Intel® Threading Tools

Paul Petersen, Intel
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

All dates provided are subject to change without notice.

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

Copyright © 2005, Intel Corporation.
Threading?

“You lock too much. You don't lock enough. The program crashes. You try to debug the program. Debugging a threaded program has a lot in common with medieval torture methods, IMHO.”

random quote found via GOOGLE search
Example: Prime Number Generation

```
#include <stdio.h>
const long N = 18;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d \n", N );
    primes[ number_of_primes++ ] = 2; // special case
    for (long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
            primes[ number_of_primes++ ] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```
Example: Prime Number Kernel

```c
for ( long number = 3; number <= N; number += 2 )
{
    long factor = 3;
    while ( number % factor ) factor += 2;
    if ( factor == number )
        primes[ number_of_primes++ ] = number;
}
```

```
Determining primes from 1-18
Found 7 primes
Press any key to continue.
```
Example: Not Quite Right

```c
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d\n", N );
    primes[number_of_primes++] = 2; // special case
    #pragma omp parallel for
    for ( long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
            primes[number_of_primes++] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```
Intel® Threading Tools

Intel® Thread Checker

Pinpoint notorious threading bugs like data races, stalls and deadlocks

http://www.intel.com/software/products/threading/tcwin

Intel® Thread Profiler

• Visualize your threads over time
• Identify synchronization objects that cause delays

http://www.intel.com/software/products/threading/tp
Data Flows

VTune™ Performance Analyzer

Intel® Thread Checker

Binary Instrumentation

Runtime Data Collector

Primes.exe

(Instrumented)

+DLLs (Instrumented)

threadchecker.thr
(results)

Win32* threads, POSIX* threads, OpenMP*

*Third party marks and brands are the property of their respective owners.
Intel® Thread Checker

Memory read of number_of_primes at "2_openmp.cpp":14 conflicts with a prior memory write of number_of_primes at "2_openmp.cpp":14 (flow dependence)

```
long factor = 3;
while ( number % factor ) factor += 2;
if ( factor == number )
    primes[ number_of_primes++ ] = number;

printf( "Found %d primes\n", number_of_primes );
```

```
long factor = 3;
while ( number % factor ) factor += 2;
if ( factor == number )
    primes[ number_of_primes++ ] = number;

printf( "Found %d primes\n", number_of_primes );
```
Example: A Bit Better

```c
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d \n", N);
    primes[number_of_primes++] = 2; // special case
    #pragma omp parallel for
    for ( long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
            #pragma omp critical
            primes[number_of_primes++] = number;
    }
    printf( "Found %d primes\n", number_of_primes);
}
```

Fundamental Concepts

- Thread
- Synchronization Instruction
- Segment
- Precedence Relation between Segments
- Parallel Segments
- Vector Clock of a Segment
- Data Race
Definitions

• Thread
  – An Independent Sequence of Instructions

• Sync Instruction
  – Creates a thread
  – Destroys a thread
  – Posts information about host thread
  – Brings information posted by another thread to host thread

• Segments
  – Parts of a thread separated by sync instructions
Precedence Relation between Segments

Partial Order

S_{11} < S_{12} < S_{13}
S_{21} < S_{22} < S_{23}
S_{31} < S_{32}
S_{21} < S_{12} < S_{23}
S_{11} < S_{23}
Parallel Segments

Two segments $S$ and $S'$ are parallel, if $S < S'$ and $S' < S$ are both false.

Examples:
$S_{11}$ and $S_{21}$
$S_{11}$ and $S_{22}$
$S_{11}$ and $S_{32}$
$S_{22}$ and $S_{31}$.
Vector Clock of a Segment

- Vector clocks represent numerically the precedence relation between segments.
- Vector Clock $V_S$ of a segment $S$ is a function on threads with values in $\{0, 1, 2, \ldots\}$.
- Take a segment $S$ on a thread $T$. Take a thread $T'$ known to $S$. Then $V_S(T')$ is one more than the number of segments $S'$ on $T'$ such that $S' < S$.
- Vector clock of a segment is computed from the vector clocks of its immediate predecessors.
Conditions in Terms of Vector Clocks

Let $S$ denote a segment on a thread $T$ and $S'$ a segment on a thread $T'$. 

$S < S'$ iff $V_S(T) < V_{S'}(T)$. 

$S$ and $S'$ are parallel iff

$V_S(T) \geq V_{S'}(T)$ and $V_{S'}(T') \geq V_S(T')$. 

- 16 -
Data Race

• There is a data race between two segments S and S’ if
  – The segments are parallel;
  – There is a location x in shared memory that both segments access, and at least one of the accesses is a “write.”
Data Race Detection

• Basic Result:
  
x: A location in shared memory
  S: A segment that has accessed x
  S’: A segment that accesses x after S.
  There is a race in x, if
  - $V_S(T) \geq V_{S'}(T)$
  - x is written by either S, or S’, or both.
Summary : Thread Checker

- The Intel® Thread Checker provides a safety net for developers
  - Increases confidence in developers ability to debug threaded software.
- The techniques used by the Thread Checker allow analysis of “production” applications.
- Industry direction moving to multi-core solutions makes confidence tools even more critical to success.
Intel® Thread Profiler

• The Thread Profiler consists contains two modes of analysis
  – OpenMP analysis
    • Single-level fork-join parallelism
  – Native Threading analysis
    • Arbitrary concurrency
Thread Profiler: OpenMP
Thread Profiler: Native threads

- Thread Profiler motivation
- Understand Threading Behavior
- Optimize turnaround performance
  - Serial Optimizations:
    - learn where to perform optimizations on code that will most likely affect program run-time
  - Parallel Optimizations:
    - Target concurrency level to fully utilize processing resources
    - Thread blocking time is not always bad
Product Status

• 1.0 – OpenMP Profiler only
• 2.0 (2003)– Windows Threads
• 2.1 (2004)– Linux pthreads (via RDC)
• 2.2 (2005)– 32-bit app support on em64t / incremental improvements
Tool Components

- User App
- Binary or Compiler Instrumentation
- Runtime Library
- Instrumented App
- Thread Profiler
- Profile Data
- Iterative Optimization Process
Analyzer Concepts
System APIs

– Thread and Process Control APIs
  • Create, Terminate, Suspend, Resume, Exit

– Synchronization APIs
  • Mutexes, Critical Sections, Locks, Semaphores, Thread Pools, Timers, Messages, APCs, Events, Condition vars

– Blocking APIs
  • Sleeping, Timeouts
  • I/O: Files, Pipes, Ports, Messages, Network, Sockets
  • User I/O: Standard, GUI, Dialog Boxes
User Synchronization

• Thread Profiler provides an API for users to instrument user synchronization.

```c
__itt_notify_sync_prepare( &spin );
while( wait for spin ) {
  if( timeout ) {
    __itt_notify_sync_cancel( &spin );
    return;
  }
}
__itt_notify_sync_acquired( &spin );
do stuff;
__itt_notify_sync_releasing( &spin );
release spin;
```
Instrumentation

- Usually low run-time overhead because only select events are instrumented
- Target overhead of less than 2x slowdown with reasonable synchronization
- Common case is much less

**RECORDED EVENTS**

- Create Thread (Fork)
- Thread Entry
- Wait for synchronization object or event
- Acquire synchronization object or event
- Release or signal synchronization object or event
- Wait for external event
- Receive external event
- Thread Exit

Diagram showing the timeline of events for threads T1, T2, and T3, with specific events such as creating threads, entry, synchronization acquisition and release, and thread exit.
Uncontended Events

- i.e. Thread acquires lock that no one else holds. It continues without any pause.
- Thread Profiler will not consider these transactions.
Contended Transitions

- i.e. Thread attempts to acquire lock that another thread holds. It must wait to acquire it before continuing.
Thread Blocking

- Thread waits for external event such as user I/O, file operation, another process, &c.
Execution Flows

• Flow **splits** when a thread **creates** a new thread or **signals (unblocks)** another thread

• Flow **ends** when a thread **waits** for another thread or **terminates**
Critical Path Analysis

- The **continuous flow to target location** is the **critical path**
- Default target is the program termination
- Where to focus optimization energy
- Next, we will “color” the critical path to offer more information
Transition Overhead Time

- Indicate the time spent between one thread signaling and another thread receiving the signal. This is marked as overhead time.
Processor Utilization

- Measure processor utilization so user can see how parallel their program really is.
- Concurrency is the number of threads that are active (not waiting, sleeping, blocked, &c.) at a given time.

(Example reflects 2 processor machine)
Characterize parallel behavior of Critical Path

- Associate spans of time with the objects that caused the CP transitions
- This provides which objects/barriers cause parallel scalability problems that are directly affecting execution time to critical path target.

**Cruise Time:** Time a thread does not delay the next thread on the CP

**Overhead (transition) Time:** Threading synchronization or OS scheduling overhead

**Blocking Time:** Time a thread spends waiting for an external event or blocking while still on the CP (includes timeouts)

**Impact Time:** Time a thread on the CP delays next thread from getting on the CP
Combine concepts for Profile

- Start with the CP
- Mark overhead
- Break it down by Utilization
- Further categorize by behavior

(Example reflects 2 processor machine)
Analyze and Optimize Code

• Serial Optimizations
  – Serial optimizations along the critical path should affect execution time.

• Parallel Optimizations
  – Red/Orange: Look to locations where the concurrency level is less than optimal
  – Cruise, Blocking, Impact: Try increasing the level of parallelism
  – Blocking time: Try reducing wait for external object
  – Impact time: This represents parallel imbalance or synchronization object contention

• Analyze benefit of increasing number of processors
Case Study
Case Study: simple data decomposition

- Master thread does:
  - initial work
  - creates 4 worker threads and 1 verifier thread
  - waits for them to finish
- Worker threads work in parallel
- Verifier thread verifies each work segment
Parallel Optimizations
Step 1
Understand Threading Behavior
Critical Paths View

- Able to compare multiple runs of the same or different applications
- Colors quickly give general idea of program behavior
• Gain an overview perspective on thread behavior and CP flow
• Notice master thread forking 4 worker threads and 1 verifier thread
• Notice imbalance in top example
View all transitions

- By default, 2.2 will only record transitions on the critical path.
- Set “TP_PREVIEW” environment variable to view details of all contended transitions.
Step 2
Select Data set
Change Critical Path Target

- If the end of the application is not target of interest, then enable technology preview features with “TP_PREVIEW=1”
- This allows you to specify an arbitrary location for the critical path target
Time-based Filtering

- Select time region of interest. Focus on processing time, or at least ignore startup time.
Step 3
Analyze Results
Profile View

- The Critical Path is represented as a stacked histogram
- Data may be divided up in different ways
  - Concurrency
  - Time thread is on CP
  - Objects causing contention/imbalance for CP
  - Source locations of waiting for the CP
Concurrency Profile

(example reflects 4 processor machine)

- Understand how parallel your application really is, how well is it utilizing the machine
- Impact time means there is contention or imbalance preventing a higher concurrency level
- Focus on underutilized time. Demonstrates OK vs. Bad impact time
Threads Profile

• Focus on undersubscribed time
• See which threads are on the Critical Path preventing parallelism
• *Halos* represent total thread lifetime and active thread lifetime

Undersubscribed imbalance Fixed!
Threads Profile

- Focus on undersubscribed time
- This startup time could have been filtered out of results
- Let’s assume we can’t easily parallelize this work
Threads Profile

- Focus on undersubscribed time
- Verifier thread still has serial time
- Let's filter and group the bar by synchronization object
Object Profile

- See which synchronization object are causing imbalance or contention
- Now let’s view the source code for one of these objects

Let’s show transition source view

Tooltips provide object information

Semaphore 84
Creator = 1
lifetime = 1.55235
Create loc: h:\win\tsys\complex\diihil\ssg2003_dir.c 63
Close loc: h:\win\tsys\complex\diihil\ssg2003_dir.c 171
Step 4
Modify Application
Transition Source Locations

• Current thread transitions **out** to the next thread on the critical path.

• **next** thread receives the signal.
Source View

- View previous, in, out, and next callsites associated with this transition.
- Traverse the callstack for each one
- Directly view the source code to see where code optimizations should be made.
Step 5
REPEAT
Verify and Compare New Results

- Critical Paths View
- Able to compare multiple runs of the same or different applications
- Colors give general idea of program behavior
• **Serial** impact time has become **parallel** time
• Total bar heights (thus total run-time) is less
Serial Optimizations
Sampling Integration

• Use Critical Path to focus serial optimization energy on code that will more likely affect program run time.
  1. Enable both TP and Sampling collectors in VTune
  2. From TP Timeline view select a region of time:
     • View sampling for specified region(s)
     • View sampling for specified region(s), but just samples on Critical Path

• Also useful to gain context of what threads are doing at source locations other than contended transitions.
Summary : Thread Profiler

- Use to profile and optimize threaded behavior in an iterative process
- Thread Profiler collects utilization and critical path; correlates with synchronization objects, threads, and source
- GUI allows you to slice and dice profile data and view the corresponding source code