

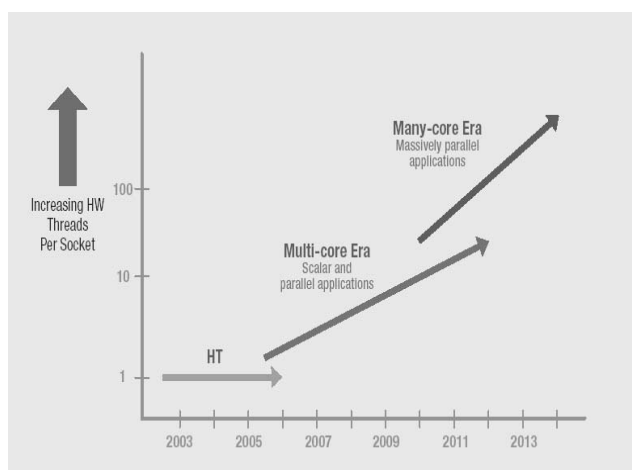


## Improving Software Performance and Correctness with Intel® Threading Tools

Intel  
Software and Solutions Group (SSG)  
Developer Products Division (DPD)



## Intel® Processor and Platform Evolution for the Next Decade



2

Source: "Platform 2015: Intel® Processor and Platform Evolution for the Next Decade"



## Paths to taking advantage of Multi-core Processors

- **Do Nothing:**

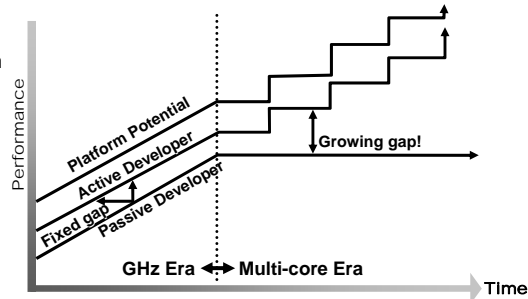
- Background tasks benefit from more compute resources
- Limited potential for single-application performance

- **Process-level parallelism**

- Can be cumbersome to work on a shared data set

- **Application threading for performance:**

- Use native threads or threading abstraction libraries
- OpenMP is a cross-platform standard useful for quickly parallelizing with domain decomposition
- Intel software tools can aid developer in efficiently threading



3



## A Generic Development Cycle



### Analyze

- VTune™ Performance Analyzer

### Design (Introduce Threads)

- Intel® Performance libraries: IPP and MKL
- OpenMP\* (Intel® Compiler)
- Explicit threading (Win32\*, Pthreads\*)
- Intel® Threading Building Blocks

### Debug for correctness

- Intel® Thread Checker
- Intel® Debugger

### Tune for performance

- Intel® Thread Profiler
- VTune™ Performance Analyzer



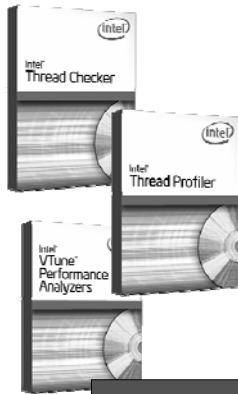
4

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



## Multi-core Software Tools

How can we help more developers use parallelism?



**Intel® Thread Checker**  
pinpoints latent threading errors

**Intel® Thread Profiler**  
Insight into threaded application/lock  
level performance

**Intel® VTune™ Performance Analyzer**  
Insight into system level performance

We help make it easier



5

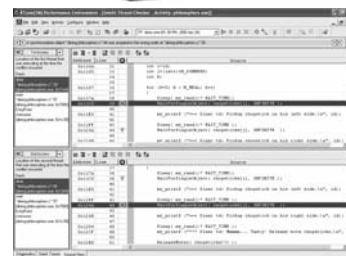


## Intel® Thread Checker

*Create Threads Faster*

### Key Features

- Detects challenging data races and deadlocks
- Pinpoints errors to the source code line
- Works on standard debug builds without recompiling
- Recommends modules to instrument by usage (windows\* product)
- Scriptable interface for test environment integration
- Supports 32 and 64-bit applications



\* Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.



6

Intel Confidential – NDA Required



## Thread Checker: Overview

### Features & Benefits

- Pinpoint the function, context, line, variable, and call stack in the source code to aid analysis and repair of bugs
- Identify nearly impossible-to-find data races and deadlocks using an advanced error detection engine
- Instrumental for effective design of threaded applications
- Errors do not need to actually occur to be detected

### Platforms

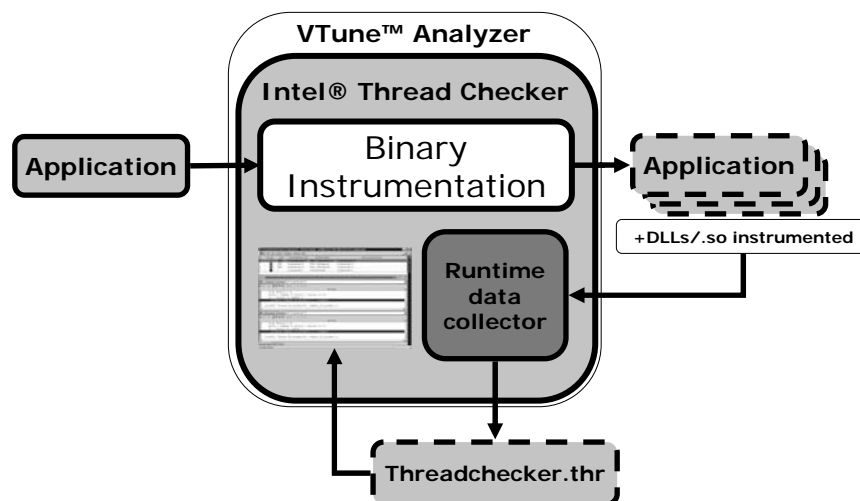
- Supports Windows\* threads or OpenMP\* applications on Windows for IA32/EM64T
- Supports POSIX\* threads or OpenMP\* for applications on Linux for IA32/EM64T/IPF from a Windows host
- Command-line-only version for Linux is in beta



7



## Thread Checker Phases



8



## Thread Checker Analysis

Dynamic analysis as software runs

- Data (workload)-driven execution
- If code path not executed, no analysis of path

Includes monitoring of:

- Thread and synchronization API's used
- Thread execution order
  - Scheduler impacts results
- Memory accesses between threads



9



## Thread Checker: Before You Start

Instrumentation: Background

- Adds calls to library to record information
  - Thread and synchronization API's, memory accesses
- Increases execution **time** and **size**

Use *small* data sets (workloads)

- Execution time and space is **expanded**
- Multiple runs over different paths yield best results

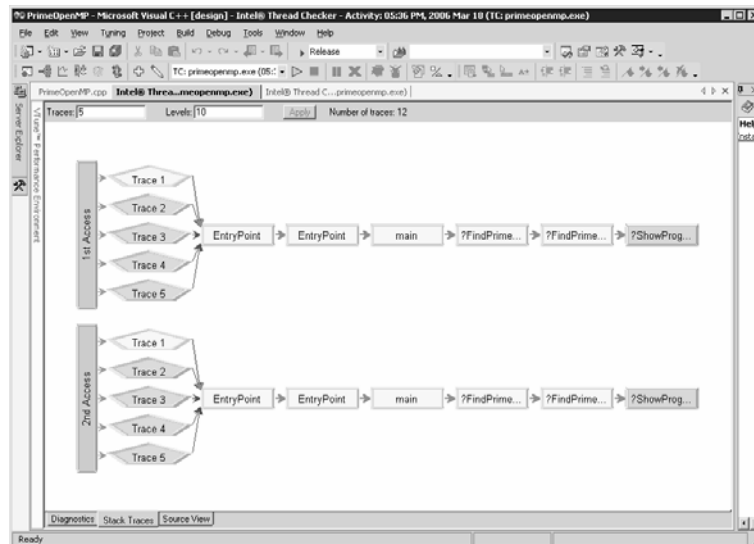
**Workload selection is important!**



10



## Thread Checker Views



11



## Thread Checker Command-Line

```

Intel(R) C++ Compiler 9.0.022 build environment for 32-bit applications
N:\PrimeOpenMP\Release>tcheck_cl PrimeOpenMP.exe 1 100
Intel(R) Thread Checker and Intel(R) Thread Profiler command line instrumentation driver.
Copyright (c) 2006 Intel Corporation. All rights reserved.
Building project
Running: C:\Documents and Settings\otovink\My Documents\Presentations\Parallel...
Running PrimeOpenMP\Release\PrimeOpenMP.exe 1 100
Verifying new module: C:\WINNT\system32\advapi32.dll
Verifying new module: C:\Program Files\Intel\VTune\Analyzer\bin\libittnotify.dll
100%
25 primes found between 1 and 100 in 0.11 secs
Application finished
pFileSB->FindSource return $FALSE for rva 5808 - no SourceLocation
ID|Short|Severity|Count|Context|Description|1st Acc|2nd Acc|
|:-----|:-----|:-----|:-----|:-----|:-----|:-----|:-----|
|ID|Short|Severity|Count|Context|Description|1st Acc|2nd Acc|
|:-----|:-----|:-----|:-----|:-----|:-----|:-----|:-----|
11|Read|Error|12|Prime0|Memory write at "PrimeOpenMP!Prime0|Prime0|
|penMP.c|cpp":77 conflicts with a pr|penMP.c|penMP.c| |
|te data|pp":74|for memory read at "PrimeOpe|pp":77|pp":77|
|race|inMP.c|pp":77|<anti dependence|
|)|
12|Write|Error|12|Prime0|Memory read at "PrimeOpenMP!Prime0|Prime0|
|penMP.c|cpp":77 conflicts with a pr|penMP.c|penMP.c| |
|id data|pp":74|for memory write at "PrimeOpe|pp":77|pp":77|
|race|inMP.c|pp":77|<flow dependence|
|)|
13|Write|Error|12|Prime0|Memory write at "PrimeOpenMP!Prime0|Prime0|
|penMP.c|cpp":77 conflicts with a pr|penMP.c|penMP.c| |
|te data|pp":74|for memory write at "PrimeOpe|pp":77|pp":77|
|race|inMP.c|pp":77|<flow dependence|
|)|

```

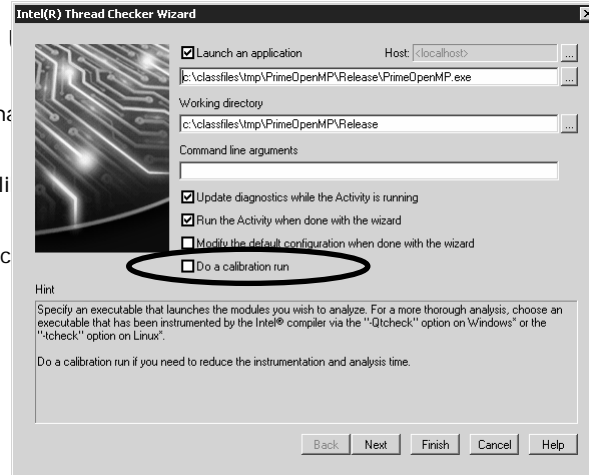
Thread Checker can be integrated into automatic test system



## Getting Applications to Run

Solution -

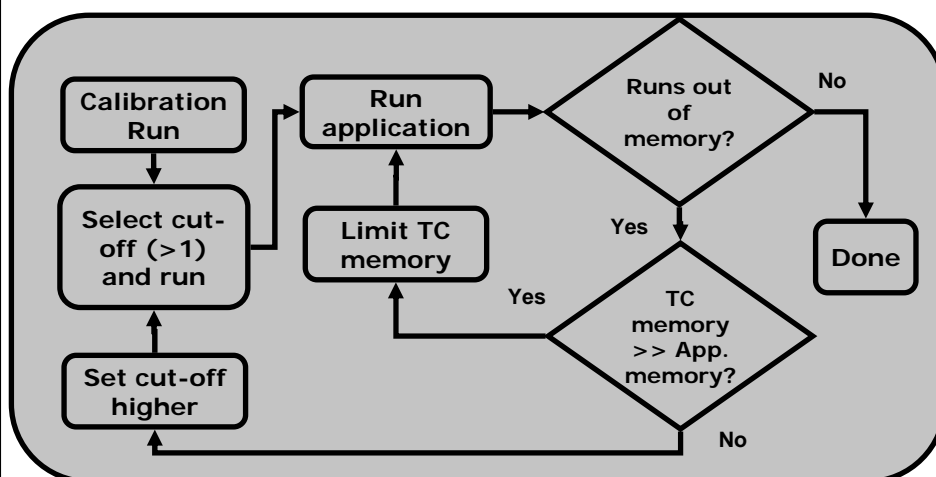
- **Step 0:**
  - Use sma
- **Step 1:**
  - Use cali
- **Step 2:**
  - Select c



13



## Selective Instrumentation Using Calibration



14



## Dealing with High Diagnostics Count

Where do you begin debugging?

Are all the diagnostic messages equally important/serious?

Steps:

- Add "1st Access" column
- Group by "1st Access"
- Sort by "Short Description" column



15



## Dealing with High Diagnostics Count

main.cpp | PrimeDLL.h | PrimeDLL.cpp | Thread Checker... (using dll.exe)

1st Access	ID	Short Description	Description
"main.cpp":7	5	Write -> Write data-race	Memory write of Unknown at "main.cpp":80 conflicts with a prior memory write of Unknown at "main.cpp":80 (output dependence)
"main.cpp":7	4	Write -> Read data-race	Memory read of Unknown at "main.cpp":80 conflicts with a prior memory write of Unknown at "main.cpp":80 (flow dependence)
Group 4: "main.cpp":82 (Diagnostics: 1; Filtered: 0)			
"main.cpp":7	3	Read -> Write data-race	Memory write of Unknown at "main.cpp":80 conflicts with a prior memory read Unknown at "main.cpp":82 (anti dependence)
Group 5: "primedll.cpp":51 (Diagnostics: 2; Filtered: 0)			
"primedll.cpp":10		Write -> Write data-race	Memory write of Unknown at "primedll.cpp":55 conflicts with a prior memory write of Unknown at "primedll.cpp":51 (output dependence)
"primedll.cpp":9		Write -> Read data-race	Memory read of Unknown at "primedll.cpp":55 conflicts with a prior memory write of Unknown at "primedll.cpp":51 (flow dependence)
Group 6: "primedll.cpp":55 (Diagnostics: 5; Filtered: 0)			
"primedll.cpp":1		Write -> Write data-race	Memory write of Unknown at "primedll.cpp":51 conflicts with a prior memory write of Unknown at "primedll.cpp":55 (out
"primedll.cpp":13		Write -> Write data-race	Memory write of Unknown at "primedll.cpp":55 conflicts with a prior memory write of Unknown at "primedll.cpp":55 (output dependence)
"primedll.cpp":12		Write -> Read data-race	Memory read of Unknown at "primedll.cpp":55 conflicts with a prior memory write of Unknown at "primedll.cpp":55 (flow dependence)
"primedll.cpp":2		Read -> Write data-race	Memory write of Unknown at "primedll.cpp":51 conflicts with a prior memory read Unknown at "primedll.cpp":55 (anti dependence)
"primedll.cpp":11		Read -> Write data-race	Memory write of Unknown at "primedll.cpp":55 conflicts with a prior memory read Unknown at "primedll.cpp":55 (anti dependence)
Group 7: "primedll.cpp":60 (Diagnostics: 1; Filtered: 0)			
"primedll.cpp":17		Read -> Write data-race	Memory write of Unknown at "primedll.cpp":55 conflicts with a prior memory read Unknown at "primedll.cpp":60 (anti dependence)
Group 8: Unknown (Diagnostics: 2; Filtered: 0)			
Unknown	21	Thread termination	Thread info at Unknown - includes stack allocation of 1048576 and use

Diagnostic groups: 1-7

Number of occurrences: 0-5

Legend: Unclassified, Remark, Information, Caution, Warning, Error, Filtered



16





## Thread Checker Summary

- Thread Checker functions as a design, debugging, and quality aid
  - Automatically detects hard-to-find data races and deadlocks in multi-threaded applications
- Workload selection is extremely important for successful runs
  - Default configuration works well for unit tests; larger fully integrated applications require the use of calibration run

**Reduce time to market for threaded applications by speeding up the development process**



17



## Intel® Thread Profiler 3.0 for Windows\*

*Optimize Threads Faster*

### Key Features:

#### Understand Threading Behavior

- View potential core utilization

#### Optimize Threading Performance

- Fully utilize available cores
- Identify which synchronization objects are contended and which waits actually affect performance
- Highlight workload imbalance
- Pinpoints issues to the source code

### Supported Environments:

- Supports 32 and 64-bit applications
- Native thread-API on Microsoft Windows\* (Win32 Threads)
- Native thread-API on Linux\* (PThreads)
- OpenMP\* threads
- Intel's new parallel programming model
- Graphical visualization available on Windows\*

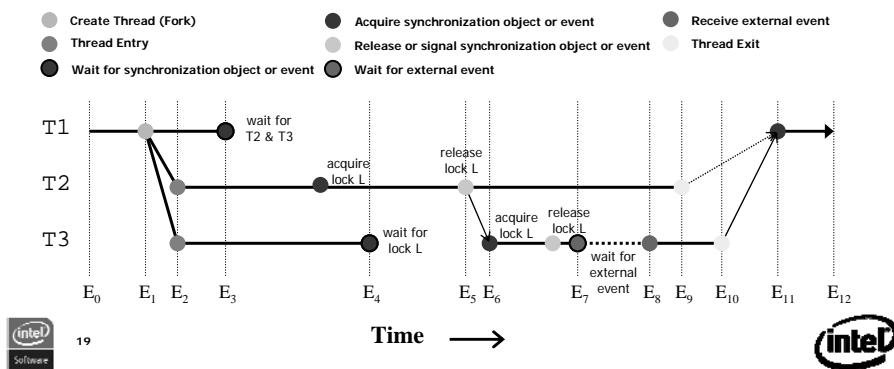


18



## Thread Profiler Execution Flow

- Thread Profiler instruments system API calls
- Run-time engine intercepts API calls and records events when instrumented application is running
  - calling thread ID, time, duration, sync object ID, call-stack
- In the end of the program run TP data file gets produced
- Thread Profiler GUI visualizes threads behavior



## System APIs – Windows\* Threads and POSIX\* threads

- Thread and Process Control APIs
  - Fork, Create, Terminate, Suspend, Resume, Exit
- Synchronization APIs
  - Mutexes, Critical Sections, Locks, Semaphores, Thread Pools, Timers, Messages, APCs, Events, Condition Variables
- Blocking APIs
  - Sleeping, Timeouts
  - I/O: Files, Pipes, Ports, Messages, Network, Sockets
  - User I/O: Standard, GUI, Dialog Boxes

\* POSIX\* threads API is supported for Linux apps



20



## User Synchronization

If custom synchronization is used, Thread Profiler provides an API for users to instrument user synchronization.

```
__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;
```



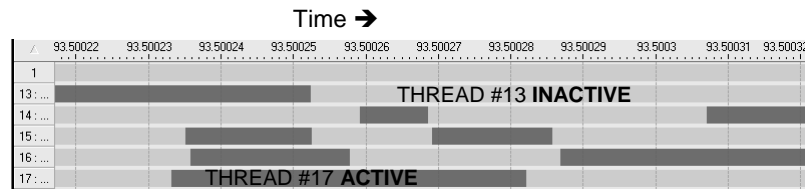
21



## Thread Profiler: Terminology

## Thread Activity

- Types of thread activity
  - Thread active: runs or ready to run
  - Thread inactive: waits for sync object, blocks on external event
- Thread Profiler view



- Rationale
  - You don't want to have too many threads active and competing for cores
  - You want to keep all cores busy with work when number of active threads equals to number of cores
  - Thread Profiler collects call-stack for wait and blocking APIs – you can locate the problem area in the source file

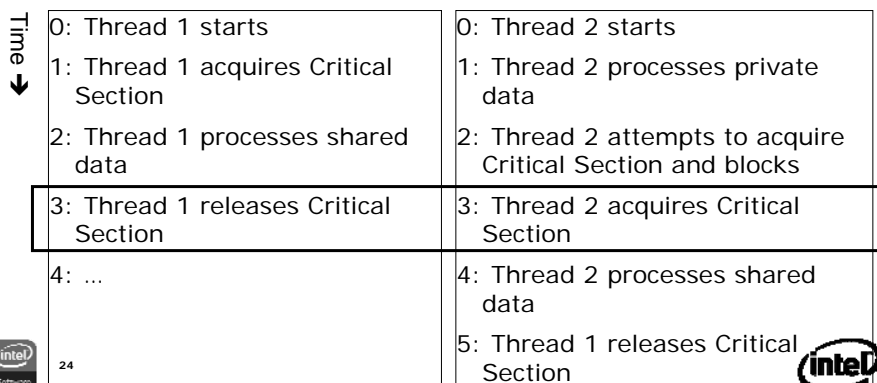


23



## Transitions

- Represents “signal” sent by one thread to another by releasing synchronization object
- Attributes:
  - Signaling and receiving thread IDs
  - Overhead – time spent between “send” and “receive” events
  - “Send” and “receive” events call-stacks

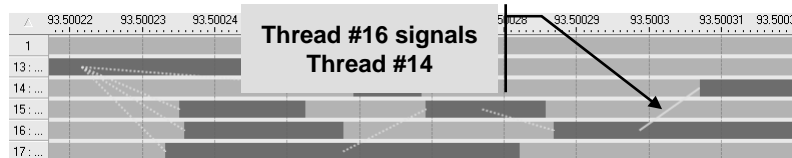


24



## Transitions (cont.)

- Types of transitions
  - Contended: receiving thread had to wait for the signal
  - Uncontended: thread acquired the sync object without contention
- Thread Profiler view



- Rationale
  - Thread Profiler collects call-stack for transitions – you can locate the problem area in the source file
  - Thread Profiler attributes transition overhead to the specific sync object – you can find the most expensive sync objects
  - Thread Profiler visualize transitions – you can spot the area with the most excessive synchronization and focus your analysis effort on it

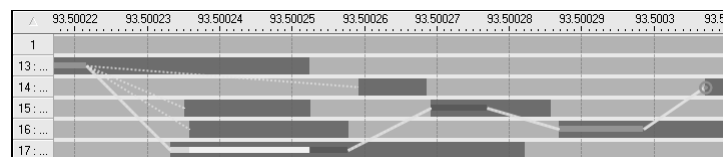


25

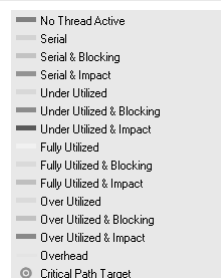


## Critical Path

- Definition
  - Longest execution flow; emphasizes segments of the threaded program that are worth optimizing
  - Characterized behavior of active threads
- Thread Profiler view



- Rationale
  - Useful for throughput analysis
  - Helps understand threads behavior



26



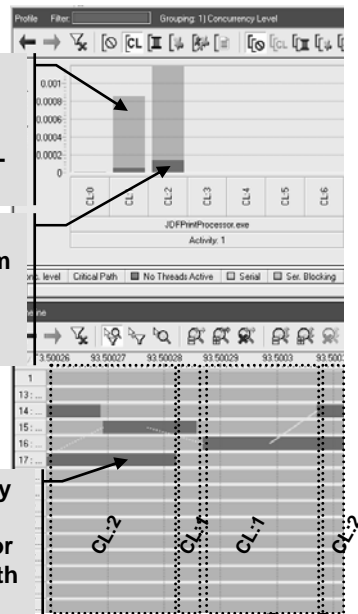
## Concurrency Level (CL)

- Definition
  - Number of active threads
- Rationale
  - Thread Profiler summarizes threads activity information and groups it by concurrency levels – you can understand overall CPUs utilization
  - Thread Profiler Timeline and Profile view are synchronized – you can zoom-in and filter data and get from summary view into detailed view
  - CL grouping (like any other view) may be combined with critical path data – you can view threaded behavior at a higher level

Wait time sum for all waiting threads while running single-threaded

Active time sum for all running threads

Thread activity times are summed up for the regions with equal CL



27



## Groupings and Filtering

- Profile View
  - Grouping by Concurrency Level
  - Grouping by Threads
  - Grouping by Objects
  - Grouping by Source Locations
- Timeline View
  - Manual filtering of any region of interest
  - Synchronized automatic filtering: double-click on any bar at Profile View and Timeline data gets filtered



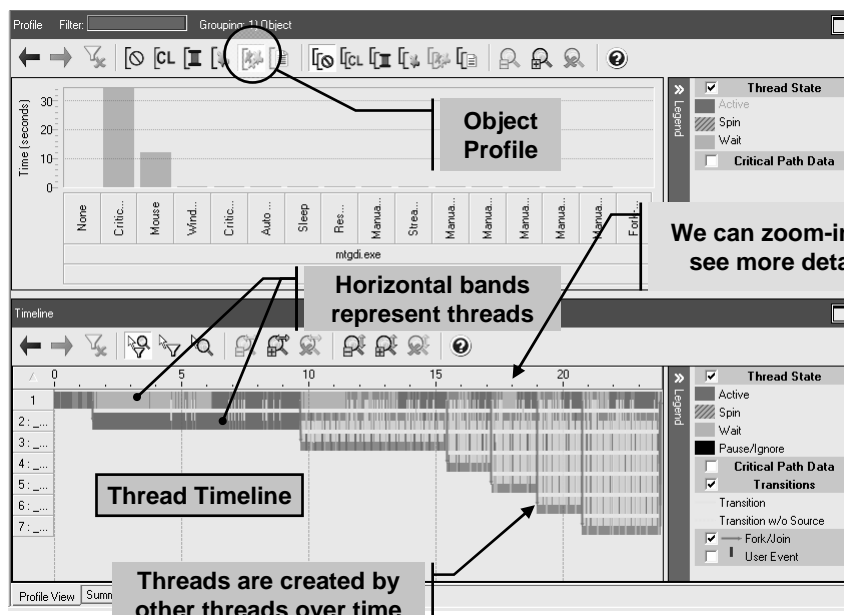
28



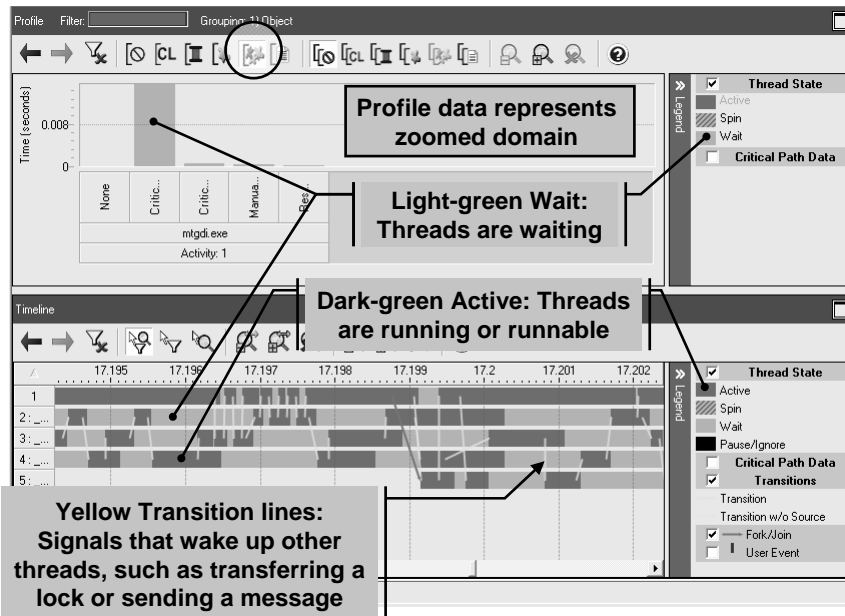


## Understand Threading Behavior

### Object Profile and Thread Timeline

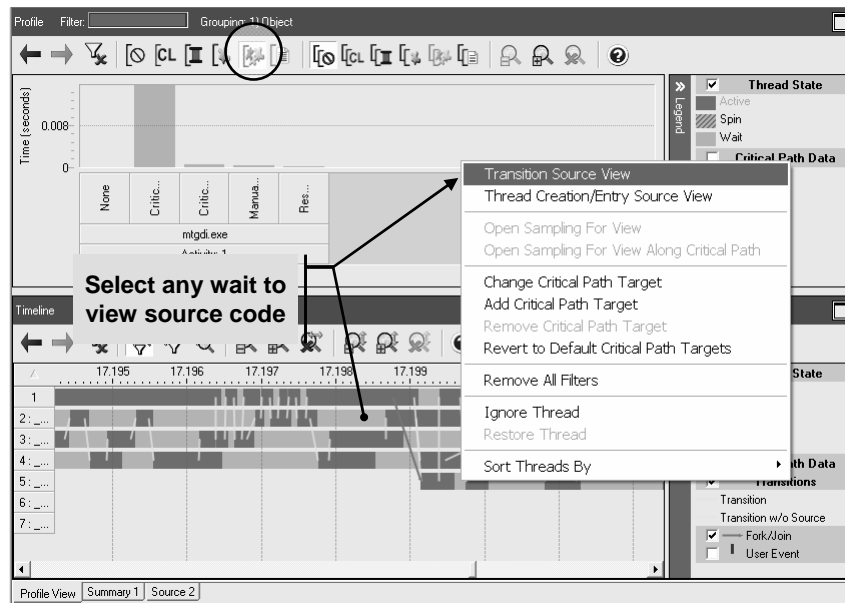


## Zoom-in for more detail



31

## View source for any thread wait



32



## Synchronization source view



The screenshot displays the Synchronization Source View in a debugger, showing two panels: 'Release critical section' and 'Acquire critical section'.

**Release critical section:** This panel shows the source code for the 'Release critical section' function. The 'Sync Object' is 'Critical Section 52'. The 'Stack' shows the following addresses and lines:

Address	Line
0x420B	229
0x4211	231
0x421C	232
0x421D	233
0x421E	234
0x421F	235
0x4220	236
0x4221	237

The source code for the 'Release critical section' function is shown on the right, with the following lines highlighted:

```

// threads
// other th
// I mean,
// watch the results.
GdiFlush();
LeaveCriticalSection(&CGDIThread::m_csGDILock);

```

**Acquire critical section:** This panel shows the source code for the 'Acquire critical section' function. The 'Sync Object' is 'Critical Section 52'. The 'Stack' shows the following addresses and lines:

Address	Line
0x414C	206
0x414D	207
0x414E	208
0x414F	209
0x4150	210
0x4151	211
0x4152	212
0x4153	213
0x4154	214
0x4155	215
0x4156	216
0x4157	217
0x4158	218

The source code for the 'Acquire critical section' function is shown on the right, with the following lines highlighted:

```

++MY_BUG;
// Since all threads share the same HDC it is necessary
// to block all other threads while we render in the HDC
EnterCriticalSection(&CGDIThread::m_csGDILock);
{
    CBrush* oldbrush;

    STALL_HERE( 750, 5 );

    oldbrush = m_dc.SelectObject(&m_brush);
}

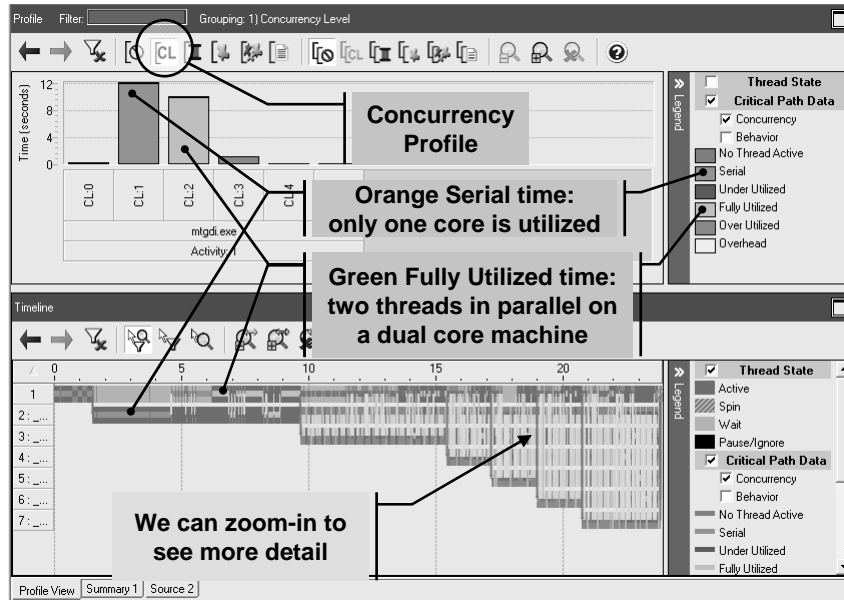
```

33



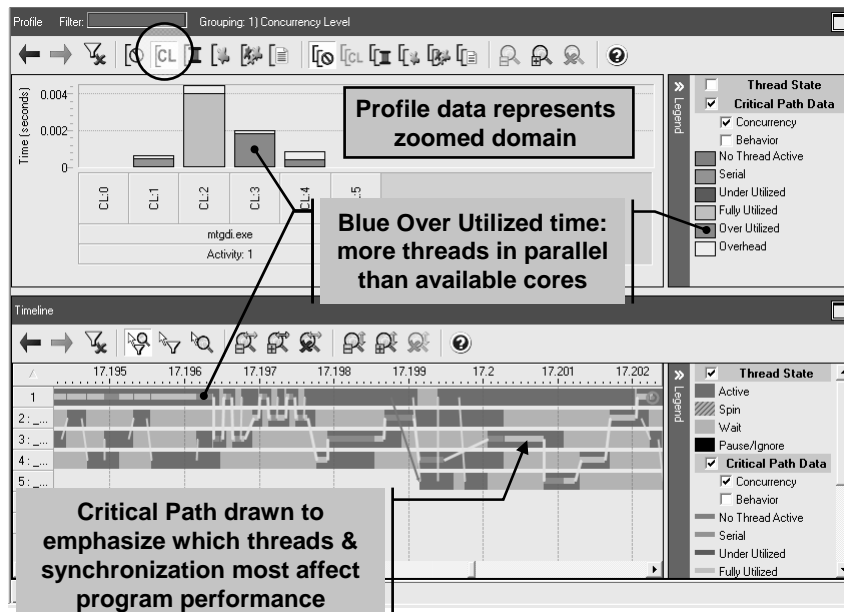
Optimize  
Threading Performance

## Concurrency Profile and Thread Timeline



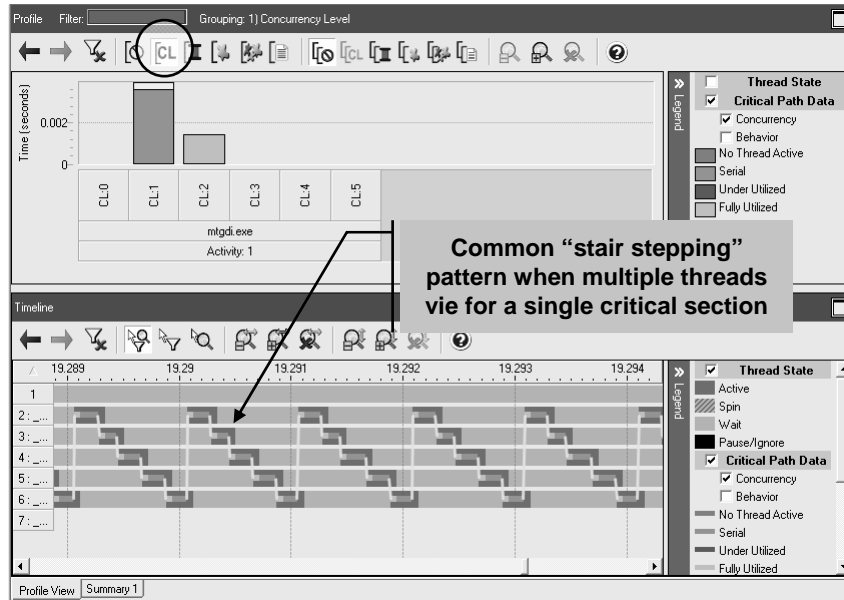
35

## Zoom-in for more detail



36

## Example lock “stair stepping” pattern



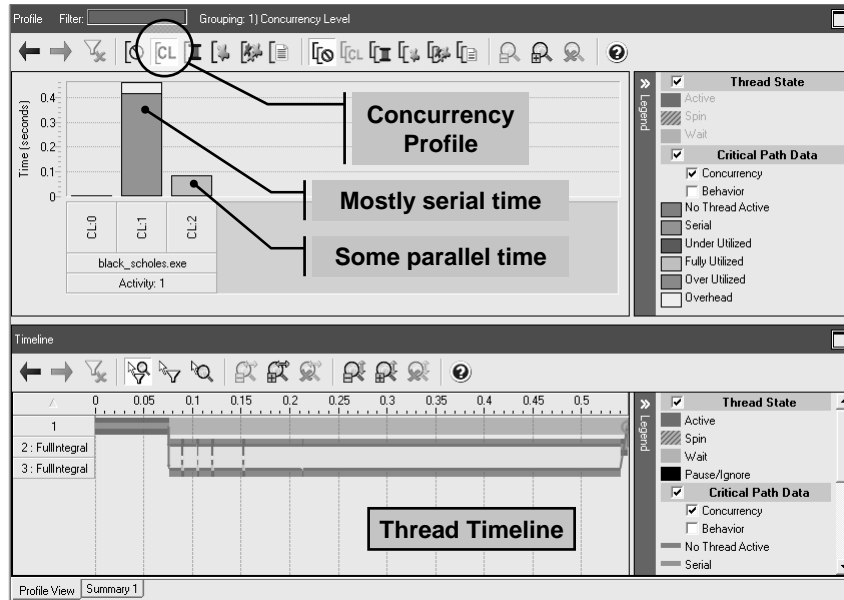
37



## Example Performance Analysis

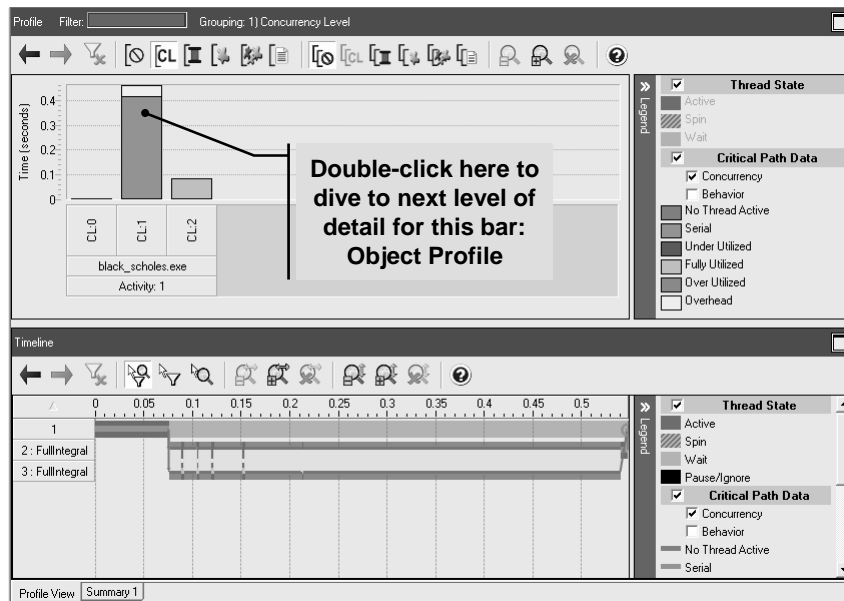
Numerical Simulation

## View initial Thread Profiler results



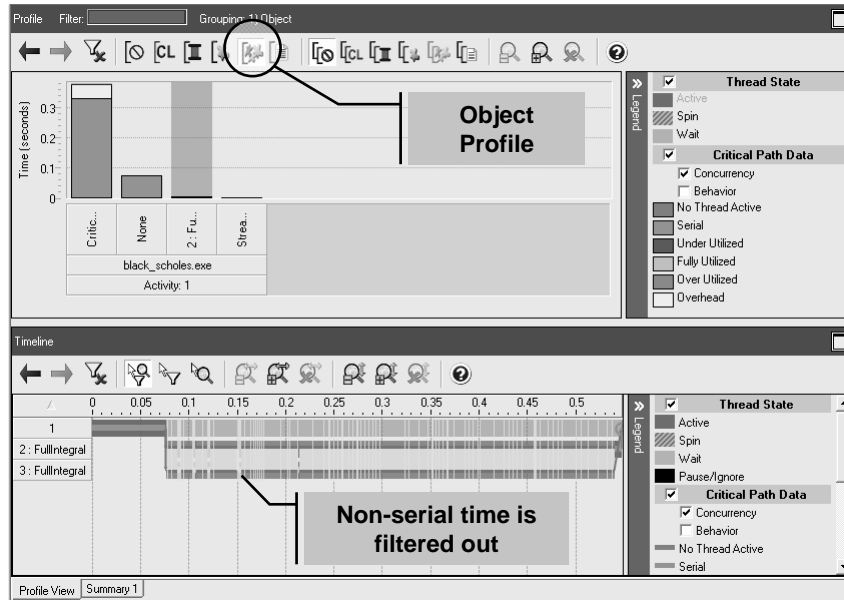
39

## Dive for more detail about serial time



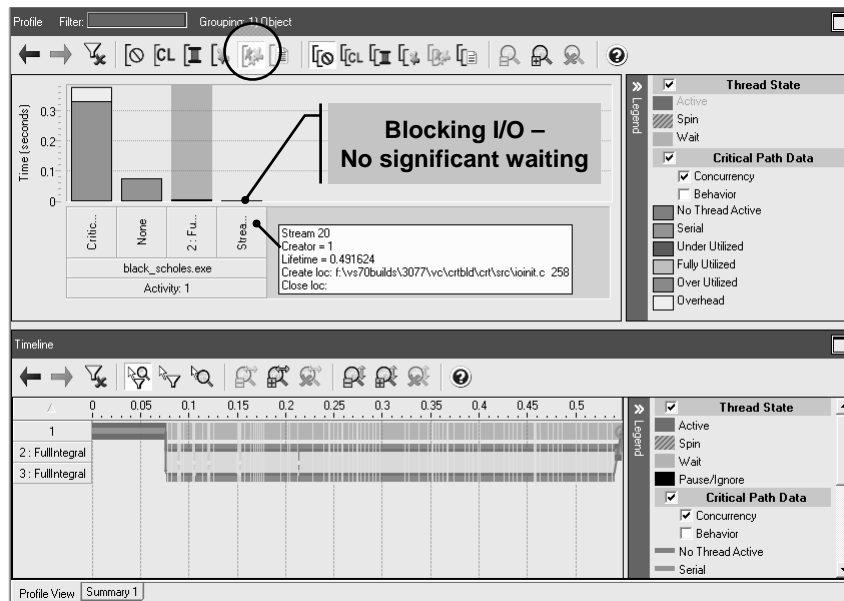
40

## Object Profile for serial time



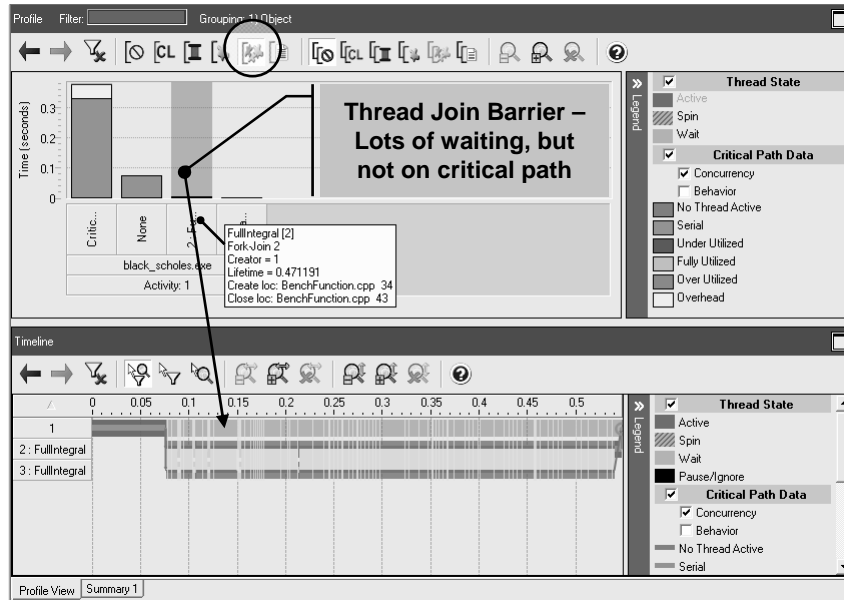
41

## Object Profile – Blocking I/O



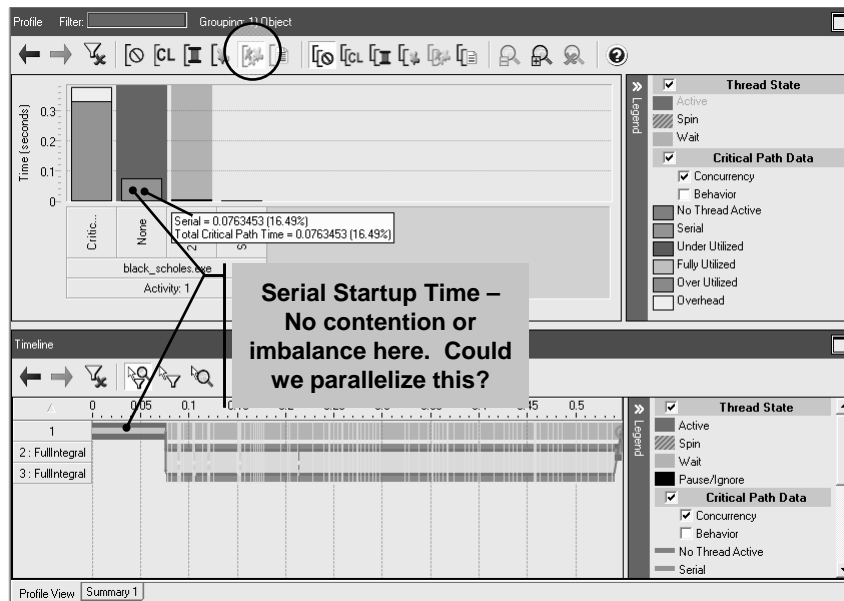
42

## Object Profile – Thread Join Barrier



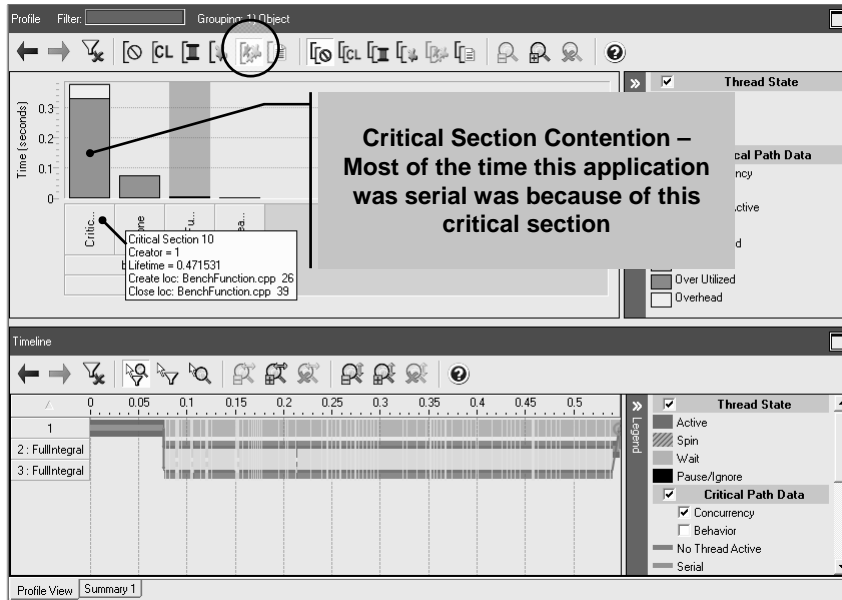
43

## Object Profile – Serial Startup



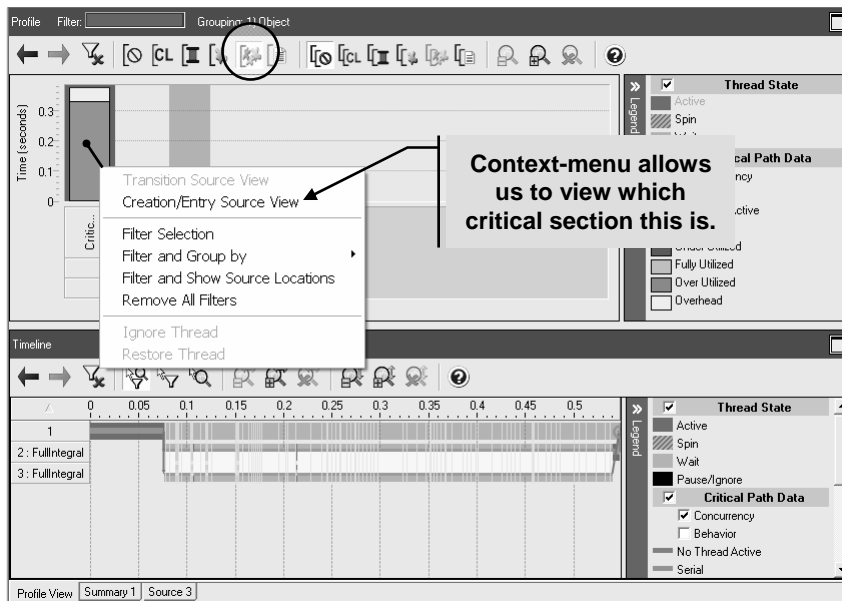
44

## Object Profile – Critical Section



45

## View source for object creation point



46

## Object create/close source view



The screenshot displays the 'Object create/close source view' in the Intel VTune Profiler. It is divided into two main sections: 'Create' and 'Close'. Both sections show a 'Call Stack' on the left, a table of 'Address' and 'Line' numbers in the middle, and the corresponding 'Source' code on the right.

**Object creation source location:** The 'Create' section shows the source code for the object's creation. The call stack includes 'mainCRTStartup' and 'main'. The source code shows the initialization of a critical section and the creation of a thread.

**Object close source location:** The 'Close' section shows the source code for the object's closure. The call stack includes 'mainCRTStartup' and 'main'. The source code shows the deletion of the critical section and the return of the thread.

47

## Dive for more detail about this object



The screenshot displays the 'Dive for more detail about this object' view in the Intel VTune Profiler. It shows a 'Source Profile' view at the top and a 'Timeline' view at the bottom. The 'Source Profile' view shows a bar chart of time spent in different source locations. The 'Timeline' view shows a detailed timeline of the object's execution, including the creation and closure of the object.

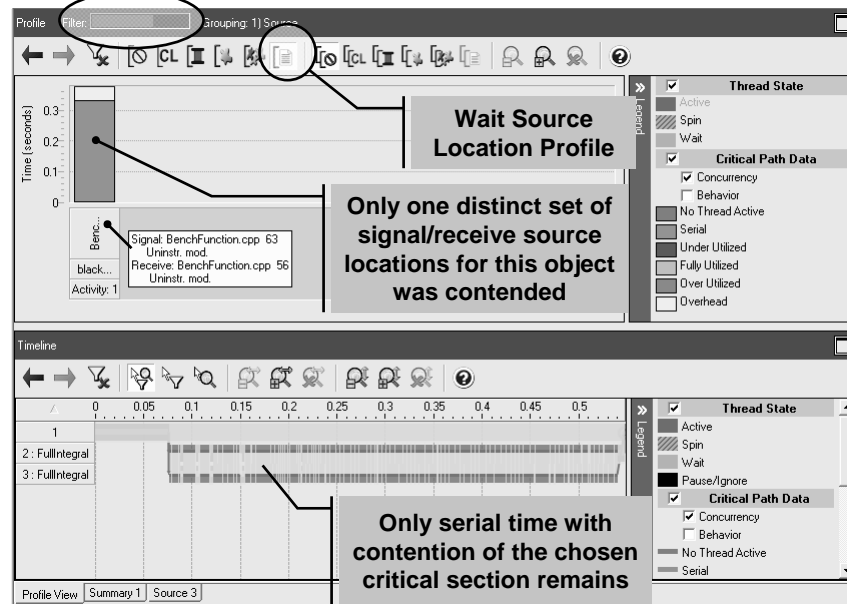
**Double-click here to dive to next level of detail for this bar: Source Profile**

The 'Source Profile' view shows a bar chart of time spent in different source locations. The 'Timeline' view shows a detailed timeline of the object's execution, including the creation and closure of the object.

48

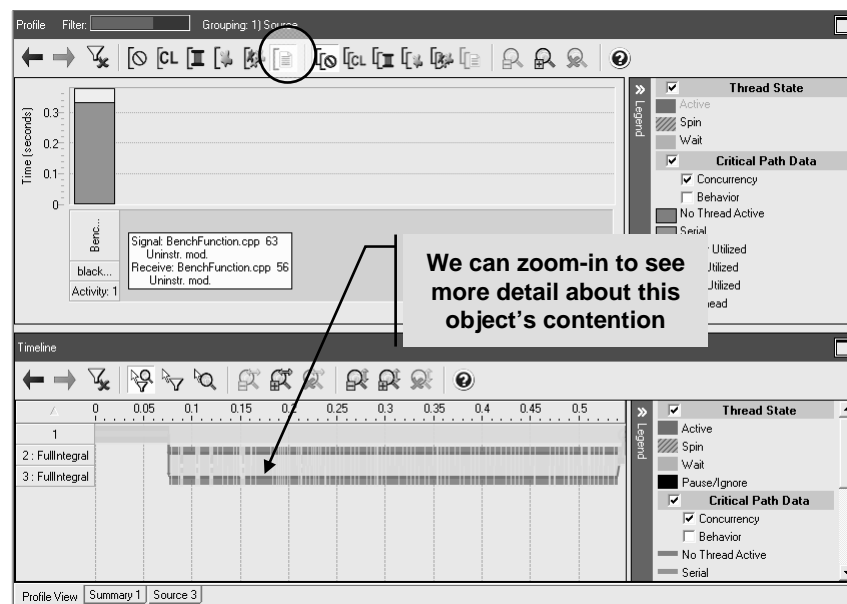


## Wait source location profile



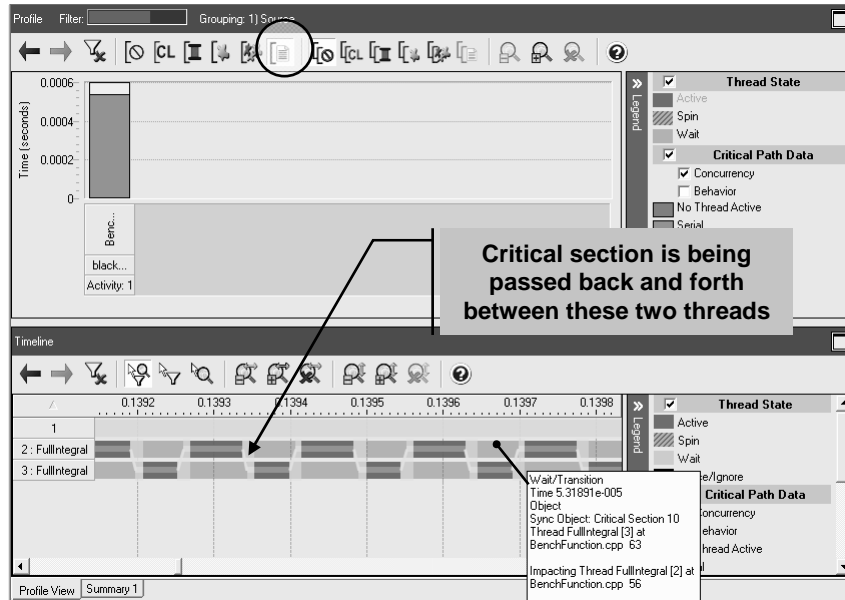
49

## Zoom-in for more detail at any time



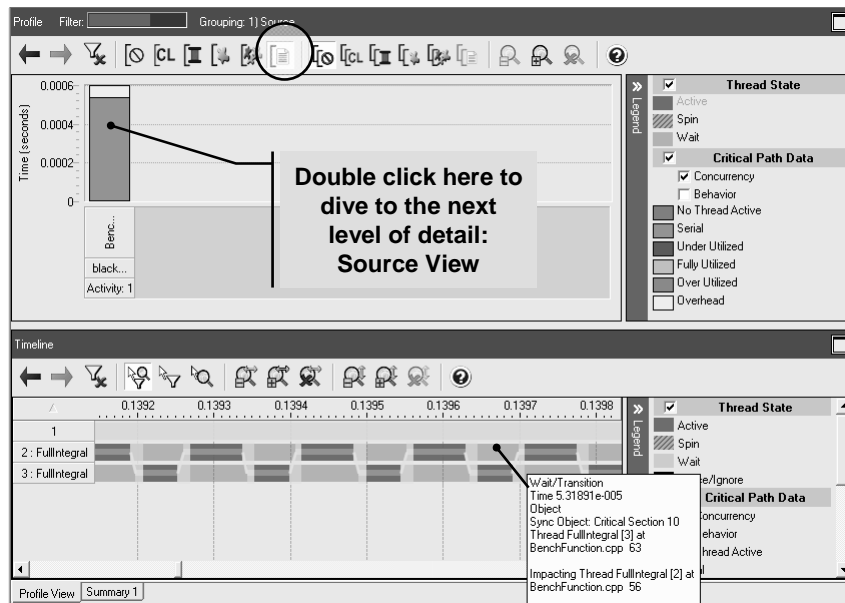
50

## Critical Section is causing serialization



51

## Dive for source view of this location



52

## Synchronization source view



**Release critical section**

Address	Line	Source
0x116D	59	{
	60	int k;
	61	double Local_Result = 0 ;
	62	EnterCriticalSection(&guard);
	63	for (k = 0; (x <= 1.0) && (k < 128);
	64	{
		Local_Result += BlackScholes(100, 110, x,
		Result += Local_Result;
		LeaveCriticalSection(&guard);
		} // while

**Acquire critical section**

Address	Line	Source
0x1165	51	while (x <= 1.0)
	52	{
	53	int k;
	54	double Local_Result = 0 ;
	55	EnterCriticalSection(&guard);
	56	for (k = 0; (x <= 1.0) && (k < 128);
	57	{
		Local_Result += BlackScholes(100, 110, x,
		Result += Local_Result;
		LeaveCriticalSection(&guard);
		} // while

53

## Analyze this critical section



**Critical section protects entire work loop iteration**

**But we think this only operates on local data**

```

{
    int k;
    double Local_Result = 0 ;

    EnterCriticalSection(&guard);
    for (k = 0; (x <= 1.0) && (k < 128); )
    {
        Local_Result += BlackScholes(100, 110, x,
    }
    Result += Local_Result;
    LeaveCriticalSection(&guard);
} // while
    
```

54

## Optimize this critical section



Let's reduce the size of the critical section so it only protects the reduction variable

```

{
    int k;
    double Local_Result = 0 ;

    EnterCriticalSection(&guard);
    for (k = 0; (x <= 1.0) && (k < 128); )
    {
        Local_Result += BlackScholes(100, 110, x,
    )
    }
    Result += Local_Result;
    LeaveCriticalSection(&guard);
} // while

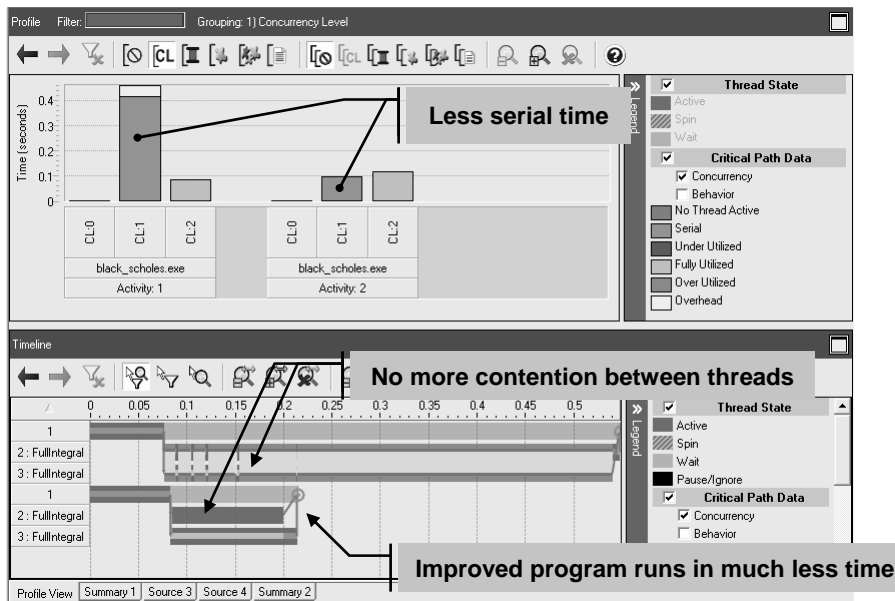
{
    int k;
    double Local_Result = 0 ;

    for (k = 0; (x <= 1.0) && (k < 128); )
    {
        Local_Result += BlackScholes(100, 110, x,
    )
    }
    EnterCriticalSection(&guard);
    Result += Local_Result;
    LeaveCriticalSection(&guard);
} // while
    
```

Verify the correctness of this assumption with Thread Checker

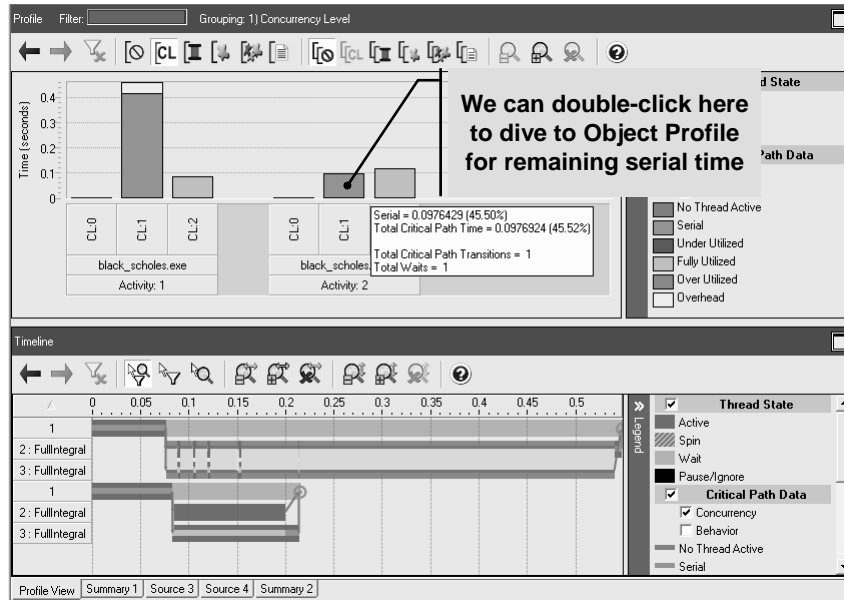
55

## Compare results of new program run



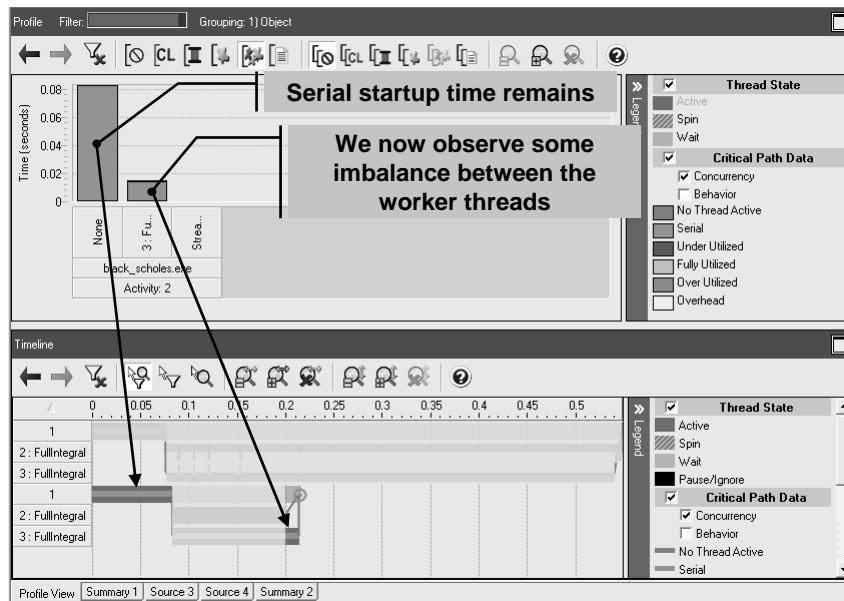
56

## Dive to see what serial time is left



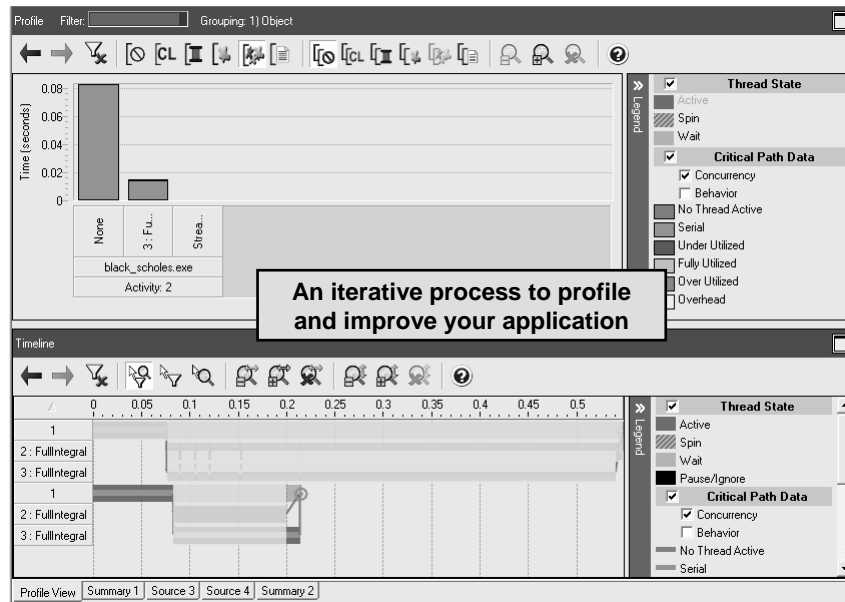
57

## Object profile for serial time



58

## Iterative process

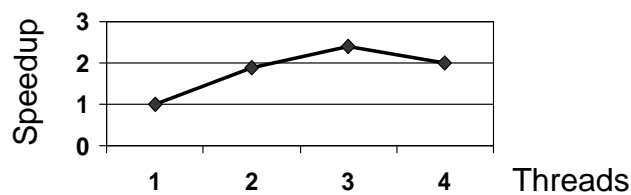


59

## Different causes of poor scalability



A tool could help determine the actual cause



1. Insufficient parallel work
2. Synchronization overhead
3. Contention
4. Load imbalance
5. Task granularity
6. Memory bandwidth / false sharing
7. ...

Thread Profiler can help you detect many issues that limit scalability

60



61

