Tuning Threading Code with Intel® Thread Profiler for Explicit Threads
Objectives

After successful completion of this module you will be able to...

- Use Thread Profiler to recognize and fix common performance problems in applications using Windows* threads
Agenda

Look at Intel® Thread Profiler features

Define Critical Path Analysis

Examine Thread Profiler data views available

Review common performance issues of multithreaded applications
  • Focus on Load imbalance
  • Focus on Synchronization contention

Describe general optimizations to gain better performance
Motivation

Developing efficient multithreaded applications is hard

New performance problems are caused by the interaction between concurrent threads

- Load imbalance
- Contention on synchronization objects
- Threading overhead
Intel® Thread Profiler

Plugs in to the VTune™ performance environment

- Instrumentation-based data collector in VTune

Identifies performance issues in OpenMP® or threaded applications using the Win32® API and POSIX® threads

Pinpoints performance bottlenecks that directly affect execution time
Intel® Thread Profiler Features

Supports several different compilers
- Intel® C++ and Fortran Compilers, v7 and higher
- Microsoft* Visual* C++, v6
  - Integrated into Microsoft Visual Studio .NET* IDE

Binary instrumentation of applications

Different views and filters available to assist and organize analysis

Uses critical path analysis
What is the Critical Path?

Threaded applications contain multiple execution flows

- A new flow is created when a thread is created or resumes
- Flow ends when a thread terminates or blocks on a synchronization primitive

The critical path is the longest execution flow
Critical Path Analysis

System Utilization

- Relative to the system executing the application

  **Idle**: no threads
  **Serial**: a single thread
  **Under-subscribed**: more than one thread, less than cores
  **Parallel**: # threads == # cores
  **Oversubscribed**: # threads > # cores

Thread interaction categories

  **Cruise**: threads running without interference
  **Overhead**: thread operation overhead
  **Blocking**: thread waiting on external event
  **Impact**: thread preventing some other thread from executing

**If the critical path is shortened, the application will run in less time**
System Utilization

Examines processor utilization to determine concurrency level of the application

Concurrency is the number of active threads

Categorization shown for a system configuration with 2 processors
Execution Time Categories

Analyze thread interaction and behavior along critical path
Record objects that cause CP transitions

Categorization shown for a system configuration with 2 processors

- Cruise time
- Overhead
- Blocking time
- Impact time

Thread Interaction
Merging Concurrency and Behavior

Start with system utilization

Further categorize by behavior

Concurrency Level | Critical Path | Thread Interaction

- No thread, blocking time
- Serial cruise time
- Serial blocking time
- Serial impact time
- Undersubscribed cruise time
- Undersubscribed blocking time
- Undersubscribed impact time
- Fully parallel cruise time
- Fully parallel blocking time
- Fully parallel impact time
- Oversubscribed cruise time
- Oversubscribed blocking time
- Oversubscribed impact time
- Overhead time
Thread Profiler Views

Critical Path View
• Shows breakdown of the critical path

Profile View
• Shows the breakdown of selected critical paths
• User can select other views of the selected profile
• Concurrency level, threads, objects

Timeline View
• Shows thread activity and critical path transitions for the entire application

Source View
• Transition source view, creation source view
Activity 1a

Threaded version of potential code
• Is there a performance issue?

Goal
• Run application through Thread Profiler
• Examine thread activities by reviewing different views
Critical Path View

Double Click
Let’s look at the threads view. Threads ran simultaneously ~35% of the time.
Profile View – Objects View

Let’s look at Timeline View

This object caused all of the impact
Timeline View
Activity 1b

Threaded version of potential code
  • Is there a performance issue?

Goal
  • Examine thread activities by reviewing different views
  • Determine system utilization
  • Identify any performance issues
Review Activity 1

Concurrency Level view can be used to determine system utilization by the application

Timeline view enables you to understand the thread activity in your application

Instrumentation time will be included in first run results; thus, for applications running in a short amount of time, a second run may produce more realistic timings.
Common Performance Issues

Load balance
- Improper distribution of parallel work

Synchronization
- Excessive use of global data, contention for the same synchronization object

Parallel Overhead
- Due to thread creation, scheduling...

Granularity
- No sufficient parallel work
Load Imbalance

Unequal work loads lead to idle threads and wasted time
Redistribute Work to Threads

Static assignment

• Are the same number of tasks assigned to each thread?
• Do tasks take different processing time?
  • Do tasks change in a predictable pattern?
    • Rearrange (static) order of assignment to threads
  • Use dynamic assignment of tasks
Redistribute Work to Threads

Dynamic assignment

- Is there one big task being assigned?
  - Break up large task to smaller parts

- Are small computations agglomerated into larger task?
  - Adjust number of computations in a task
  - More small computations into single task?
  - Fewer small computations into single task?
  - Bin packing heuristics
Unbalanced Workloads

Threads are unbalanced

Active Times not equal
Activity 2 – Load Imbalance

Threaded version of potential code with thread pools

- Has a load balance performance issue
Review Activity 2

Threads view can be used to determine activity levels of each thread within the application.

Timeline view enables you to understand the thread activity in your application.
Synchronization

By definition, synchronization serializes execution

Lock contention means more idle time for threads

Thread 0

Thread 1

Thread 2

Thread 3

Busy

Idle

In Critical

Time
Synchronization Fixes

Eliminate synchronization

- Expensive but necessary “evil”
- Use storage local to threads
  - Use local variable for partial results, update global after local computations
  - Allocate space on thread stack (`alloca`)
  - Use thread-local storage API (TlsAlloc)
- Use atomic updates whenever possible
  - Some global data updates can use atomic operations (Interlocked API family)
Atomic Updates

Use Win32 Interlocked* functions in place of synchronization object

```c
static int counter;

// Fast
InterlockedIncrement (&counter);

// Slower
EnterCriticalSection (&cs);
    counter++;
LeaveCriticalSection (&cs);
```
Synchronization Fixes

Reduce size of critical regions protected by synchronization object

- Larger critical regions tie up sync objects longer; other threads sit idle longer waiting to acquire objects
- Only accesses to shared variables need to be protected
Synchronization Fixes

Use best synchronization object for job

- Critical Section
  - Local object
  - Available to threads within the same process
  - Lower overhead (~8X faster than mutex)
- Mutex
  - Kernel object
  - Accessible to threads within different processes
  - Deadlock safety (can only be released by owner)

Other objects are available
Object Contention

These four threads...

...are impacting threads by this object
Activity 3

Threaded version of numerical integration
• Has serious performance issues

Goal
• Understand thread activity
• Use the Thread Profiler groupings
• Examine synchronization and its effect on performance
• Fix performance issue
Review Activity 3

Grouping objects and threads provides the information on which objects impact what threads

Apply the heuristics from labs for locating bottlenecks in the source code

For longer running applications, the difference in first and second run-times is negligible
General Optimizations

Serial Optimizations

• Serial optimizations along the critical path should affect execution time

Parallel Optimizations

• Reduce synchronization object contention
• Balance workload
• Functional parallelism

Analyze benefit of increasing number of processors

Analyze the effect of increasing the number of threads on scaling performance
Intel® Thread Profiler for Explicit Threads

What’s Been Covered

Identifying performance issues can be time consuming without tools

Tools are required to understand and to optimize parallel efficiency and hardware utilization

Thread Profiler helps you understand your applications thread activity, system utilization, and scaling performance