential1
```

When this is complete, you will find a Thread Profiler results file, `tp.tp`, has been created.

5. Once the execution has terminated successfully, you should copy the source, object, and `tp.tp` files from the directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker). Depending upon the lab set-up, you may run VMware to open a Windows virtual machine and access files directly from the Linux system or use FTP or some other file transfer system.
6. On your Windows platform, start up VTune Performance Analyzer. You may wish to create a new Thread Profiler project, but you will not need to create a new activity since the application is already being executed. Under the File menu, select the Open File... command (Ctrl+O) and read in the `tp.tp` file.
7. Browse to the application's binary and source code files if the Map Binary File or Map Source File pop-up dialog boxes appear.

8. On completion of the run, you should see the Critical Path view as shown in Figure 5.1.

Figure 5.1. Thread Profiler Default View Screen



9. Expand the Profile View. The "Concurrency Level" view should be the default. If this is not the case, click on the "Concurrency Level" button in the toolbar.



10. 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 pop windows appear and what data is contained within them.

11. Expand the "Timeline View" of the performance data. Explore the window 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	<ul style="list-style-type: none">Examine the different analysis views offered by Intel Thread Profiler and determine if a performance issue is evident from the information presented



1. Expand the Profile View pane and bring up the "Concurrency Level" view.

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

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

What is this view telling you? _____



2. Select the "Threads View" by clicking on that button. This view shows all the threads in the applications that were active in this critical path.

How many total threads were used during the execution of this application?

Can you tell if there is some performance problem inherent in the application?

Yes No

If so, what problem is it? If not, why not?



3. Select "Objects View" by clicking on that button.

Are there any synchronization objects that account for a significant portion of the critical path time spent in serial or under-utilized impact time?

If so, which one(s)?

4. Expand the Timeline View pane.

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	<ul style="list-style-type: none"> 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 have been divided equally between threads, there is still a load balance issue with this code.

Build and Run Potential Threaded Program

1. Change to the code directory for the second Thread Profiler lab.

```
$ cd ~/classfiles/ThreadProfiler/potential_lab2
```

2. Looking over the source code in `potential2.c`, you should notice the definition and initialization of the two condition variables (`bSignal` and `eSignal`) and the associated mutexes (`bMutex` and `eMutex`). Also, notice the use of the `start` and `check_in` variables within the `tPoolComputePot` function. These control the coordination of worker threads and the main thread execution. The `done` variable informs threads when the simulation is complete. Change the number of threads to be created for this application (`NUM_THREADS`) to equal the number of cores (logical processors) on the system that you are using. Check the `/proc/cpuinfo` file to get this number.
3. Build the threaded application using the associated Makefile.

```
$ make
```

This will use the `-tprofile` flag in order to perform source code instrumentation on the code.

4. Run the application on the Linux platform.

```
$ ./potential2
```

When this is complete, you will find a Thread Profiler results file, `tp.tp`, has been created.

5. Once the execution has terminated successfully, you should copy the source, object, and `tp.tp` files from the directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker). Depending upon the lab set-up, you may run VMware to open a Windows virtual machine and access files directly from the Linux system or use FTP or some other file transfer system.

6. On your Windows platform, start up VTune Performance Analyzer. You may wish to create a new Thread Profiler project, but you will not need to create a new activity since the application is already being executed. Under the File menu, select the Open File... command (Ctrl+O) and read in the `tp.tp`.
7. Browse to the application's binary and source code files if the Map Binary File or Map Source File pop-up dialog boxes appear. You can Skip any system library that are requested.

Evaluate Thread Profiler Results to Diagnose Problem

1. Looking at the Profile View, you will notice that there isn't much impact time on the critical path. With each thread using the same two mutex objects, one might assume that there would be quite a bit of object contention and, consequently, a much larger percentage of time would be spent in "Impact time."
2. Bring up the Objects profile view.

Question 1: Can you tell if there is one object that has been involved with the Serial Impact Time? If so, which one?

From the data shown, what percentage of time was spent with some kind of impact on this synchronization object? (You can click on the lifetime bar for each object to populate the Statistics pane and then compare Total time.)

Note: Due to the way condition variable functions handle the associated mutexes, there isn't much time spent with multiple threads contending for the same mutex.

3. If there is no performance problem with the synchronization of threads with each other or the main thread, we need to look for other potential trouble spots. Another possible cause of non-optimal performance may be within the threads themselves. Bring up the Threads profile view.

Can you tell if one or more threads were involved in the "Serial impact time" from the critical path? If so, which ones?

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

Thread 2 Lifetime: _____	Thread 2 Active time: _____
Thread 3 Lifetime: _____	Thread 3 Active time: _____
Thread 4 Lifetime: _____	Thread 4 Active time: _____
Thread 5 Lifetime: _____	Thread 5 Active time: _____

Question 2: What does this tell you about the application's threaded execution?

Question 3: 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 your chosen editor.
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 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 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	<ul style="list-style-type: none"> 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 mutex 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. Change to the code directory for the first Thread Profiler lab

```
$ cd ~/classfiles/ThreadProfiler/NumericalIntegration
```

2. Build the threaded application using the associated Makefile.

```
$ make
```

This will use the -tprofile flag in order to perform source code instrumentation on the code.

3. Run the application on the Linux platform. Include the number of threads that you wish to execute with on the command line. This number should be the same as the number of logical cores on the system you are using. (The default for not including this argument will be 2 threads.)

```
$ ./NumericalIntegration 4
```

Did the program get an answer close to the true value? Yes No

4. Once execution has terminated successfully, you should copy the source, object, and tp.tp files from the directory on the Linux platform to a location on the Windows platform (this is where you will execute Thread Checker). Depending upon the lab set-up, you may run VMware to open a Windows virtual machine to access files directly from the Linux system or use FTP or some other file transfer system.
5. On your Windows platform, start up VTune Performance Analyzer. You may wish to create a new Thread Profiler project, but you will not need to create a new activity since the application is already being executed. Under the File menu, select the Open File... command (Ctrl+O) and read in the tp.tp file.

6. Browse to the application's binary and source code files if the Map Binary File or Map Source File pop-up dialog boxes appear. You can Skip any system library that is requested.

Evaluate Results to Diagnose Problem

On completion of the run, you should expand the Profile 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 underutilized (serial, if the application was run with 2 threads) states.



1. If the Concurrency Level view does not open as the default, click on the "Concurrency Level" button in the toolbar.
What is this view telling you? _____



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

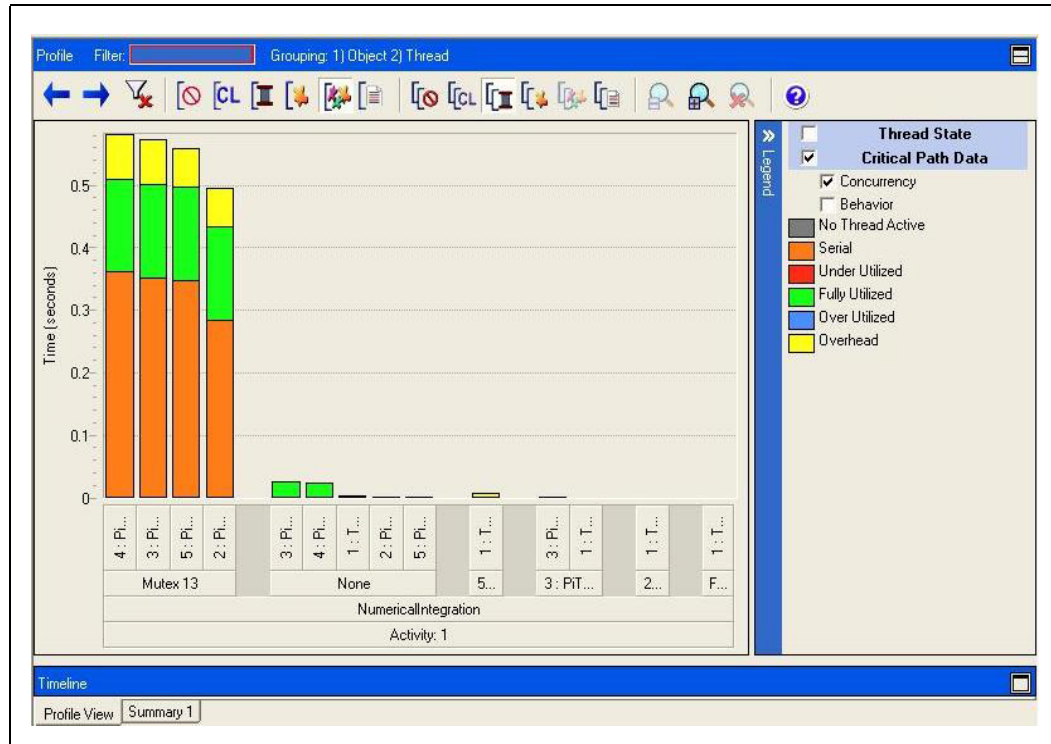


3. Select "Objects View" by clicking on that button. This view will show that the application is impacted by one Critical Section object where all of the accumulated impact is present.



4. Select a second level grouping in the Object for "Thread" to bring up the Second Level Grouping under Objects view shown in [Figure 5.2](#).

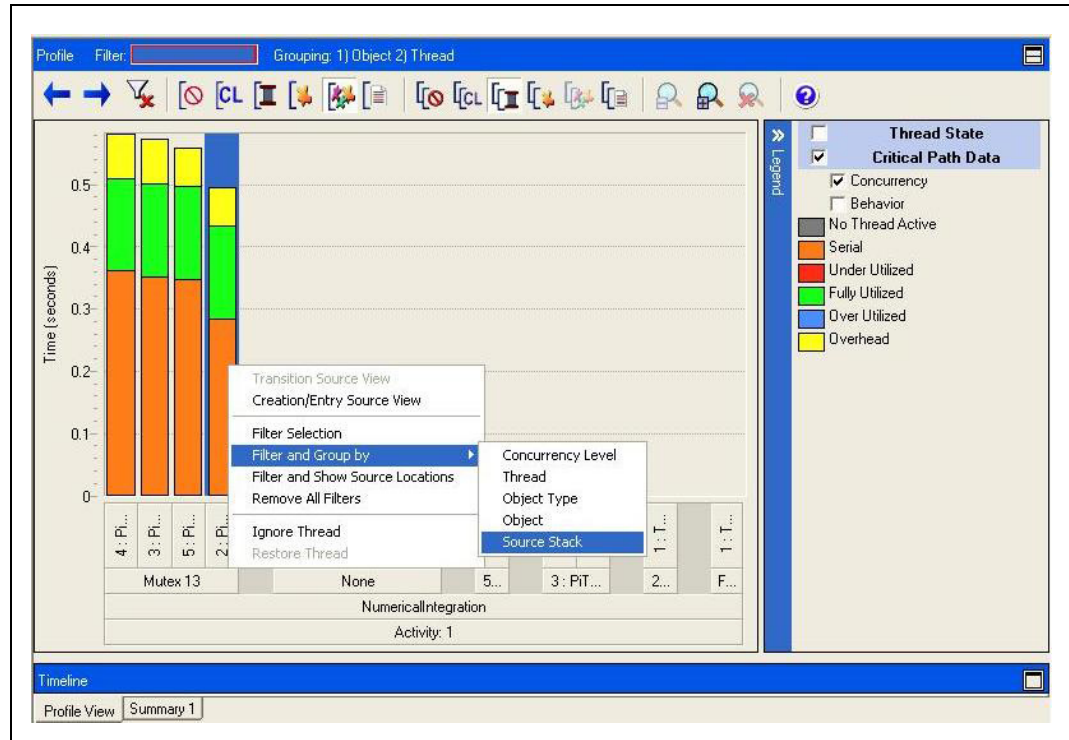
Figure 5.2. Second Level Grouping under Objects View



All worker threads are impacted by the same mutex object. The next few steps show how to get from the impacting object to the source line.

1. Right-click one of the Threads bar impacted by the mutex object to see the pop-up Filtering by Object and Grouping by Source Code Locations menu shown in [Figure 5.3](#).

Figure 5.3. Filtering by Object and Grouping by Source Code Locations

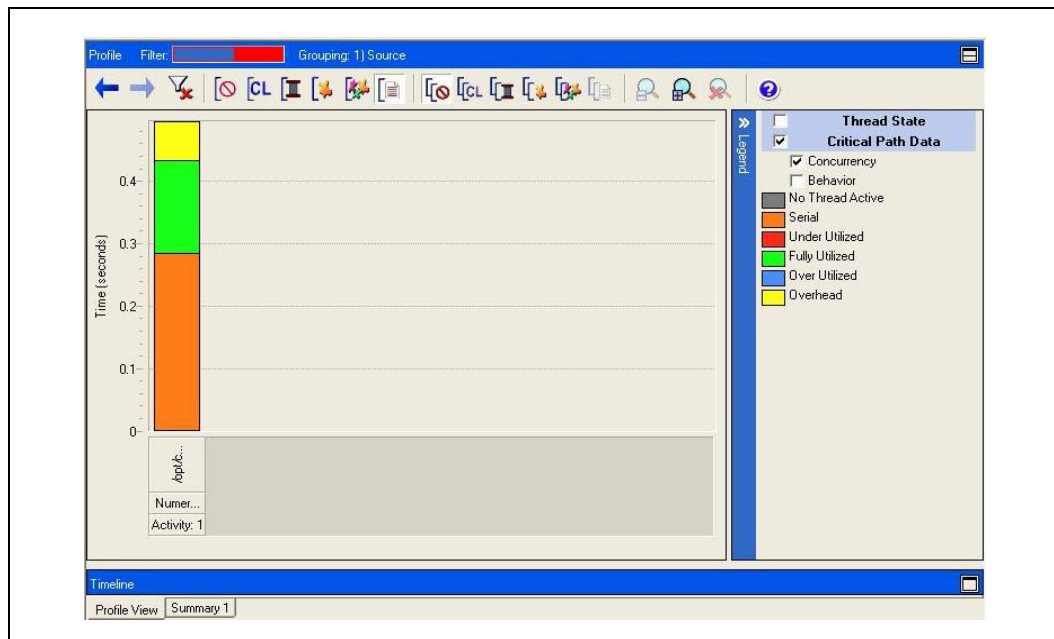


2. Select "Filter and Group by" drop down menu and in the drop-down, select "Source Stack".

This Filtered Profile View screen displays as shown in [Figure 5.4](#).


Each bar represents a thread that interacted with the thread chosen in the previous Thread Profiler screen through the selected mutex object. Right-click on one of the bars and select "Transition Source View" in the selection to get a source view of the code involved with the selected mutex object


Figure 5.4. Filtered Profile View



3. Right click on the bar graph and select "Transition Source View" in the selection to get a source view of the code involved with the selected mutex object.

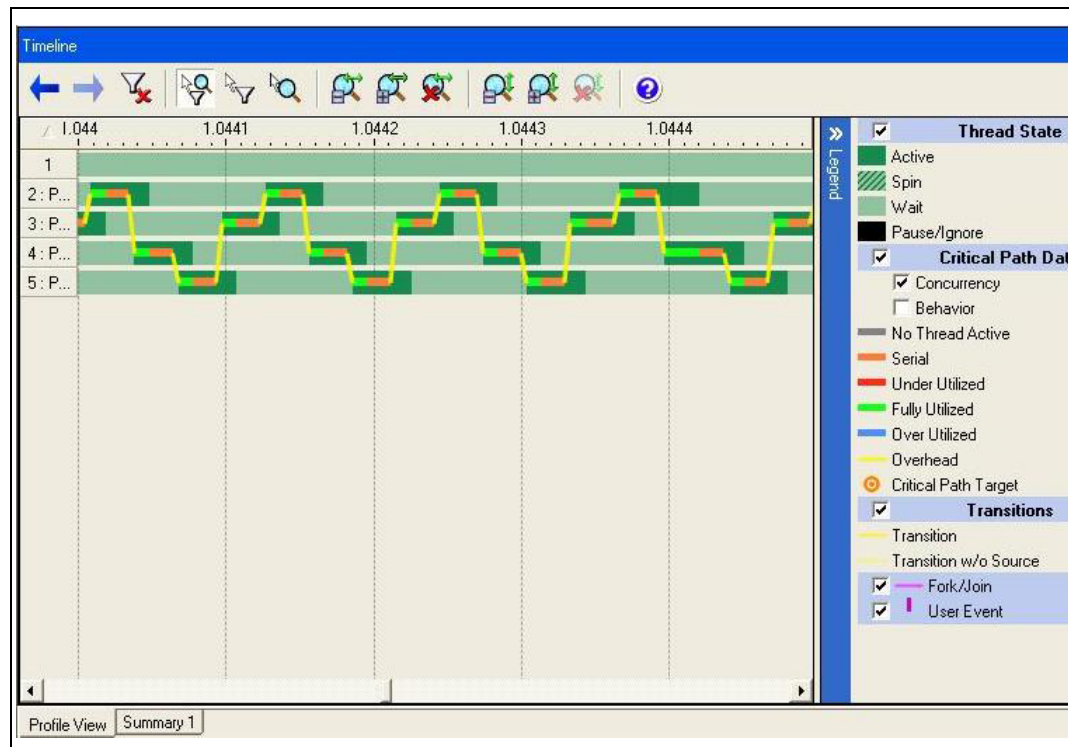
Using the Timeline View for Analysis

1. Remove the filtering by clicking on the  icon.

(Though it is not necessary, you may also want to go back to a simple display by clicking on the  icon.)

2. Expand the Timeline view.
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 Timeline View (Zoomed) shown in [Figure 5.5](#).

Figure 5.5. 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 mutex 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. Expand the Profile View.
2. Remove all filtering and grouping by clicking on the  icon.
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" tabs.

Question 1: Is there a noticeable improvement from the original execution performance?

Review Questions

Question 1: Is over utilization 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



Appendices

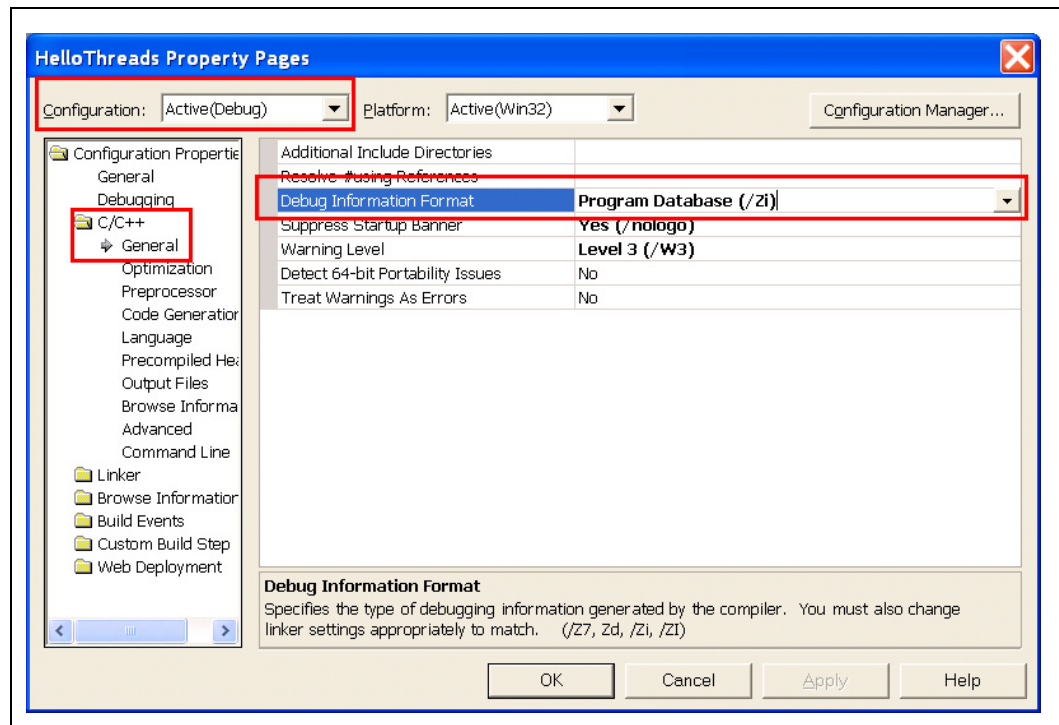
Appendix A: Setting-up Intel® Thread Checker

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.

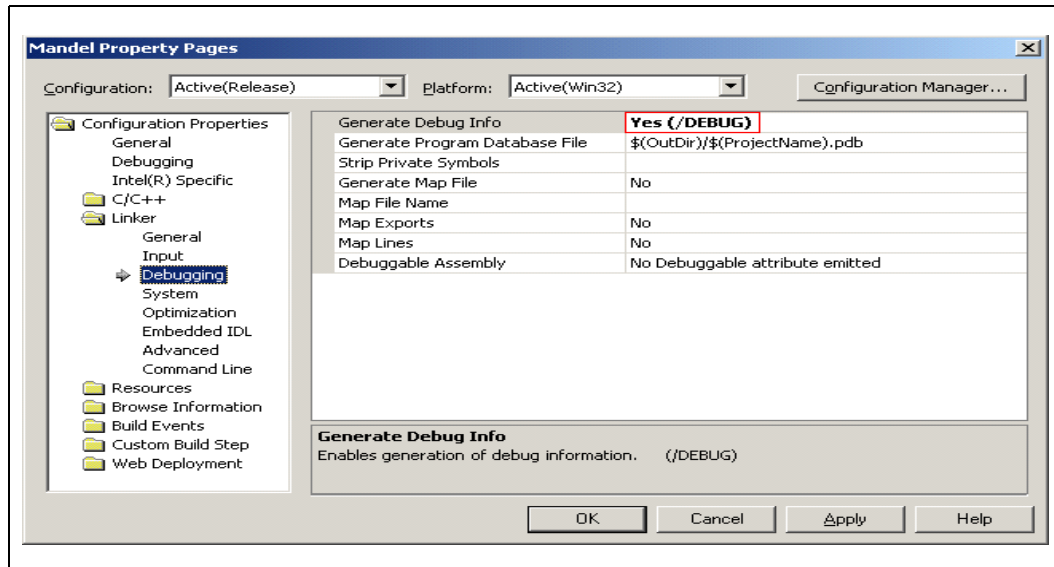
1. Ensure that the Debug options are selected, as shown in Figure A-1.

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



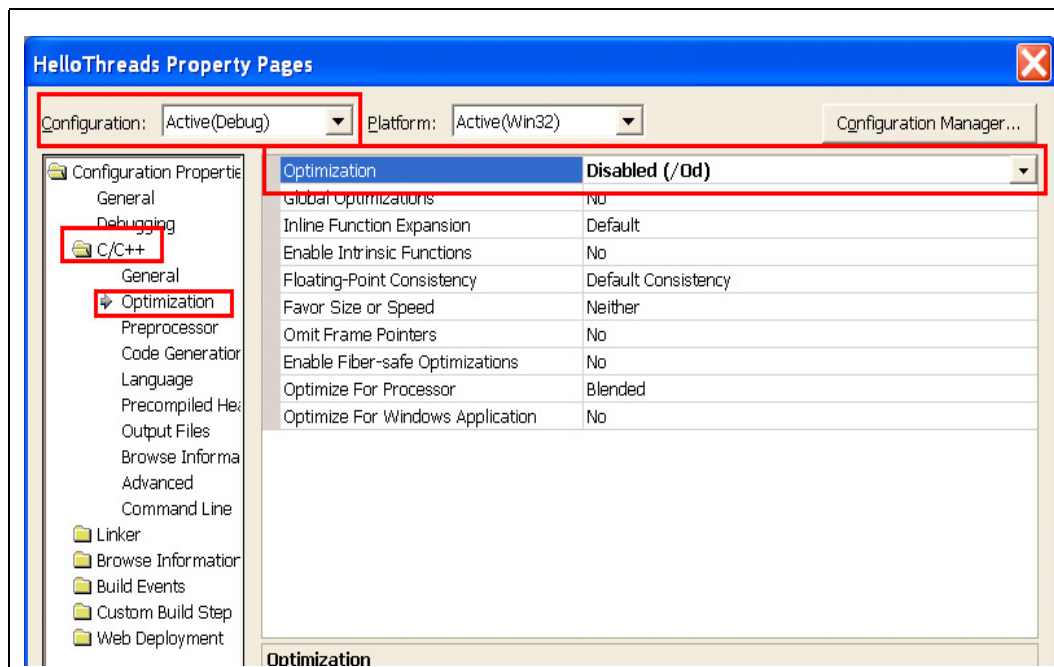
2. Ensure that the Debug symbols are preserved during the link phase, as shown in Figure A-2.

Figure A-2. Linker Settings – Debugging Folder



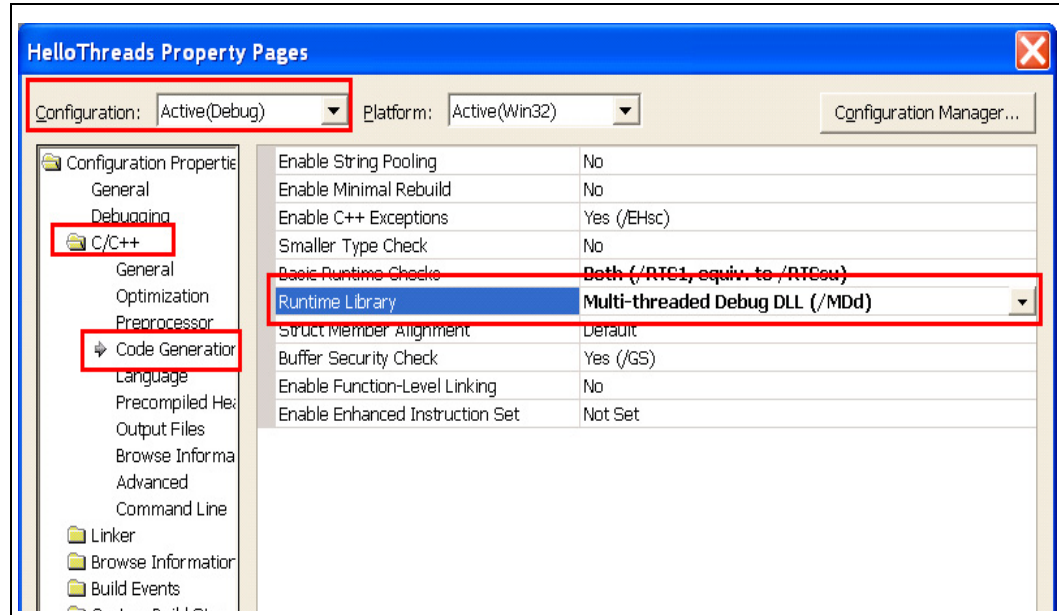
3. Ensure that the Optimization is disabled, as shown in Figure A-3.

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



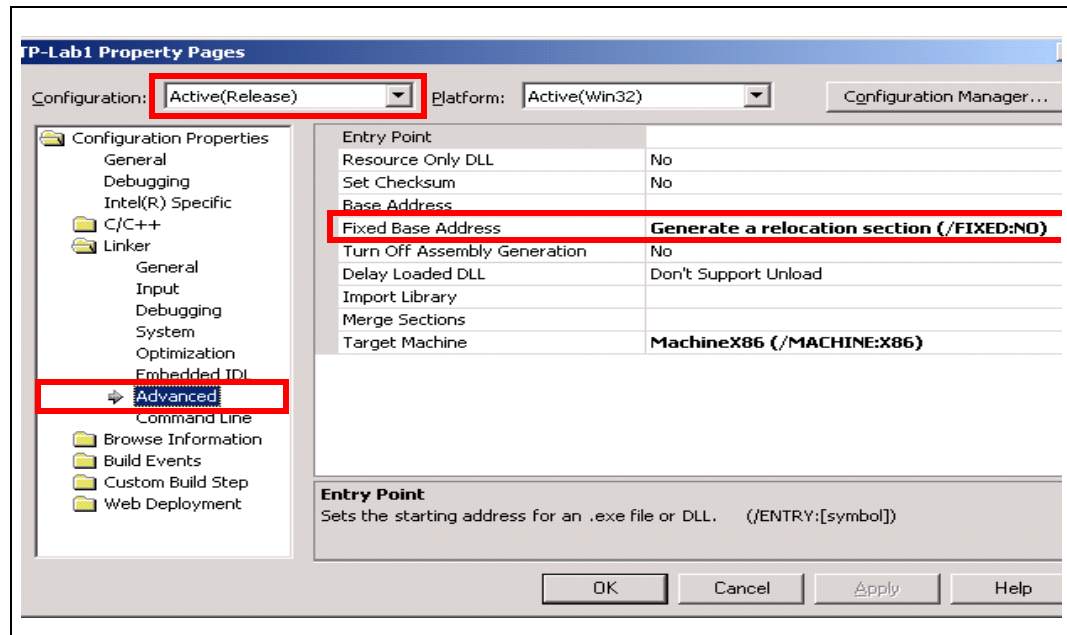
4. Ensure that the thread-safe libraries are selected, as shown in Figure A-4.

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



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

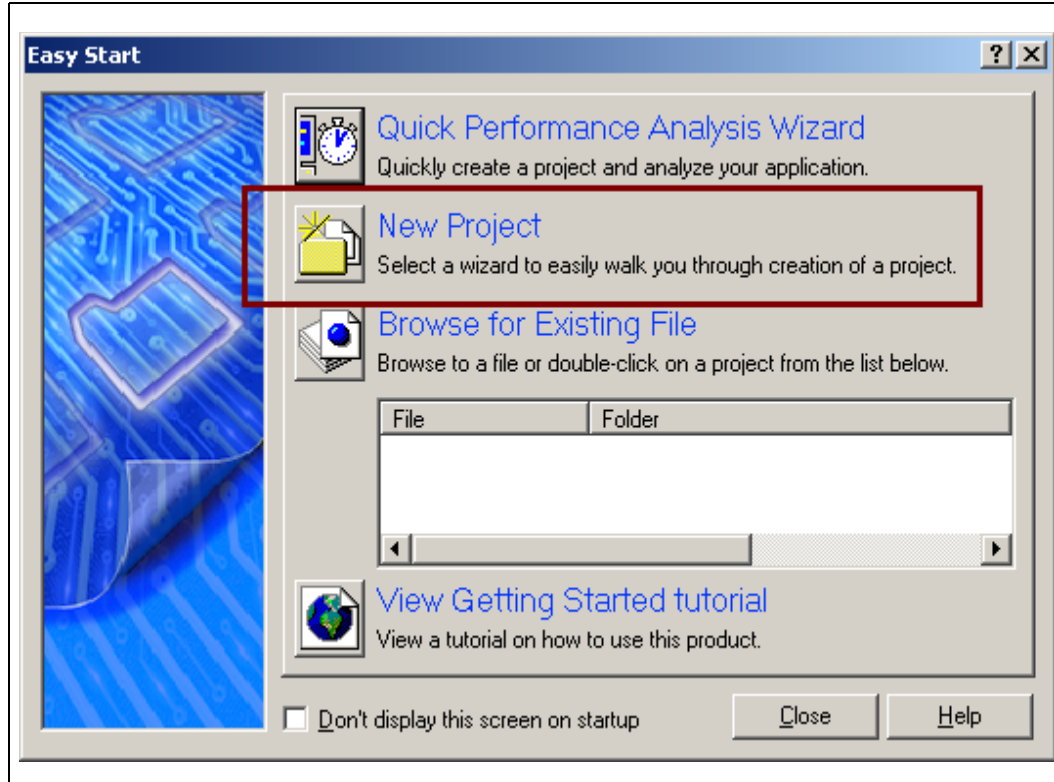
Figure A-5. Linker Settings – Advanced Attributes



Creating a New Project and Thread Checker Activity

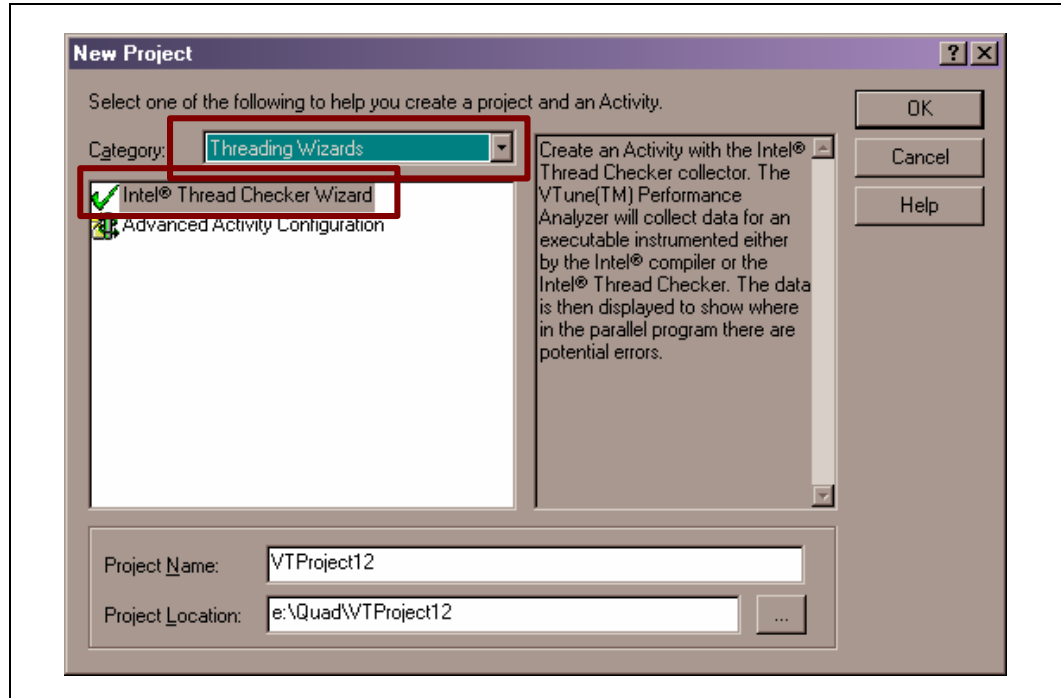
1. Start VTune Performance Analyzer and select New Project, as shown in Figure A-6.

Figure A-6. VTune™ Analyzer Easy Start Menu



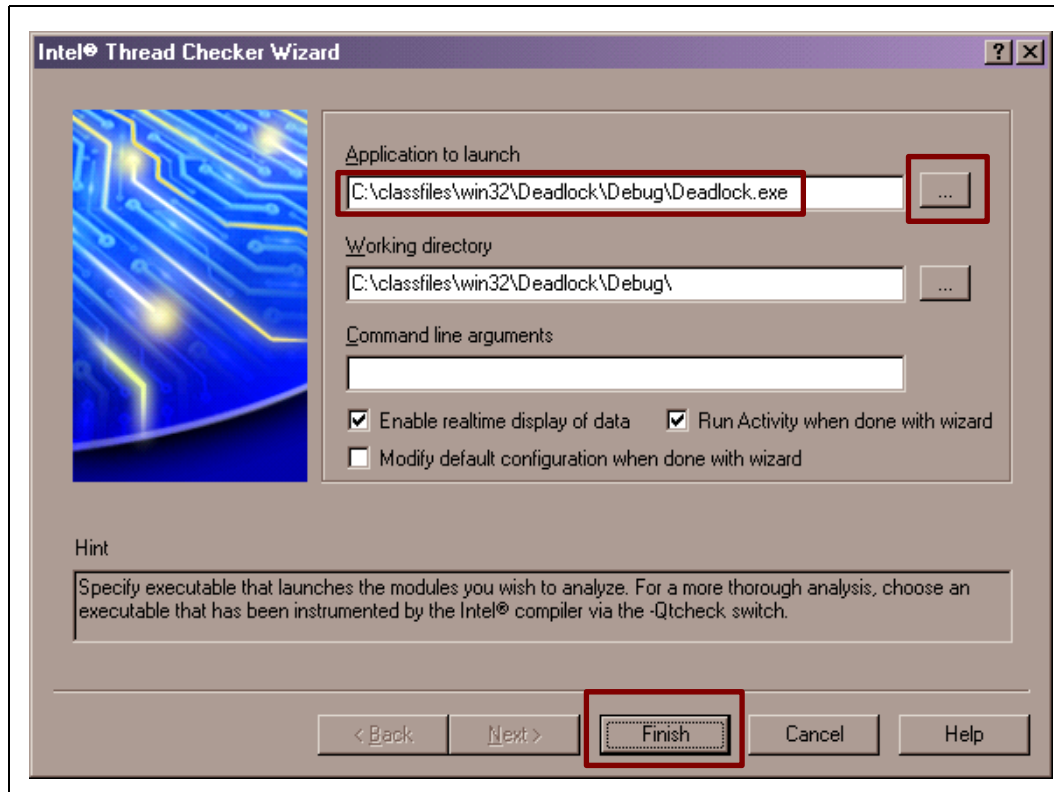
2. From the Category Threading Wizards, select the Intel® Thread Checker Wizard, as shown in Figure A-7.

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



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



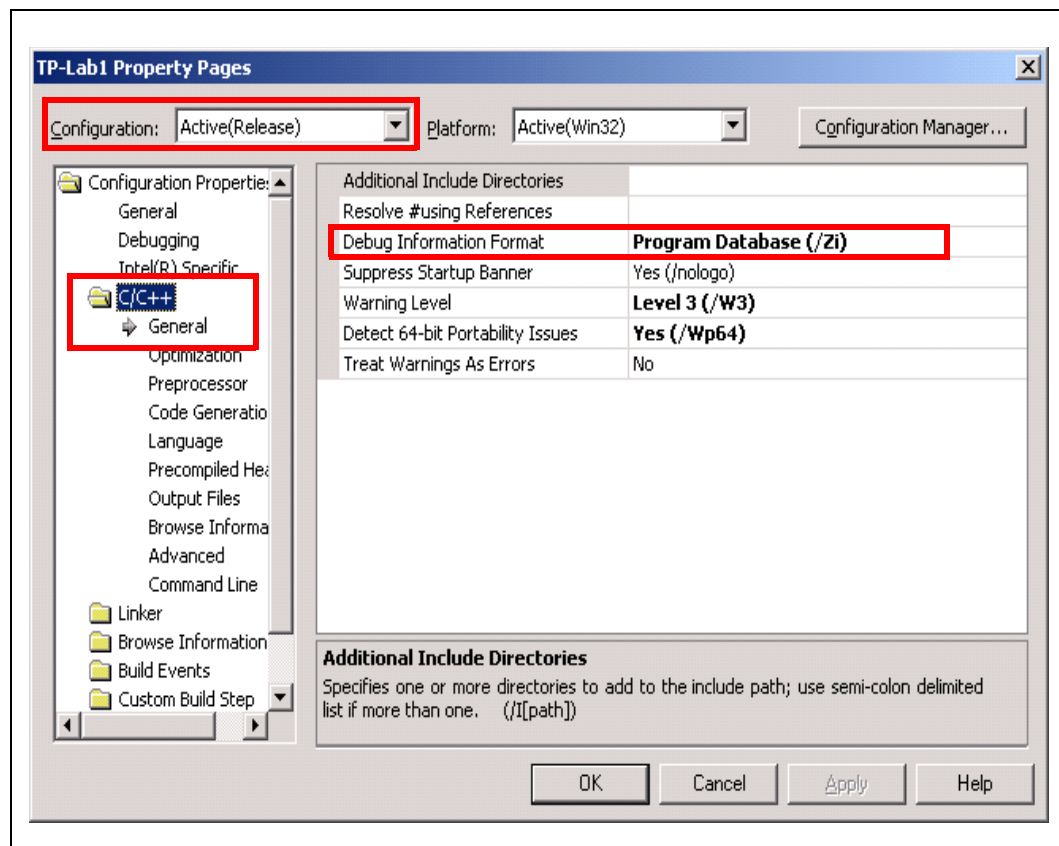
Appendix B: Setting-up Intel® Thread Profiler

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.

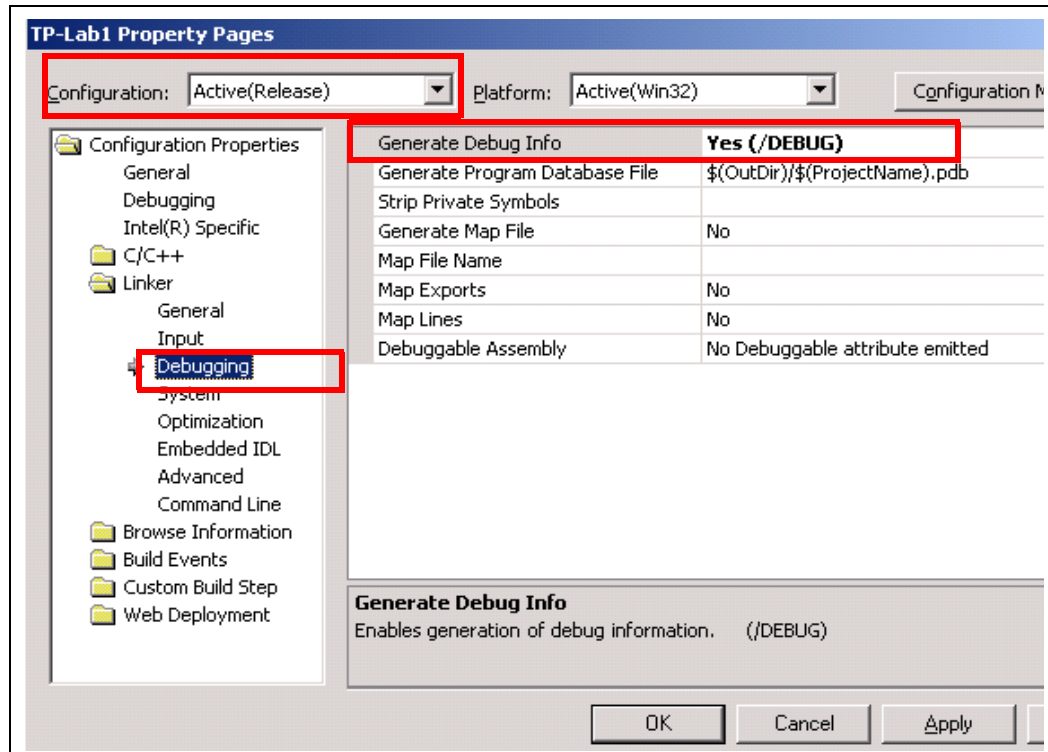
1. From the Build menu, select Configuration Manager... and then select the Release build.
2. Make sure that Debug format is specified as shown in [Figure B-1](#).

Figure B-1. Specifying Debug Format



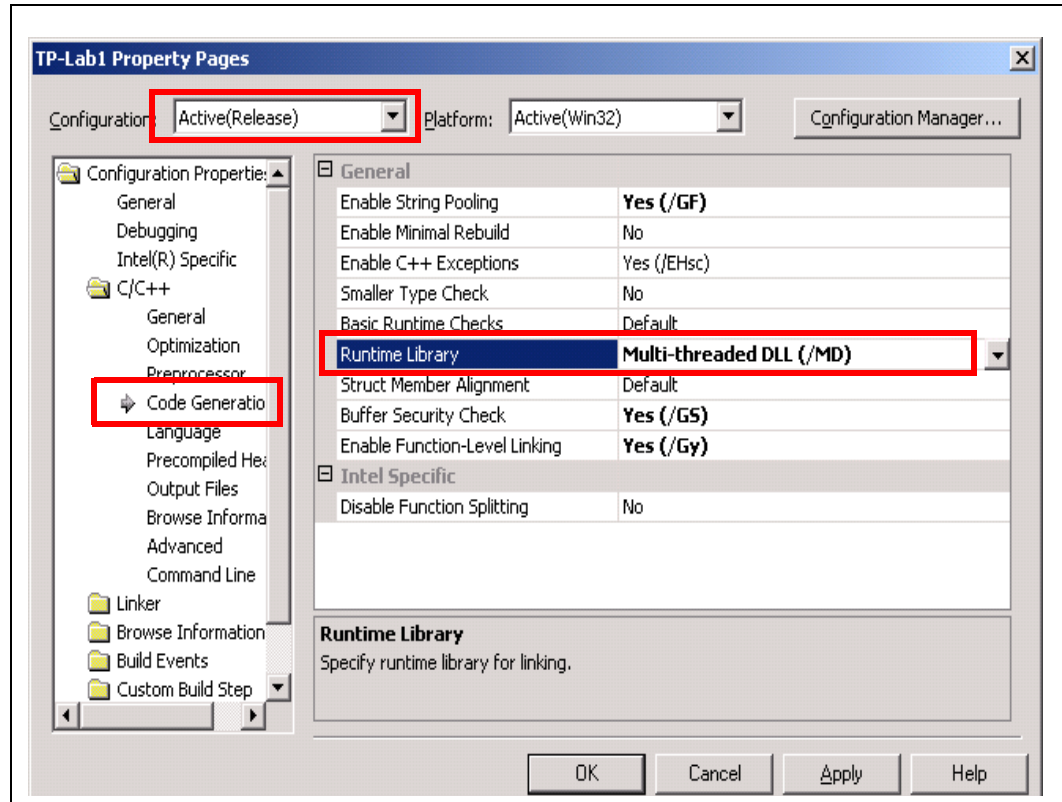
3. Make sure that the debug symbols are generated for the application as shown in Figure B-2.

Figure B-2. Generating Debug Symbols



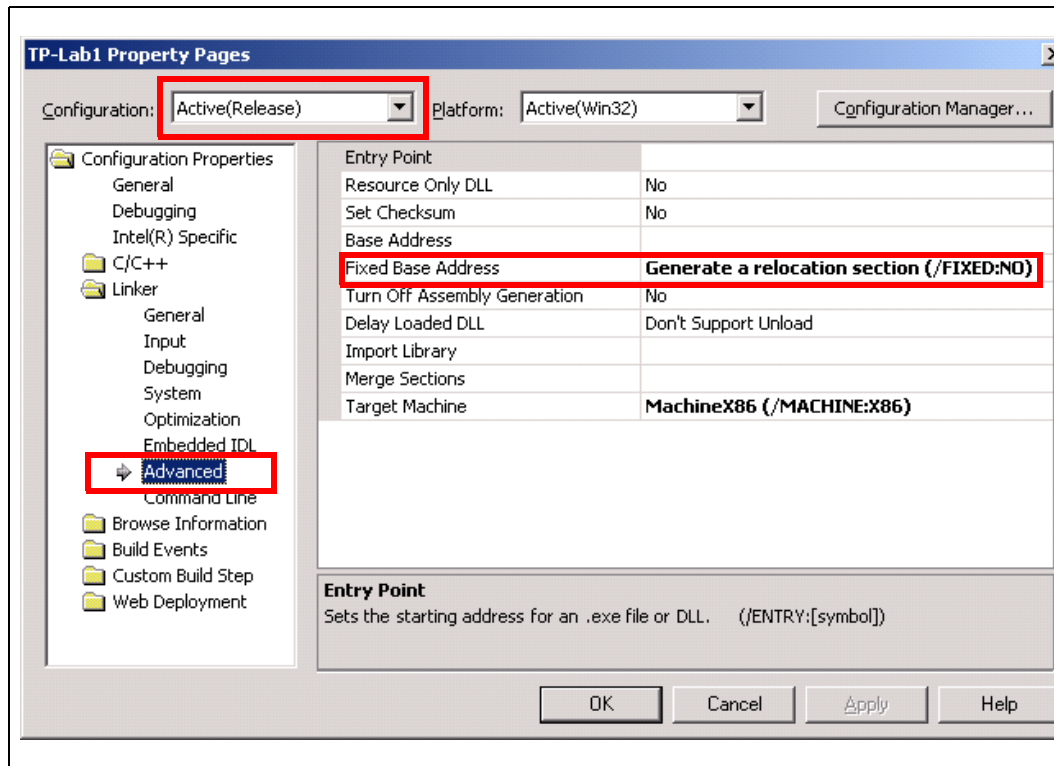
4. Make sure that thread safe libraries have been selected as shown in Figure B-3.

Figure B-3. Selecting Thread Safe Libraries



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

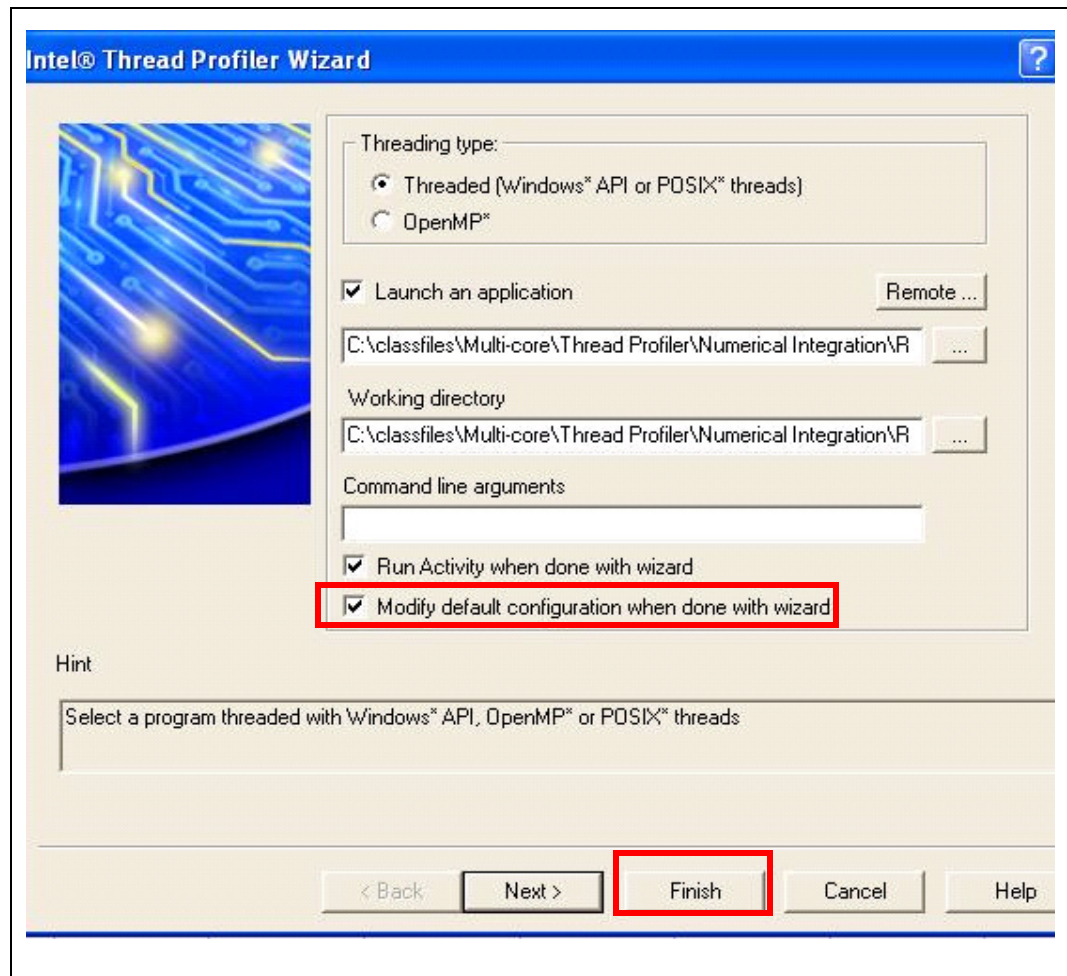


6. From the Build menu, select Build Solution to build. The application will now be ready to be run under the Thread Profiler.

Creating a New Project and Thread Checker Activity

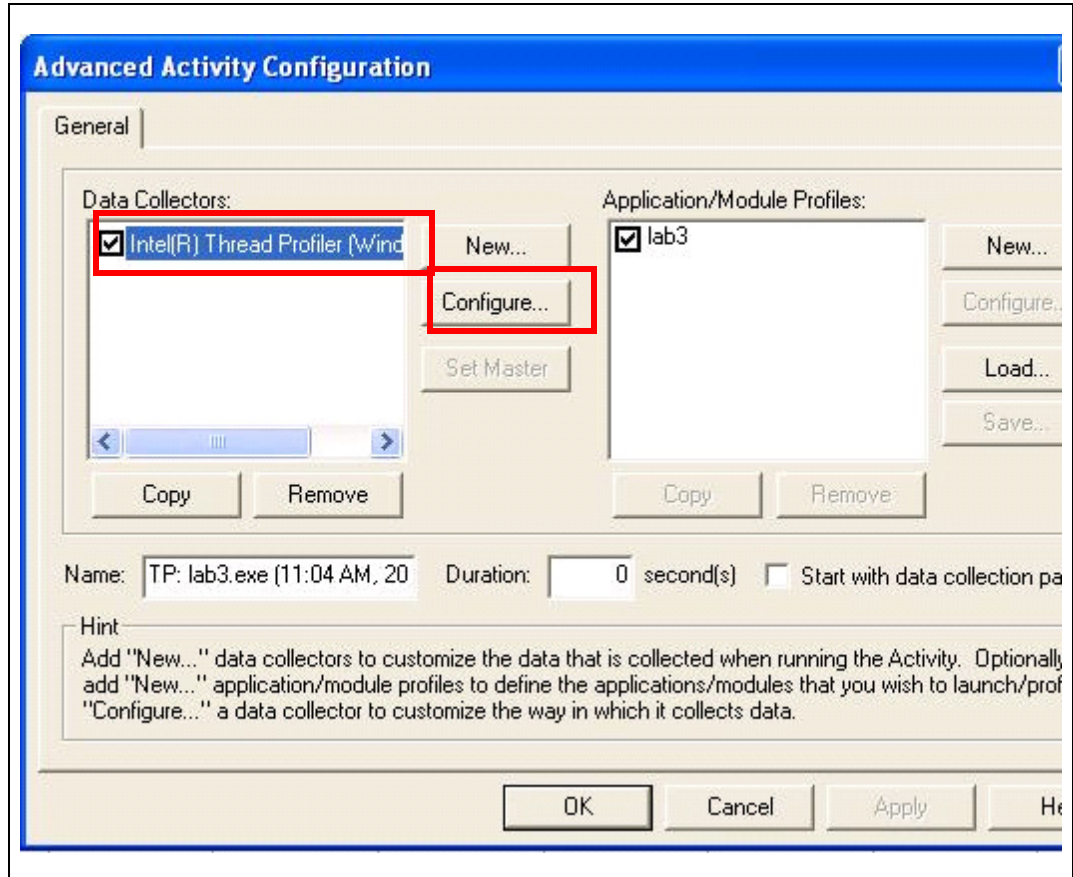
1. Start VTune™ Performance Analyzer and select New Project.
2. From the Category Threading Wizards, select the Intel® Thread Profiler Wizard. (Figure B-5)
3. Within the Thread Profiler Wizard, be sure to click on the “Threaded (Windows* API or POSIX* threads)” radio button is selected.
4. Choose an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

Figure B-5. Thread Profiler Wizard



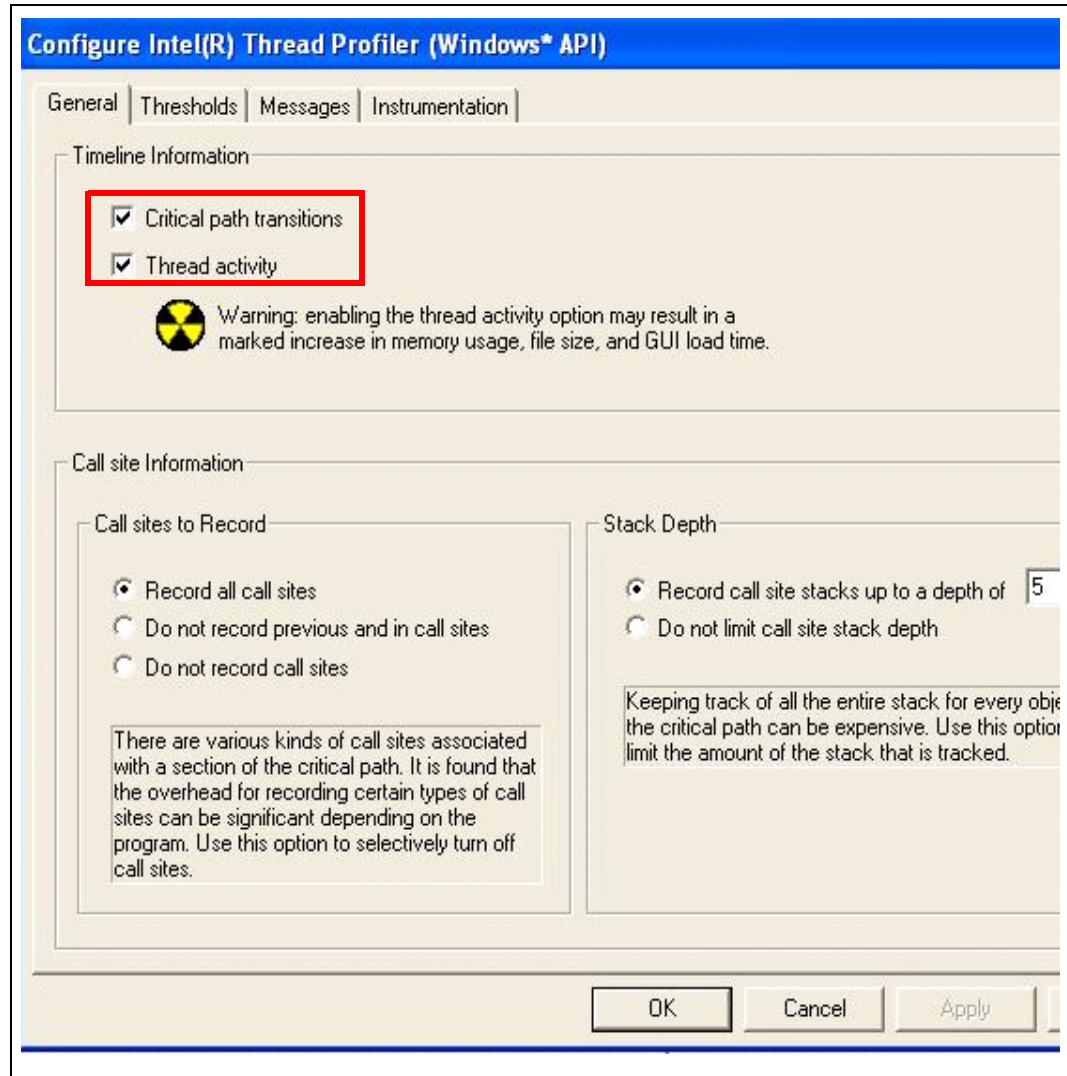
5. Select "Thread Profiler" and click on "Configure" (Figure B-6). In the resulting pop-up dialog, select the "Miscellaneous" Tab.

Figure B-6. Advanced Activity Configuration



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".

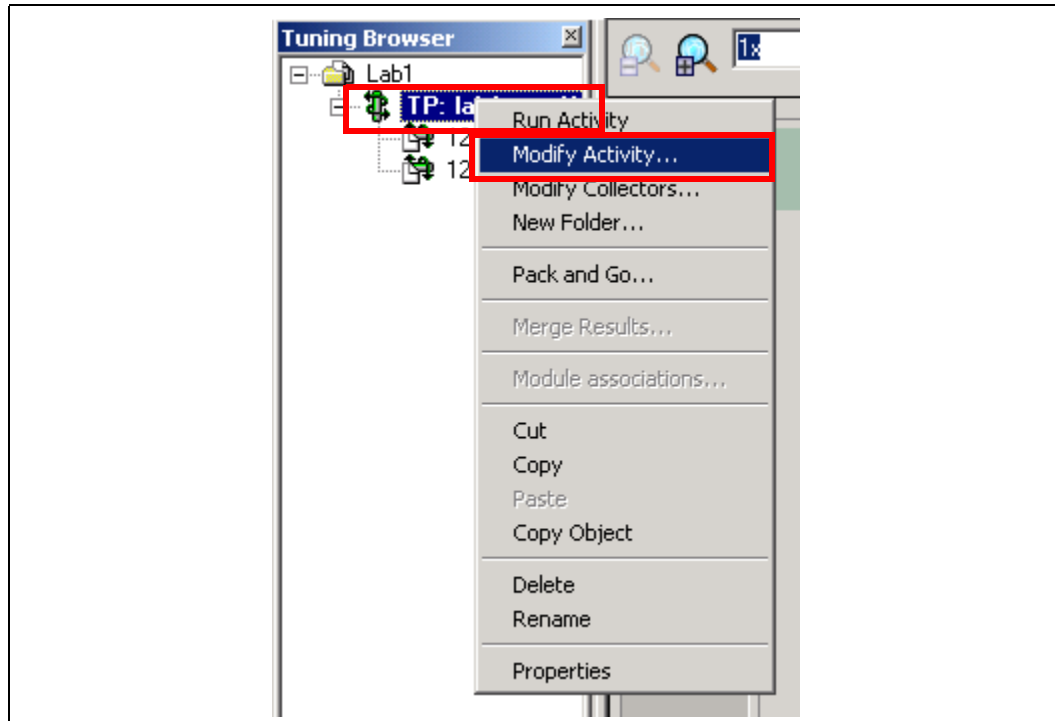
Figure B-7. Specifying Timeline Information Collection Level



Creating a New Activity by Modifying an Existing Activity

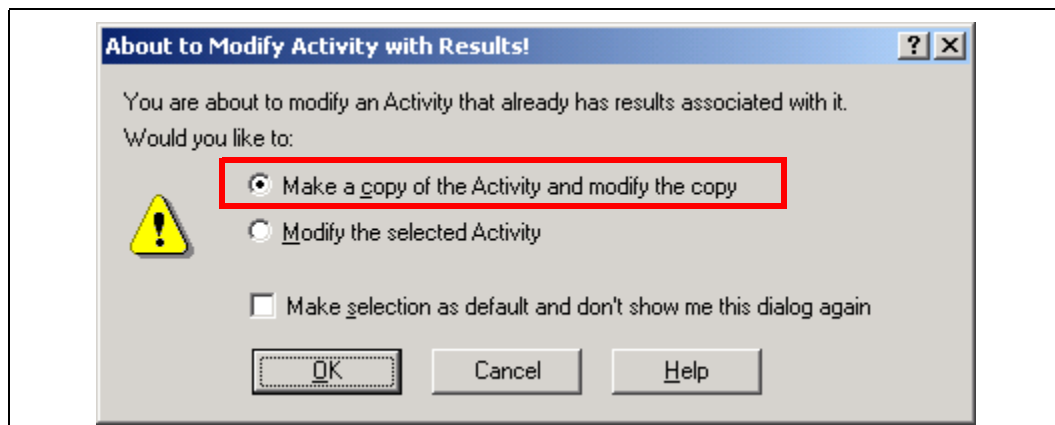
1. Right click on the current activity in the "Tuning Browser" window and select "Modify Activity". (Figure B-8)

Figure B-8. Modifying an Existing Activity



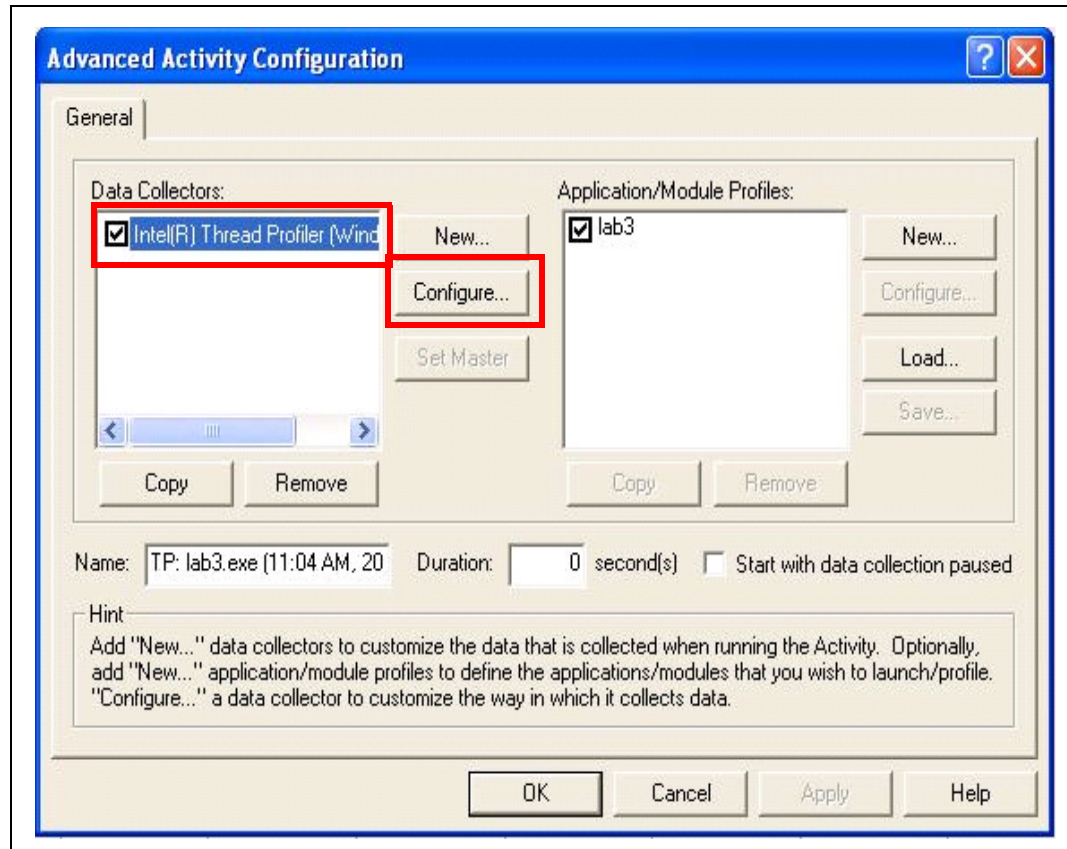
2. The About to Modify Activity pop-up dialog appears as shown in Figure B-9. Ensure that the "Make a copy of the Activity and modify the copy" radio-button is selected.

Figure B-9. About to Modify Activity dialog



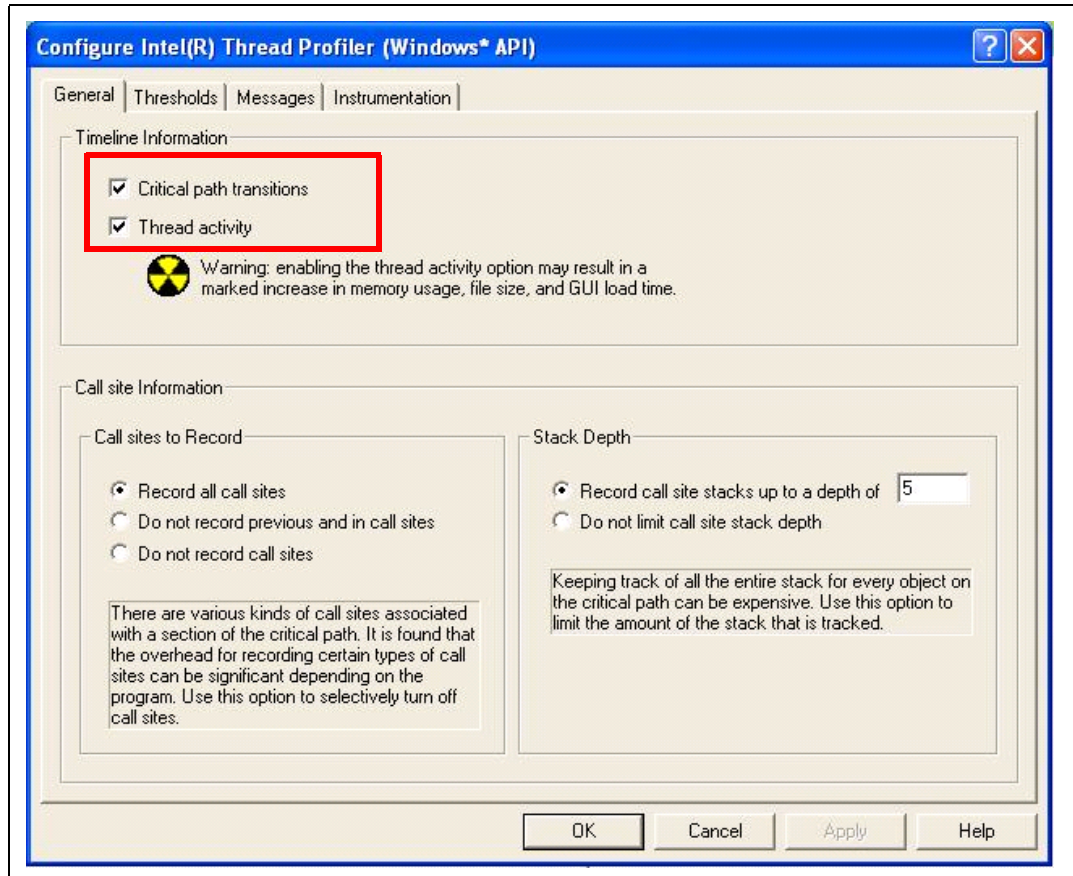
3. Select "Thread Profiler" and click on "Configure" (Figure B-10). In the resulting pop-up dialog, select the "Miscellaneous" Tab.

Figure B-10. Advanced Activity Configuration



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

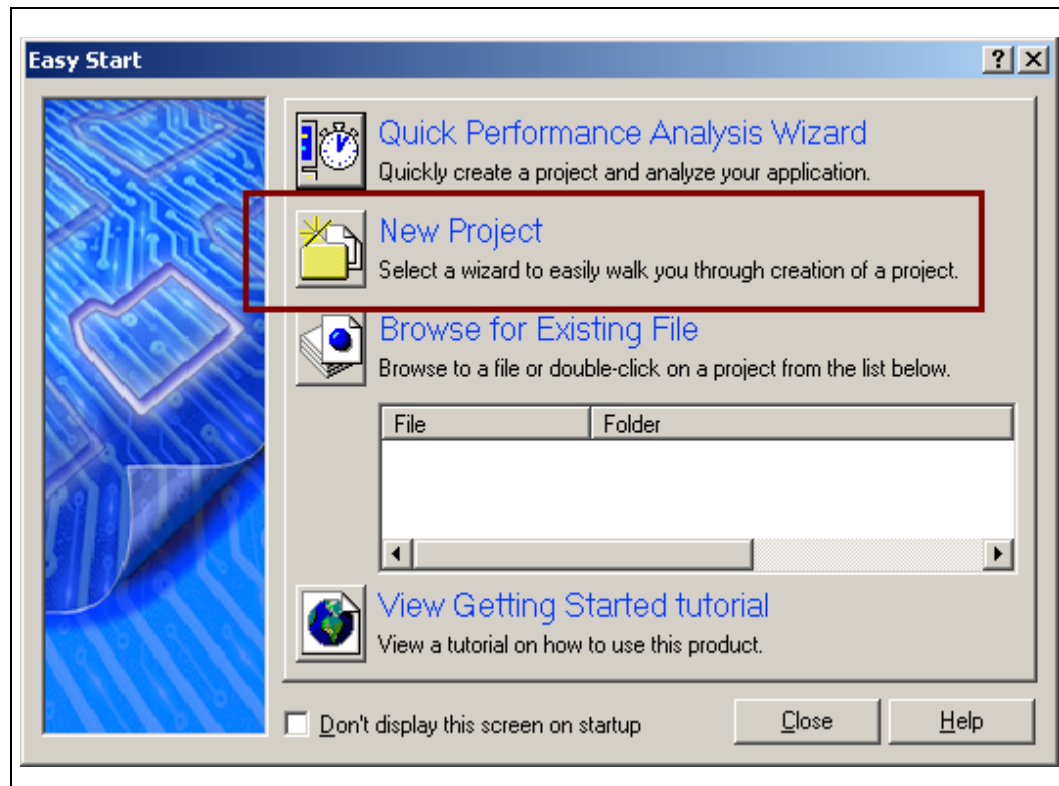


Appendix C: Creating a New Project and Thread Checker RDC Activity

This appendix details the steps needed to create a Thread Checker to be run through Remote Data Collection(RDC) on a Linux system. Before setting up a new RDC activity, ensure that the ITT Server application is running on the remote host.

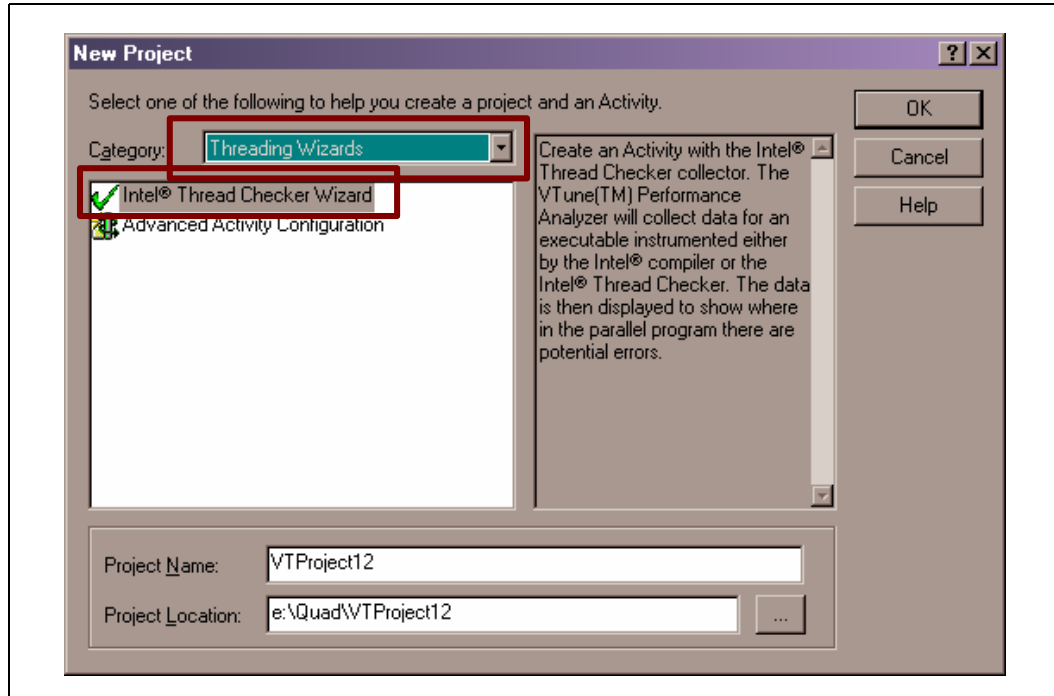
1. Start VTune Performance Analyzer and select New Project, as shown in Figure C-1.

Figure C-1. VTune™ Analyzer Easy Start Menu



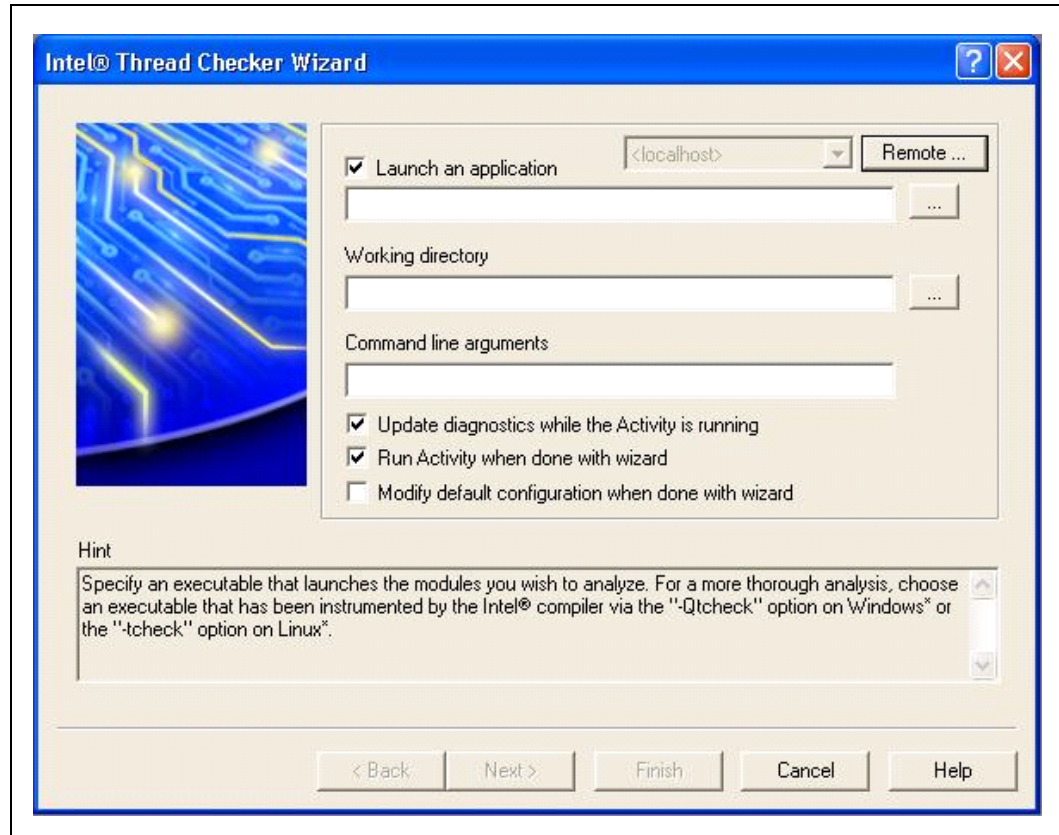
2. From the Category Threading Wizards, select the Intel® Thread Checker Wizard, as shown in [Figure C-2](#).

Figure C-2. Intel® Thread Checker New Project Menu



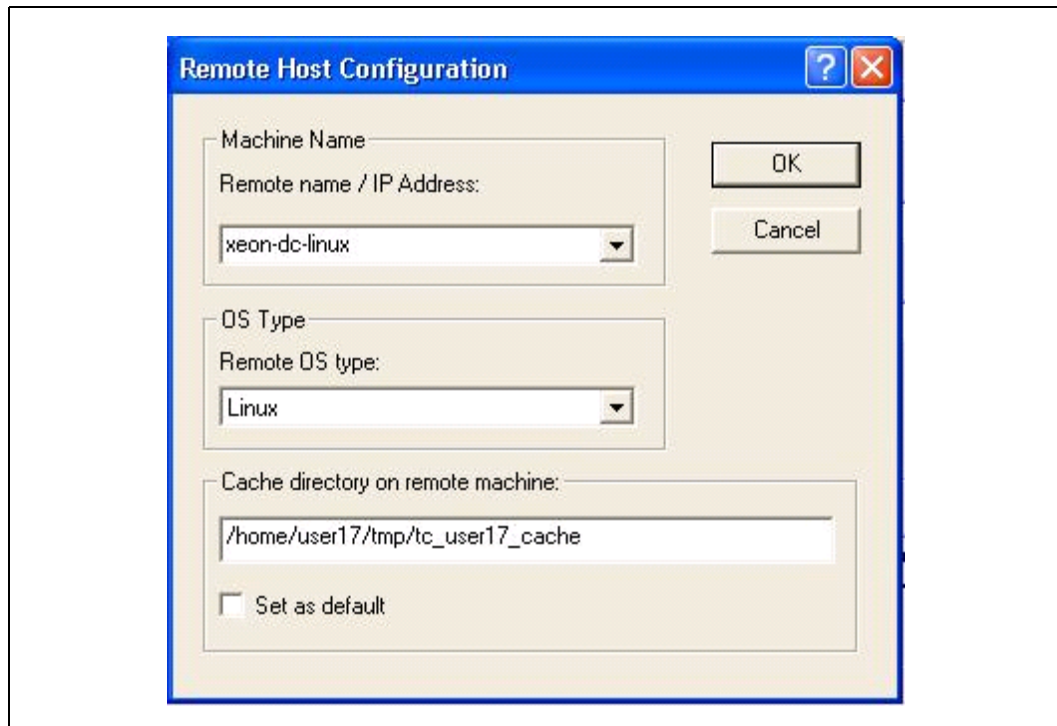
3. Before setting the application to be launched by Thread Checker, click on the Remote... button to specify the remote Linux machine that will be used. (highlighted in red in [Figure C-3.](#))

Figure C-3. Intel® Thread Checker Wizard



The Remote Host Configuration dialog, shown in [Figure C-4](#), displays.

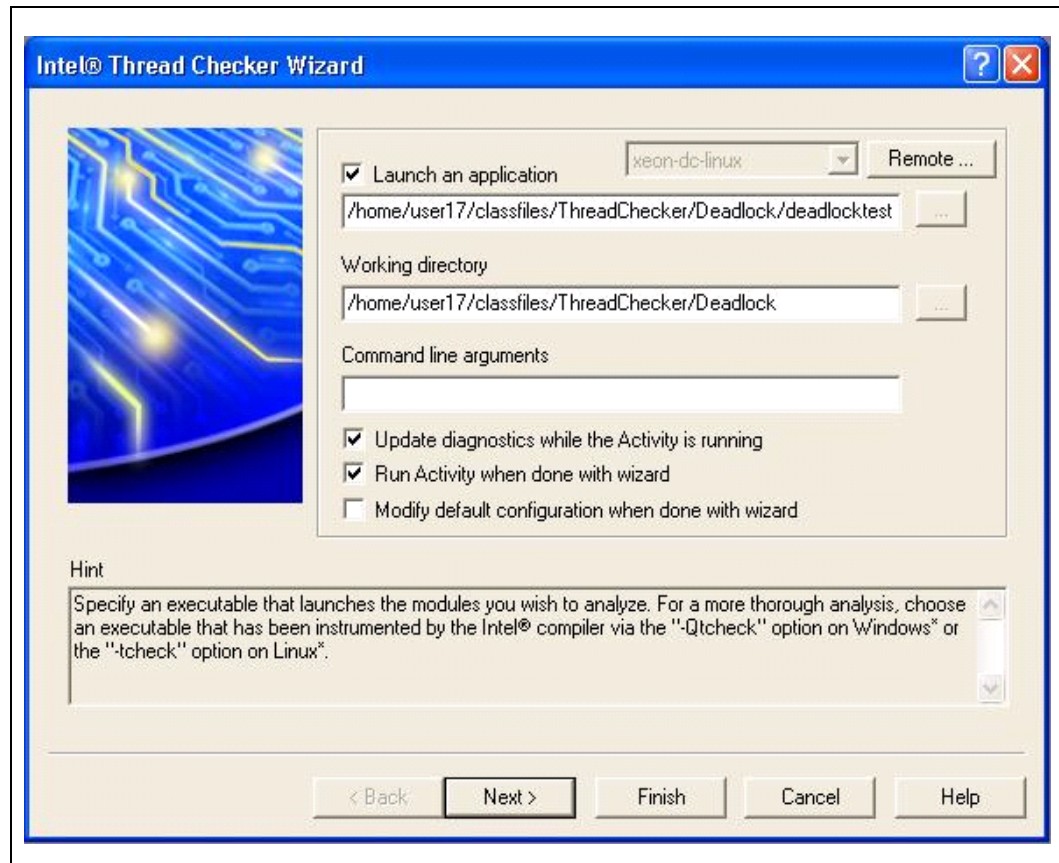
Figure C-4. Remote Host Configuration dialog



4. Enter the name of the remote host or IP address in the Machine Name area. Be sure Linux is the chose Remote OS Type. The cache directory should be some directory on the remote machine that the user has write access. Click OK after entering the relevant information.

5. Enter the full path to the executable to be run remotely in the Intel® Thread Checker Wizard dialog box, as shown in [Figure C-5](#).
The name or IP address of the remote system should appear in the pull-down to the left of the Remote... button. Be sure to put in any command line arguments necessary in the proper entry point in the dialog box.

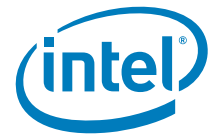
Figure C-5. Intel® Thread Checker Wizard dialog box



6. Click on the Finish button when you have entered all the information and set the check boxes as desired.

If you get any errors, at this point check whether:

- the ITT server is running on the remote system
- the path to the executable is correct
- the binary of the application to be run is available and has the proper permissions to execute



Additional Resources

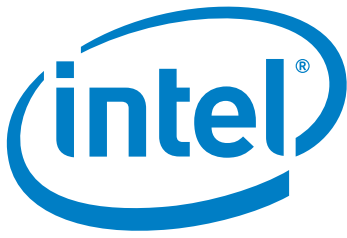
new

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 .
- 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

Copyright © 2006, Intel Corporation. All rights reserved.

□ Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries.

*Other brands and names are the property of their respective owners. □ □