Copyright © 2006 Intel Corporation. All Rights Reserved.

Threading Methodology based on Intel® Tools

Denys Kotlyarov
Vasiliy Malanin

Agenda

A Generic Development Cycle

Case Study: Prime Number Generation

Common Performance Issues

What is Parallelism?

Two or more processes or threads execute at the same time

Parallelism for threading architectures

- Multiple processes
  - Communication through Inter-Process Communication (IPC)
- Single process, multiple threads
  - Communication through shared memory

What is Parallelism?

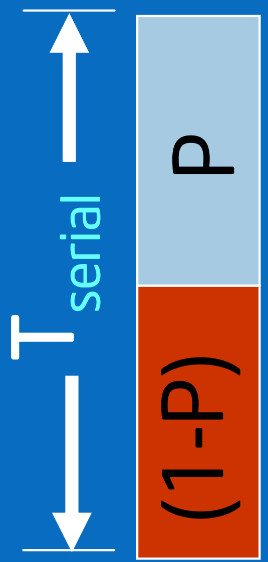
Two or more processes or threads execute at the same time

Parallelism for threading architectures

- Multiple processes
  - Communication through Inter-Process Communication (IPC)
- Single process, multiple threads
  - Communication through shared memory

Amdahl's Law

Describes the upper bound of parallel execution speedup



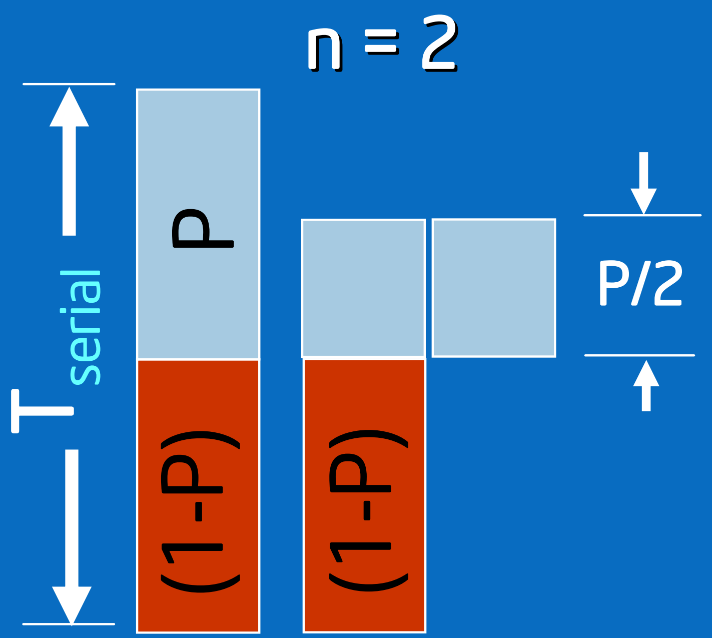
$$T_{parallel} = \{(1-P) + P/n\}\, T_{serial}$$

n = number of processors

$$Speedup = T_{serial} / T_{parallel}$$

Amdahl's Law

Describes the upper bound of parallel execution speedup



0.5 + 0.25

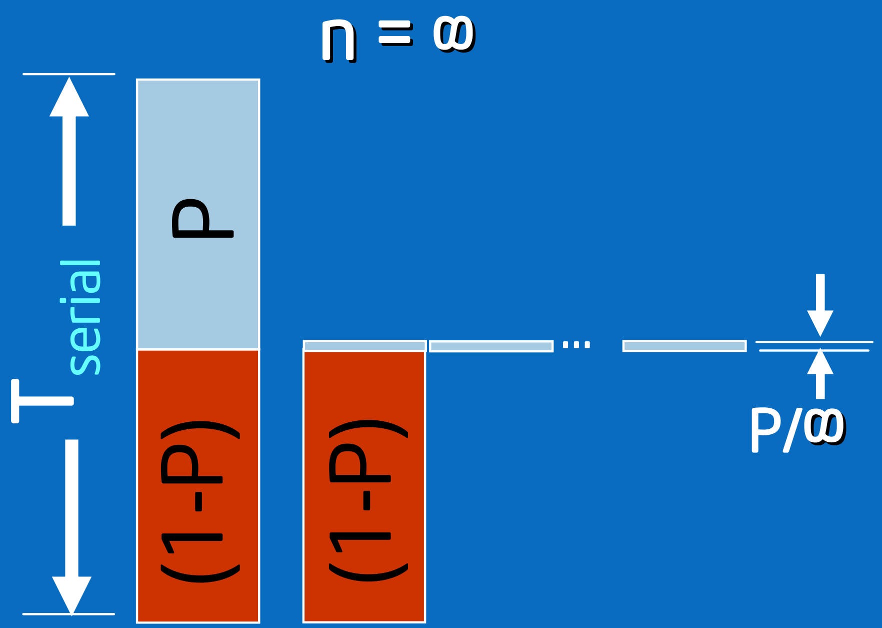
$$T_{parallel} = \{(1-P) + P/n\}\, T_{serial}$$

n = number of processors

$$Speedup = T_{serial} / T_{parallel} \quad 1.0/0.75 = 1.33$$

Amdahl's Law

Describes the upper bound of parallel execution speedup



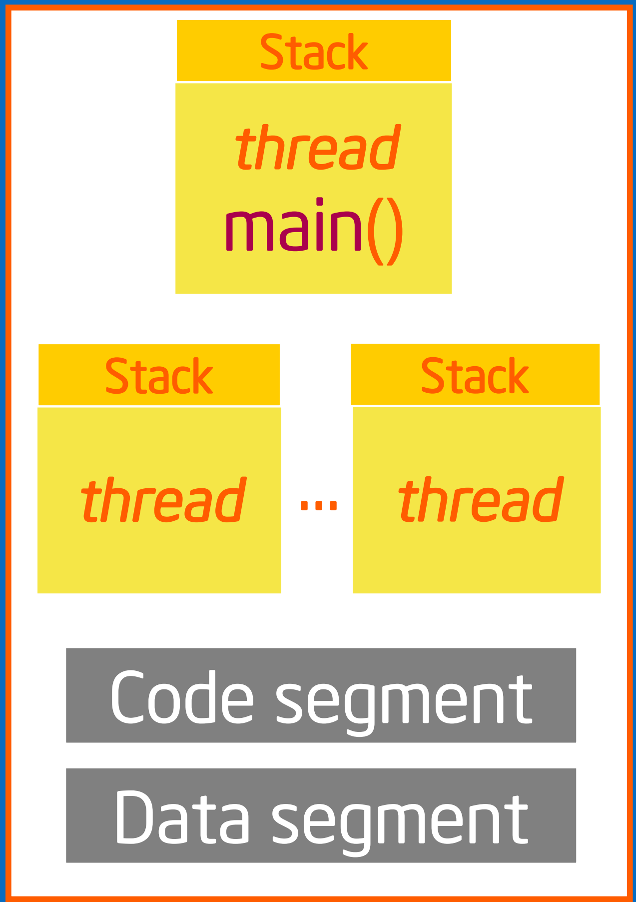
$$T_{parallel} = \{\underbrace{(1\text{-}P)}_{0.5} + \underbrace{P/n}_{0.0}\}\, T_{serial}$$

n = number of processors

$$\text{Speedup} = T_{serial} \,/\, T_{parallel} \qquad 1.0/0.5 = 2.0$$

Serial code limits speedup

Processes and Threads



Modern operating systems load programs as processes

- Resource holder
- Execution

A process starts executing at its entry point as a thread

Threads can create other threads within the process

- Each thread gets its own stack

All threads within a process share code & data segments

Threads – Benefits & Risks

Benefits

- Increased performance and better resource utilization
  - Even on single processor systems - for hiding latency and increasing throughput
- IPC through shared memory is more efficient

Risks

- Increases complexity of the application
- Difficult to debug (data races, deadlocks, etc.)

Commonly Encountered Questions with Threading Applications

Where to thread?

How long would it take to thread?

How much re-design/effort is required?

Is it worth threading a selected region?

What should the expected speedup be?

Will the performance meet expectations?

Will it scale as more threads/data are added?

Which threading model to use?

Prime Number Generation

i	factor
3	2
5	2
7	2 3
9	2 3
11	2 3
13	2 3 4
15	2 3
17	2 3 4
19	2 3 4

```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
          factor ++;

    return (factor > limit);
}

void FindPrimes(int start, int end)
{
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
```

Prime Number Generation

i	factor
3	2
5	2
7	2 3
9	2 3
11	2 3
13	2 3 4
15	2 3
17	2 3 4
19	2 3 4

```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
        factor ++;
```

```
C:\WINDOWS\system32\cmd.exe

C:\classfiles\PrimeSingle\Release>PrimeSingle.exe 1 20
100%

      8 primes found between       1 and      20 in     0.00 secs

C:\classfiles\PrimeSingle\Release>_
```

```
{
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
```

Intel Software

Demo 1

Run Serial version of Prime code

- Compile with Intel compiler in Visual Studio
- Run a few times with different ranges

Development Methodology

Analysis

- Find computationally intense code

Design (Introduce Threads)

- Determine how to implement threading solution

Debug for correctness

- Detect any problems resulting from using threads

Tune for performance

- Achieve best parallel performance

Development Cycle

Analysis

–VTune™ Performance Analyzer

Design (Introduce Threads)

–Intel® Performance libraries: IPP and MKL

–OpenMP* (Intel® Compiler)

–Explicit threading (Win32*, Pthreads*)

Debug for correctness

–Intel® Thread Checker

–Intel Debugger

Tune for performance

–Intel® Thread Profiler

–VTune™ Performance Analyzer

Analysis - Sampling

Use VTune Sampling to find hotspots in application

Let's use the project PrimeSingle for analysis

- PrimeSingle <start> <end>

Usage: **`PrimeSingle 1 1000000`**

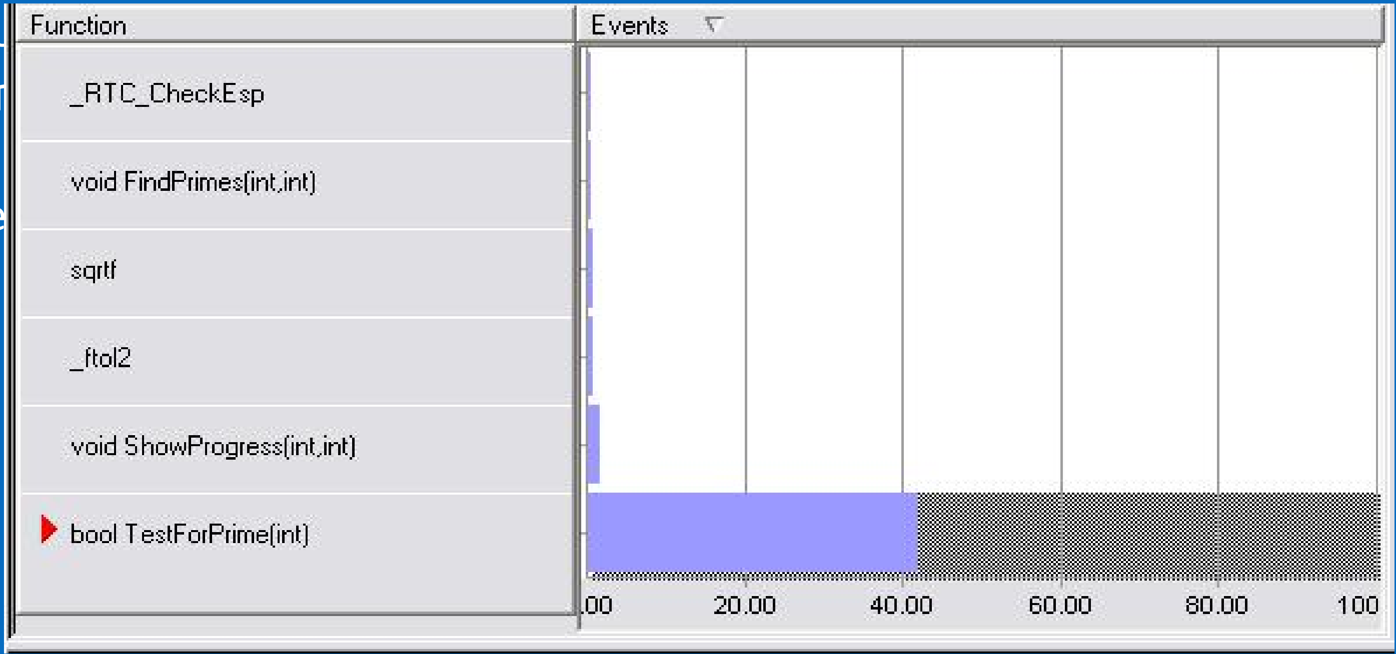
Analysis - Sampling

Use VTune Sampling to find hotspots in application

Let's u

- Pr

Usage



Analysis - Sampling

Use VTune Sampling to find ho

Let's u

- Pr

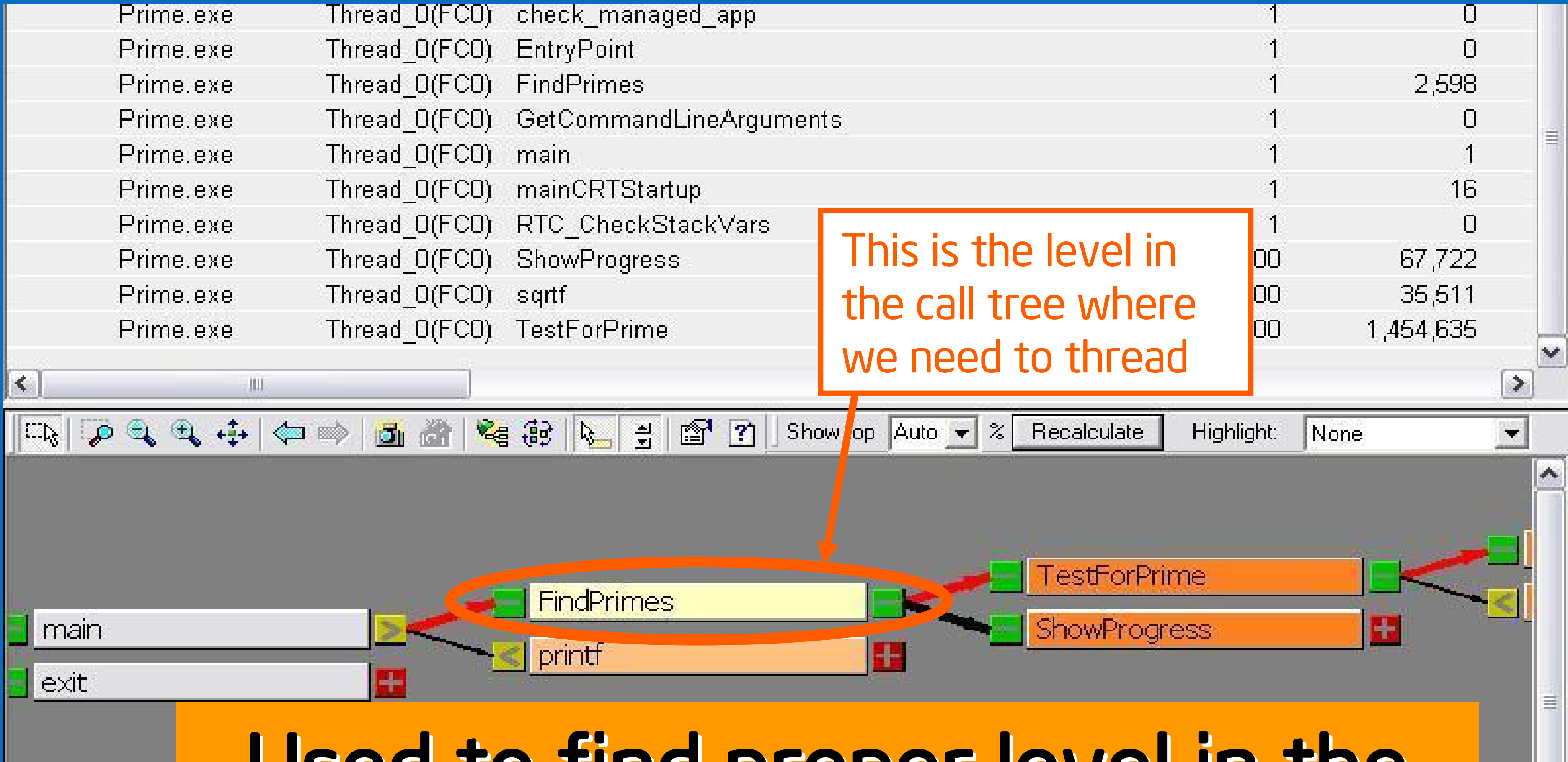
Usage


```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor))
          factor ++;

    return (factor > limit);
}
```

```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;
    for( int i = start; i <= end; i+= 2 ){
       if( TestForPrime(i) )
    }
}
```

Identifies the time consuming regions

Analysis - Call Graph



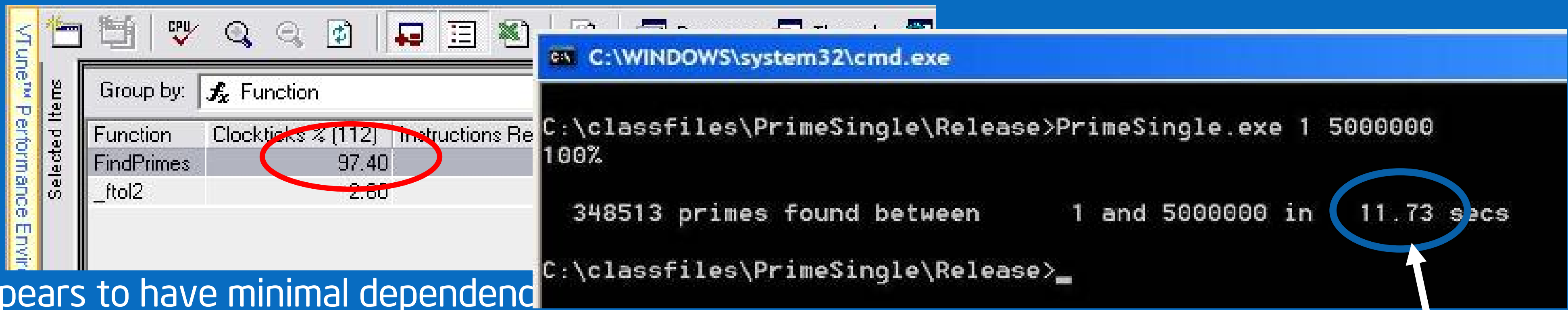
Used to find proper level in the call-tree to thread

Analysis

Where to thread?

- FindPrimes()

Is it worth threading a selected region?



- Appears to have minimal dependenc
- Appears to be data-parallel
- Consumes over 95% of the run time

Baseline measurement

Demo 2

Run code with ‘1 5000000’ range to get baseline measurement

- Make note for future reference

Run VTune analysis on serial code

- What function takes the most time?

Foster's Design Methodology

From *Designing and Building Parallel Programs* by Ian Foster

Four Steps:

- Partitioning
  - Dividing computation and data
- Communication
  - Sharing data between computations
- Agglomeration
  - Grouping tasks to improve performance
- Mapping
  - Assigning tasks to processors/threads

Designing Threaded Programs

Partition

- Divide problem into tasks

Communicate

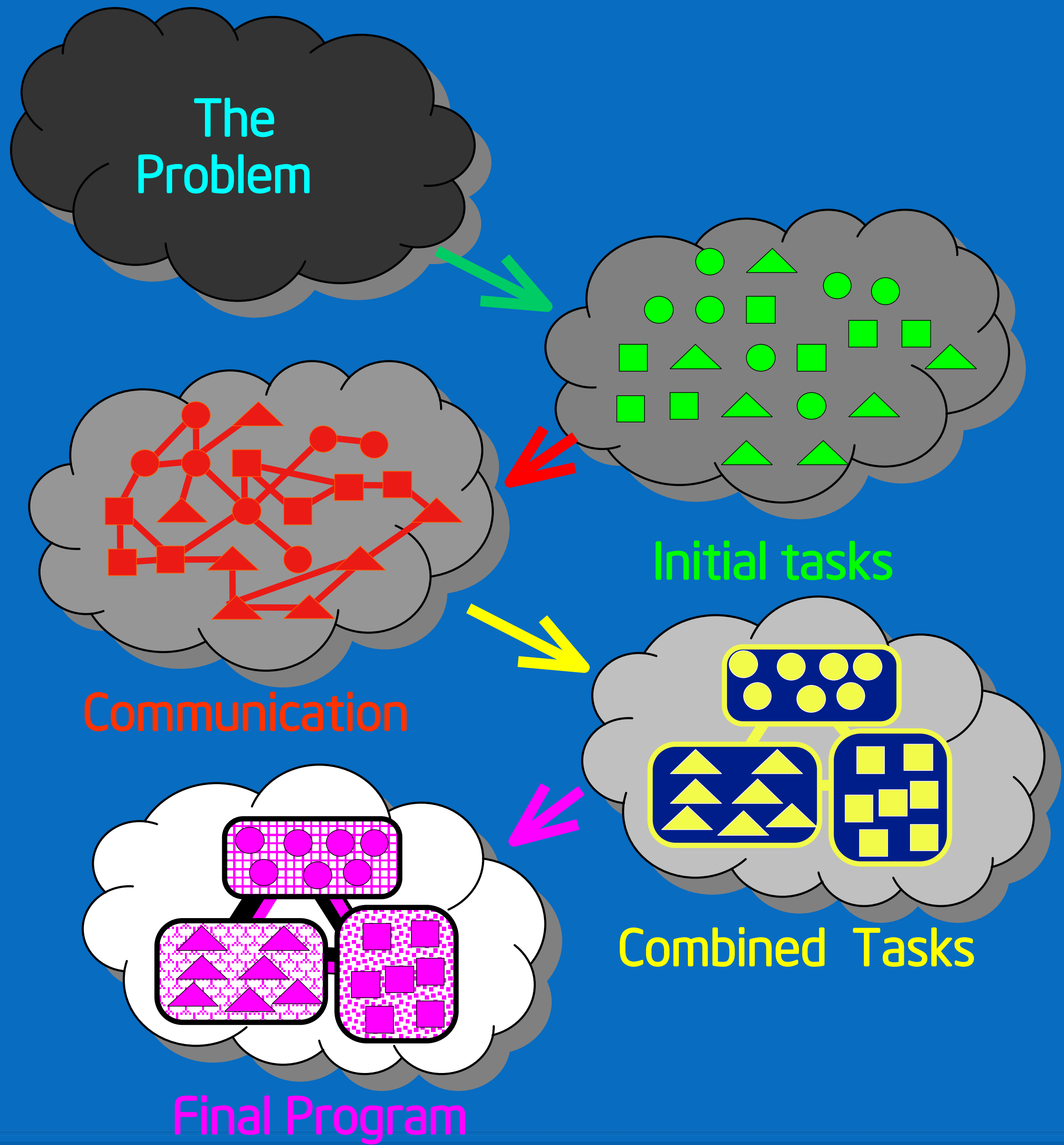
- Determine amount and pattern of communication

Agglomerate

- Combine tasks

Map

- Assign agglomerated tasks to created threads



Parallel Programming Models

Functional Decomposition

- Task parallelism
- Divide the computation, then associate the data
- Independent tasks of the same problem

Data Decomposition

- Same operation performed on different data
- Divide data into pieces, then associate computation

Design

What is the expected benefit?

Speedup(2P) = 100/(96/2+4) = ~1.92X

How do you achieve this with the least effort?

Rapid prototyping with OpenMP

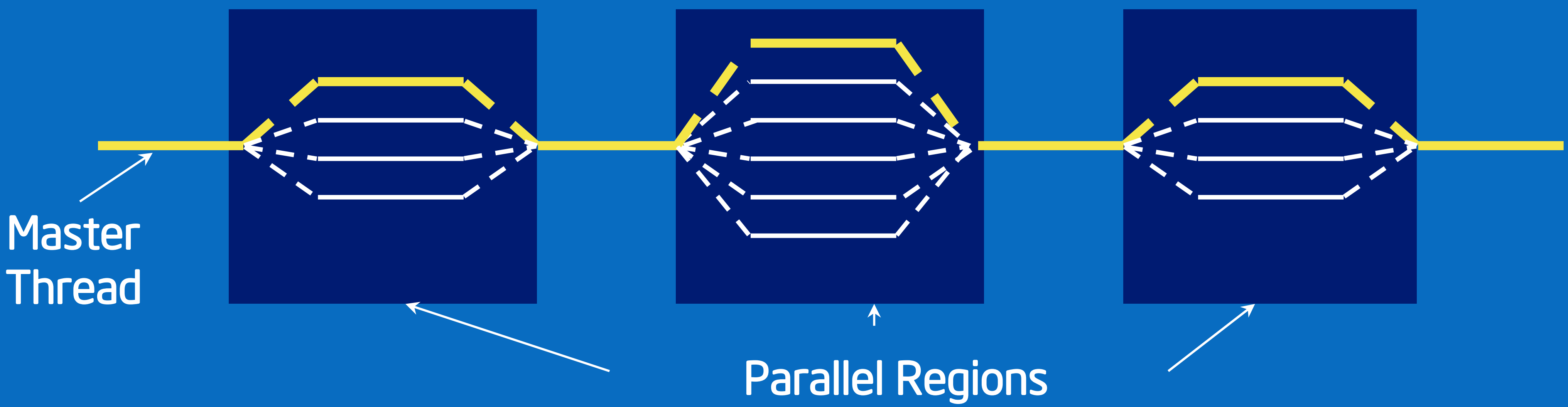
How long would it take to thread?

How much re-design/effort is required?

OpenMP

Fork-join parallelism:

- Master thread spawns a team of threads as needed
- Parallelism is added incrementally
  - Sequential program evolves into a parallel program



Design

```
#pragma omp parallel for
  for( int i = start; i <= end; i+= 2 ){
              rPrime(i) )
              rimes[gPrimesFound++] = i;
    ShowProgress(i, range);
  }
```

OpenMP

Design

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

Create threads here for this parallel region

Design

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

Divide iterations of the **for** loop

Design

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

```
C:\WINDOWS\system32\cmd.exe

C:\classfiles\PrimeOpenMP\Debug>PrimeOpenMP.exe 1 5000000
 90%

  348018 primes found between       1 and 5000000 in     8.36 secs

C:\classfiles\PrimeOpenMP\Debug>_
```

Demo 3

Run OpenMP version of code

- Compile code
- Run with ‘1 5000000’ for comparison
  - What is the speedup?

Design

What is the expected benefit?

How do you achieve this with the least effort?

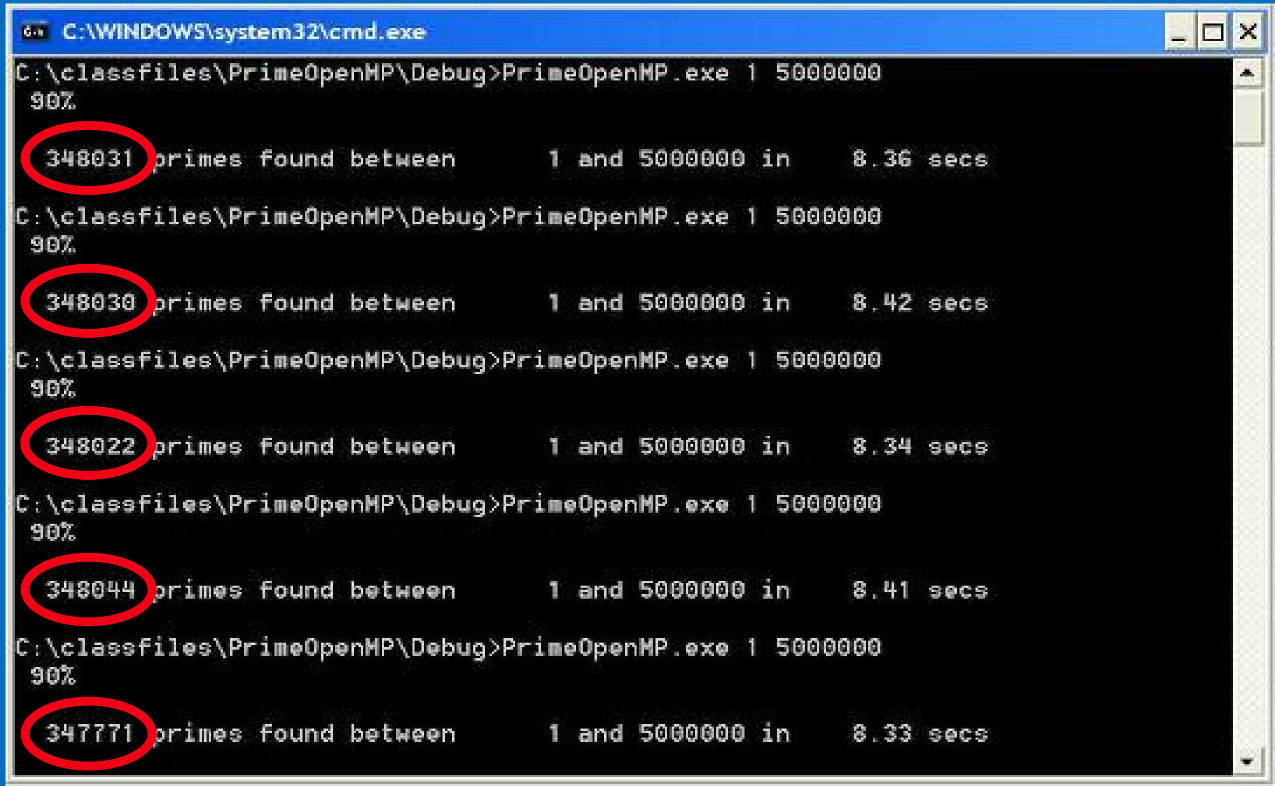
Speedup of 1.40X (less than 1.92X)

How long would it take to thread?

How much re-design/effort is required?

Is this the best speedup possible?

Debugging for Correctness

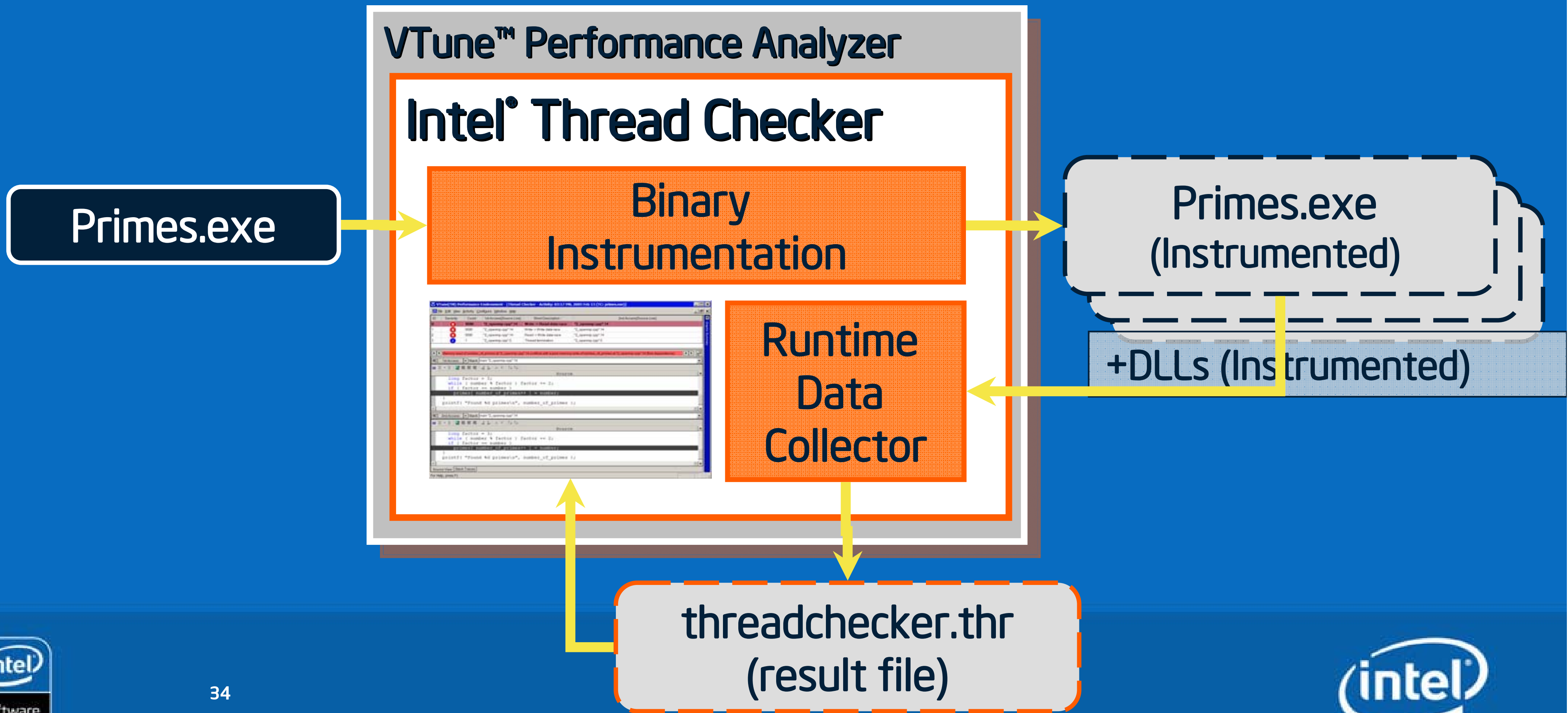


Is this threaded implementation right?

No! The answers are different each time …

Debugging for Correctness

Intel® Thread Checker pinpoints notorious threading bugs like data races, stalls and deadlocks



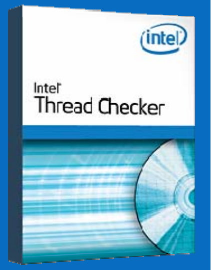
PrimeOpenMP.cpp | **Thread Checker - Activity:**

VTune™ Performance Environment

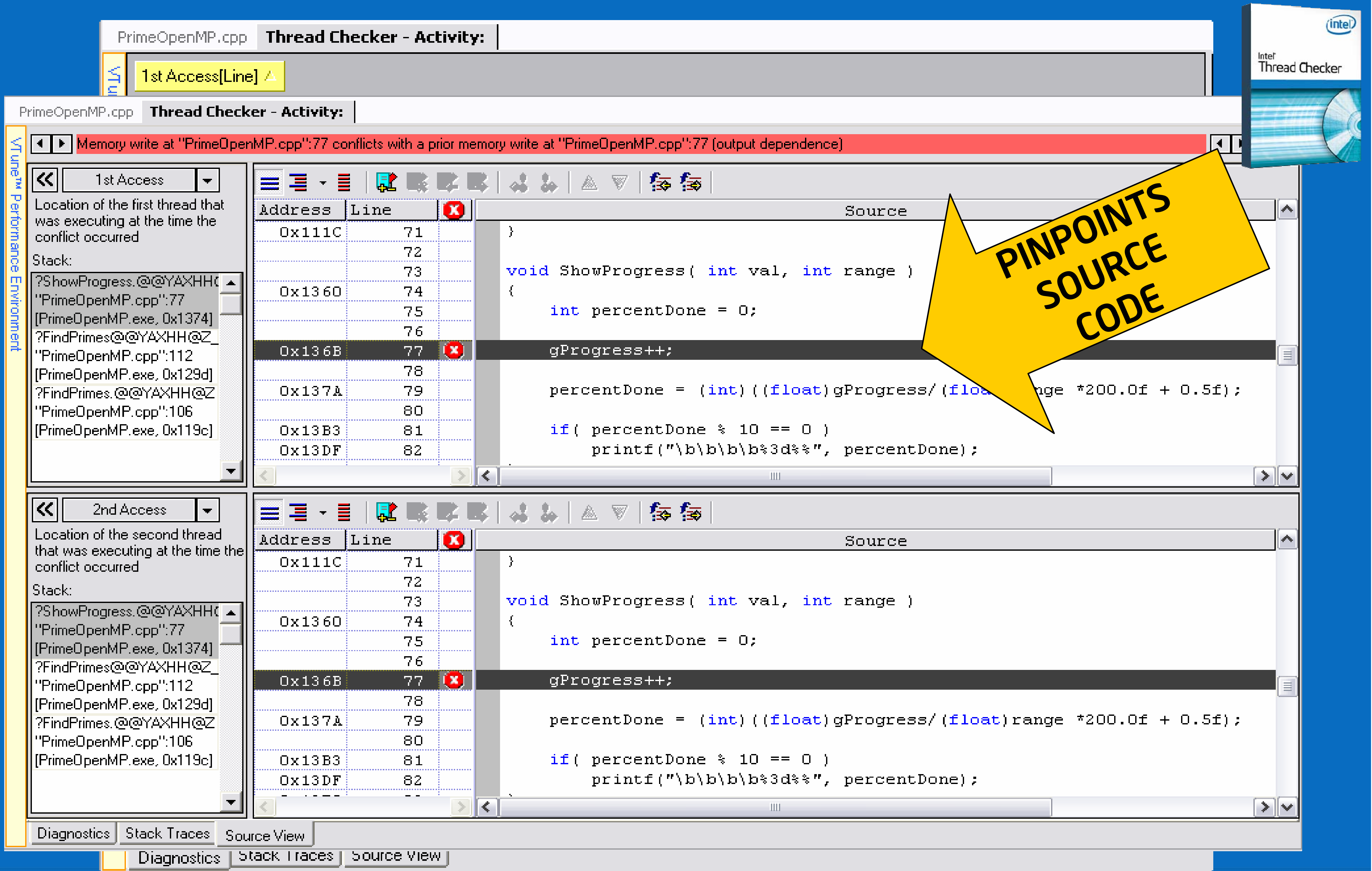
1st Access[Line]

Context[Best]	ID	Short Descrip...	Severity	Description	Count	Filtered
Group 1: 106 (Diagnostics: 2; Filtered: 0)						
Whole Program 1	6	Thread termination	i	Thread Info at "PrimeOpenMP.cpp":106 - includes stack allocation of 3145728 and use of 4096 bytes	1	False
Whole Program 2	7	Thread termination	i	Thread Info at "PrimeOpenMP.cpp":106 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Group 2: 110 (Diagnostics: 2; Filtered: 0)						
"PrimeOpenM	5	Write -> Write data-race	X	Memory write at "PrimeOpenMP.cpp":110 conflicts with a prior memory write at "PrimeOpenMP.cpp":110 (output dependence)	1	False
"PrimeOpenM	4	Write -> Read data-race	X	Memory read at "PrimeOpenMP.cpp":110 conflicts with a prior memory write at "PrimeOpenMP.cpp":110 (flow dependence)	1	False
Group 3: 117 (Diagnostics: 1; Filtered: 0)						
Whole Program 3	8	Thread termination	i	Thread Info at "PrimeOpenMP.cpp":117 - includes stack allocation of 1048576 and use of 4096 bytes	1	False
Group 4: 77 (Diagnostics: 3; Filtered: 0)						
"PrimeOpenM	3	Write -> Write data-race	X	Memory write at "PrimeOpenMP.cpp":77 conflicts with a prior memory write at "PrimeOpenMP.cpp":77 (output dependence)	1	False
"PrimeOpenM	2	Write -> Read data-race	X	Memory read at "PrimeOpenMP.cpp":77 conflicts with a prior memory write at "PrimeOpenMP.cpp":77 (flow dependence)	1	False
"PrimeOpen	**1**	**Read -> Write data-race**	X	**Memory write at "PrimeOpenMP.cpp":77 conflicts with a prior memory read at "PrimeOpenMP.cpp":77 (anti dependence)**	**1**	**False**

Diagnostics | Stack Traces | Source View



intel Software


PrimeOpenMP.cpp
Thread Checker - Activity:
1st Access[Line]
Memory write at "PrimeOpenMP.cpp":77 conflicts with a prior memory write at "PrimeOpenMP.cpp":77 (output dependence)
VTune™ Performance Environment
1st Access
Location of the first thread that was executing at the time the conflict occurred
Stack:
?ShowProgress.@@YAXHH(
"PrimeOpenMP.cpp":77
[PrimeOpenMP.exe, 0x1374]
?FindPrimes@@YAXHH@Z_
"PrimeOpenMP.cpp":112
[PrimeOpenMP.exe, 0x129d]
?FindPrimes.@@YAXHH@Z
"PrimeOpenMP.cpp":106
[PrimeOpenMP.exe, 0x119c]
Address
Line
Source
0x111C 71
72
73
0x1360 74
75
76
0x136B 77
78
0x137A 79
80
0x13B3 81
0x13DF 82
}
void ShowProgress(int val, int range)
{
int percentDone = 0;
gProgress++;
percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);
if(percentDone % 10 == 0)
printf("\b\b\b\b%3d%%", percentDone);
2nd Access
Location of the second thread that was executing at the time the conflict occurred
Diagnostics
Stack Traces
Source View
PINPOINTS SOURCE CODE
Intel Thread Checker

Demo 4

Use Thread Checker to analyze threaded application

- Create Thread Checker activity
- Run application
- Are any errors reported?

Debugging for Correctness

How much re-design/effort is required?

Thread Checker reported only 3 dependencies, so effort required should be low

How long would it take to thread?

Debugging for Correctness
Possible Solutions: Solution 1 – Not Optimal

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

Will create a critical section for this reference

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

Will create a critical section for both these references

Debugging for Correctness
Possible Solutions: Solution 2 – Optimal

```
void FindPrimes(int start, int end)
{
   // start is always odd
   int range = end - start + 1;

#pragma omp parallel for
   for( int i = start; i <= end; i += 2 )
   {
      if( TestForPrime(i) )
         globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;

      ShowProgress(i, range);
   }
}
```

Will perform atomic adding for this variable

Will perform atomic adding for this variable

```
void ShowProgress( int val, int range )
{
   long percentDone, localProgress;

   localProgress = InterlockedIncrement(&gProgress);
   percentDone = (int)((float)localProgress/(float)range*200.0f+0.5f);

   if( percentDone % 10 == 0 && lastPercentDone < percentDone / 10){
      printf("\b\b\b\b%3d%%", percentDone);
      lastPercentDone++;
   }
}
```

Demo 5

Modify and run OpenMP version of code

- Add **InterlockedIncrement** to code
- Compile code
- Run from within Thread Checker
  - If errors still present, make appropriate fixes to code and run again in Thread Checker
- Run with '1 5000000' for comparison
  - Compile and run outside Thread Checker
  - What is the speedup?

Correctness

Correct answer, but performance has slipped to ~1.33X

```
C:\WINDOWS\system32\cmd.exe

C:\classfiles\PrimeOpenMP\Debug>PrimeOpenMP.exe 1 5000000
100%

  348513 primes found between      1 and 5000000 in    8.80 secs

C:\classfiles\PrimeOpenMP\Debug>_
```

Is this the best we can expect from this algorithm?

No! From Amdahl's Law, we expect speedup close to 1.9X

Common Performance Issues

Parallel Overhead

- Due to thread creation, scheduling …

Synchronization

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

Load Imbalance

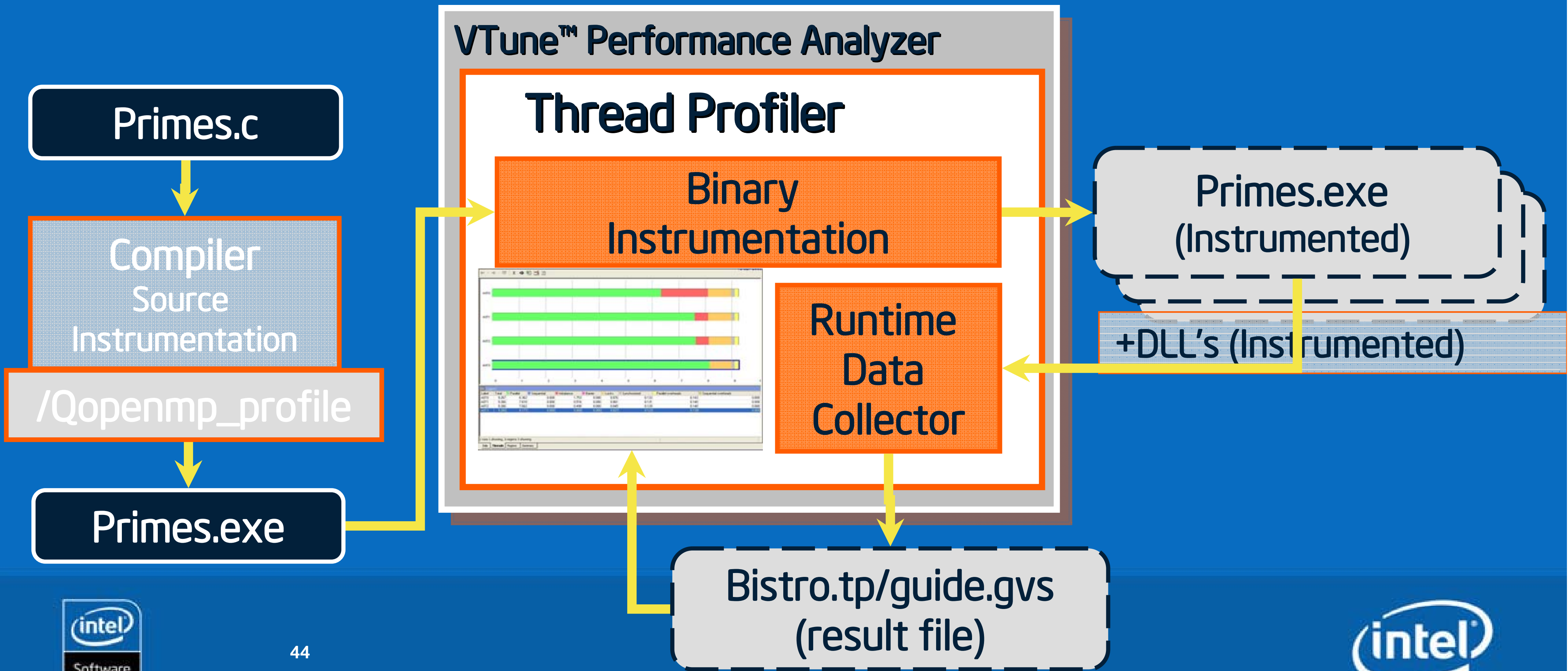
- Improper distribution of parallel work

Granularity

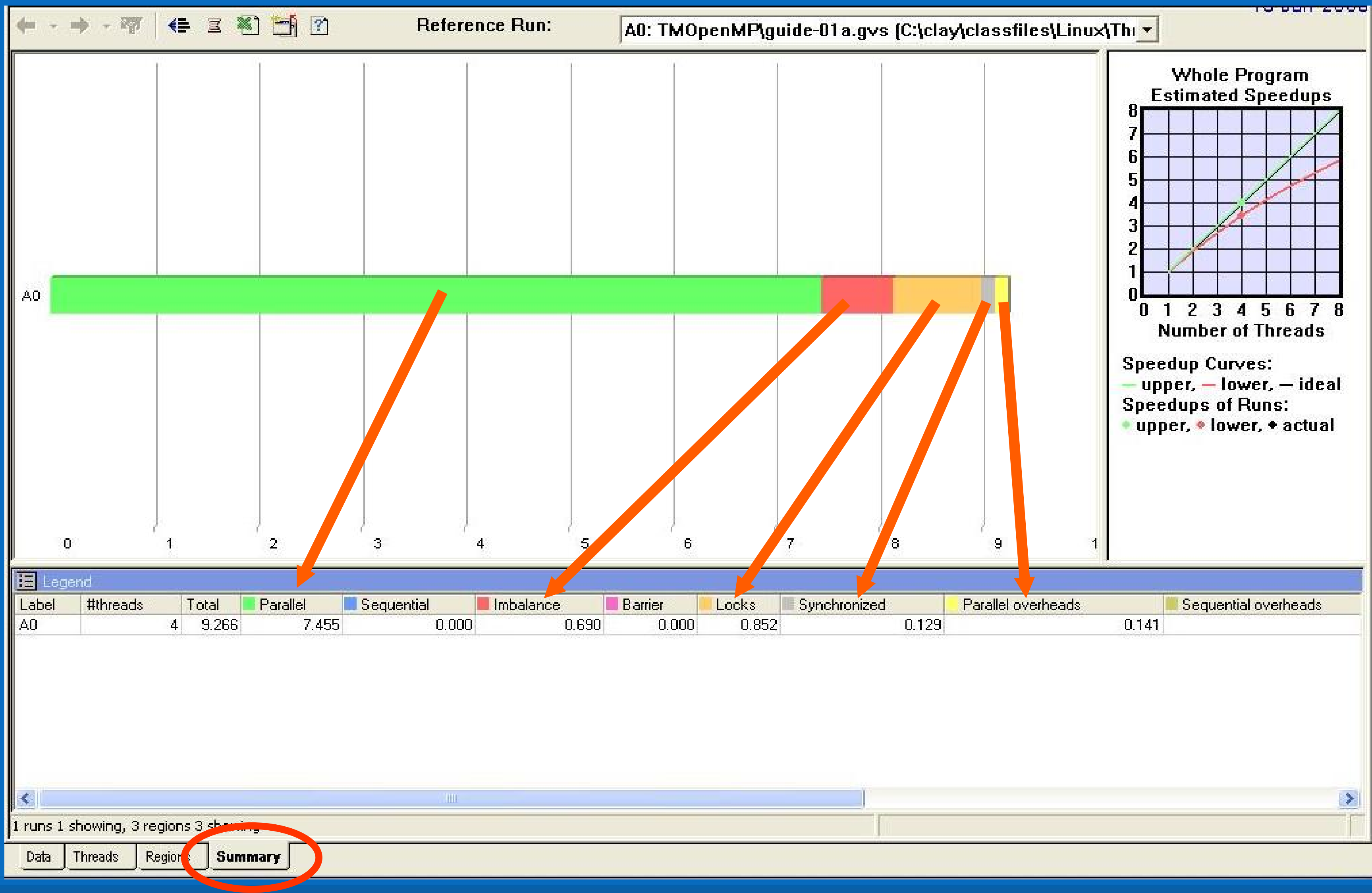
- No sufficient parallel work

Tuning for Performance

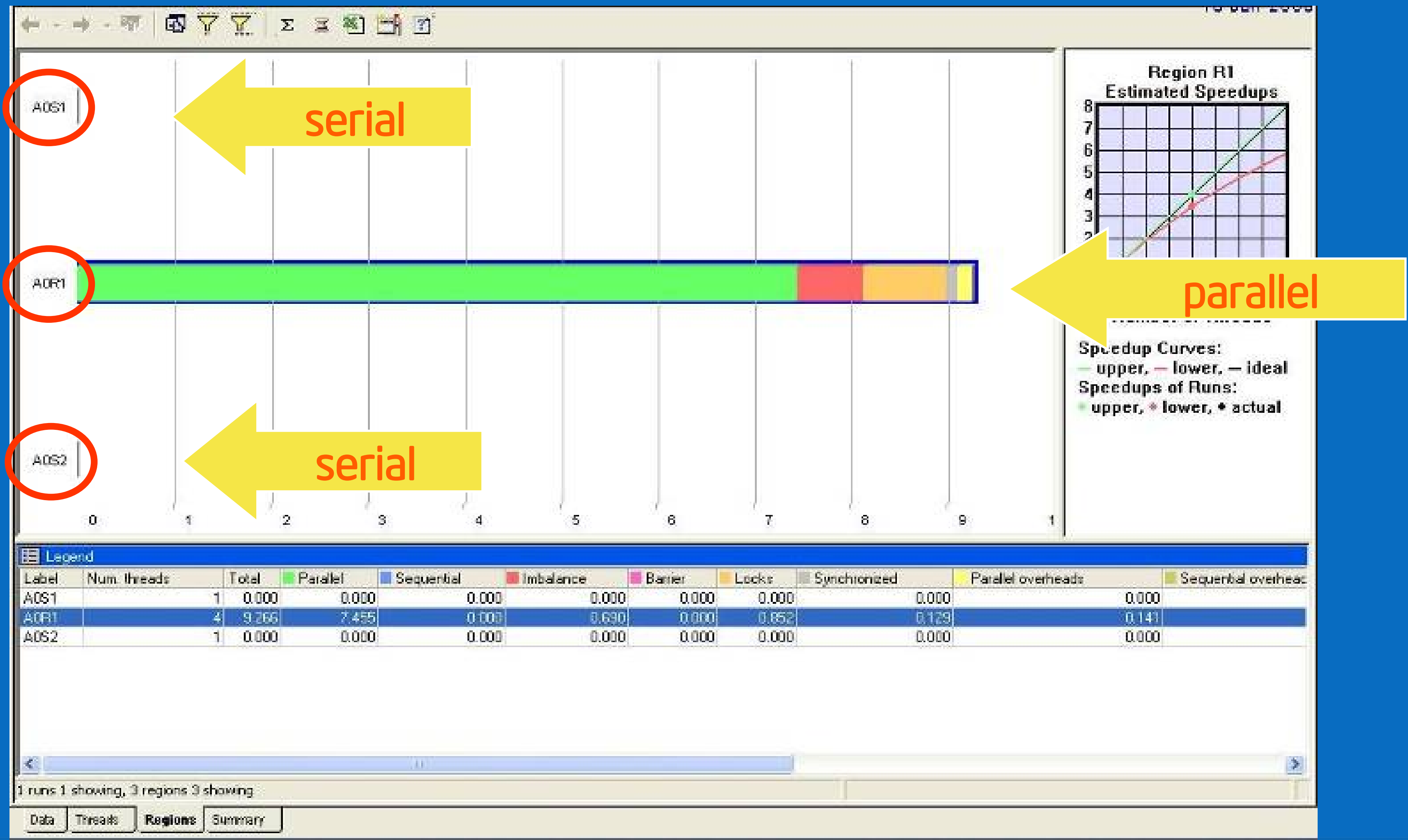
Thread Profiler pinpoints performance bottlenecks in threaded applications



Thread Profiler for OpenMP



Thread Profiler for OpenMP



Label	Num. threads	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheac
A0S1	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
A0R1	4	9.266	7.455	0.000	0.690	0.000	0.852	0.129	0.141	
A0S2	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	

Thread Profiler for OpenMP

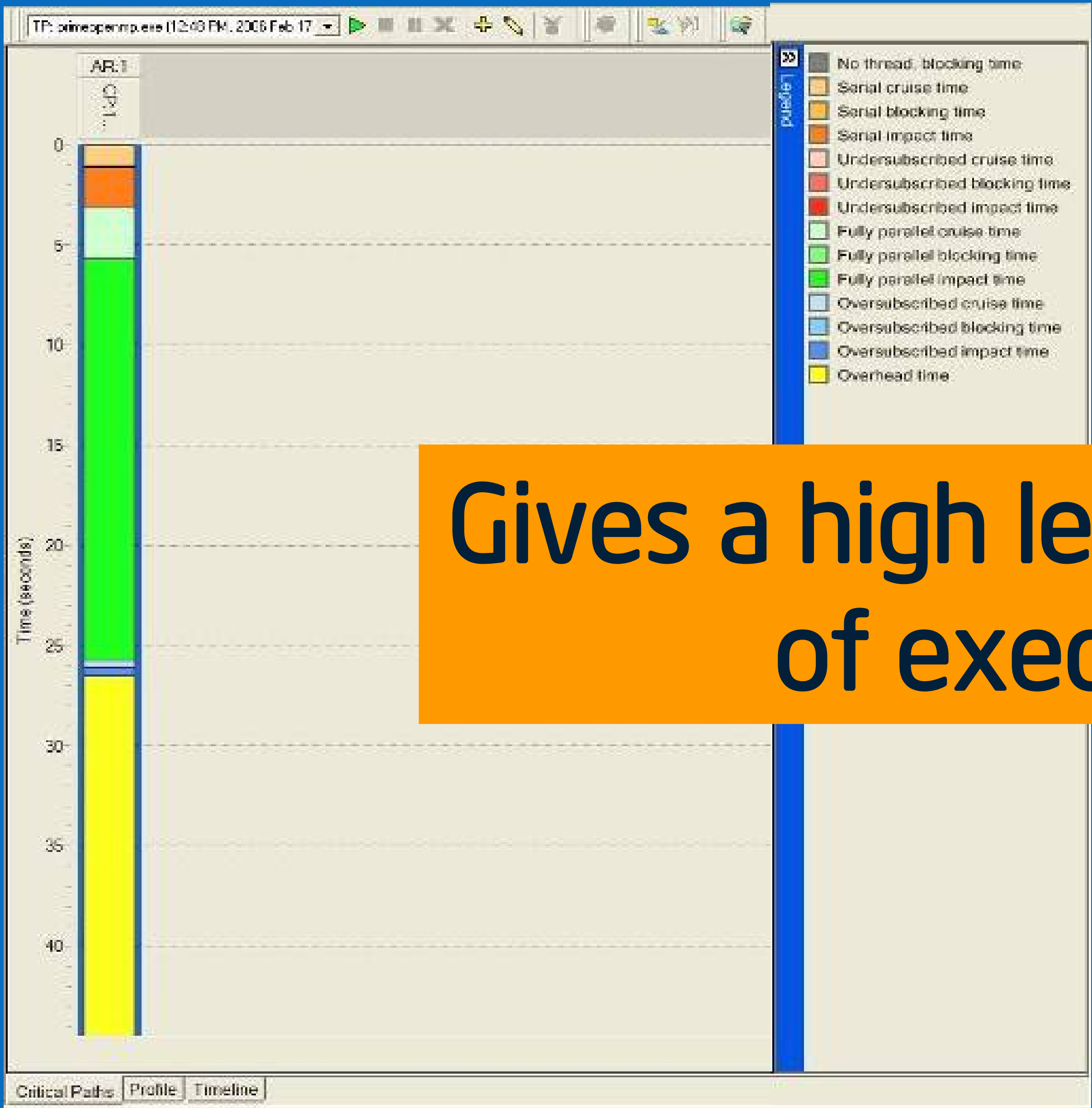
Legend

Label	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0T0	9.267	6.362	0.000	1.753	0.000	0.876	0.133	0.143	0.000
A0T1	9.266	7.618	0.000	0.516	0.000	0.861	0.131	0.140	0.000
A0T2	9.266	7.662	0.000	0.490	0.000	0.845	0.129	0.140	0.000
A0T3	9.266	8.177	0.000	0.000	0.000	0.827	0.123	0.139	0.000

1 runs 1 showing, 3 regions 3 showing

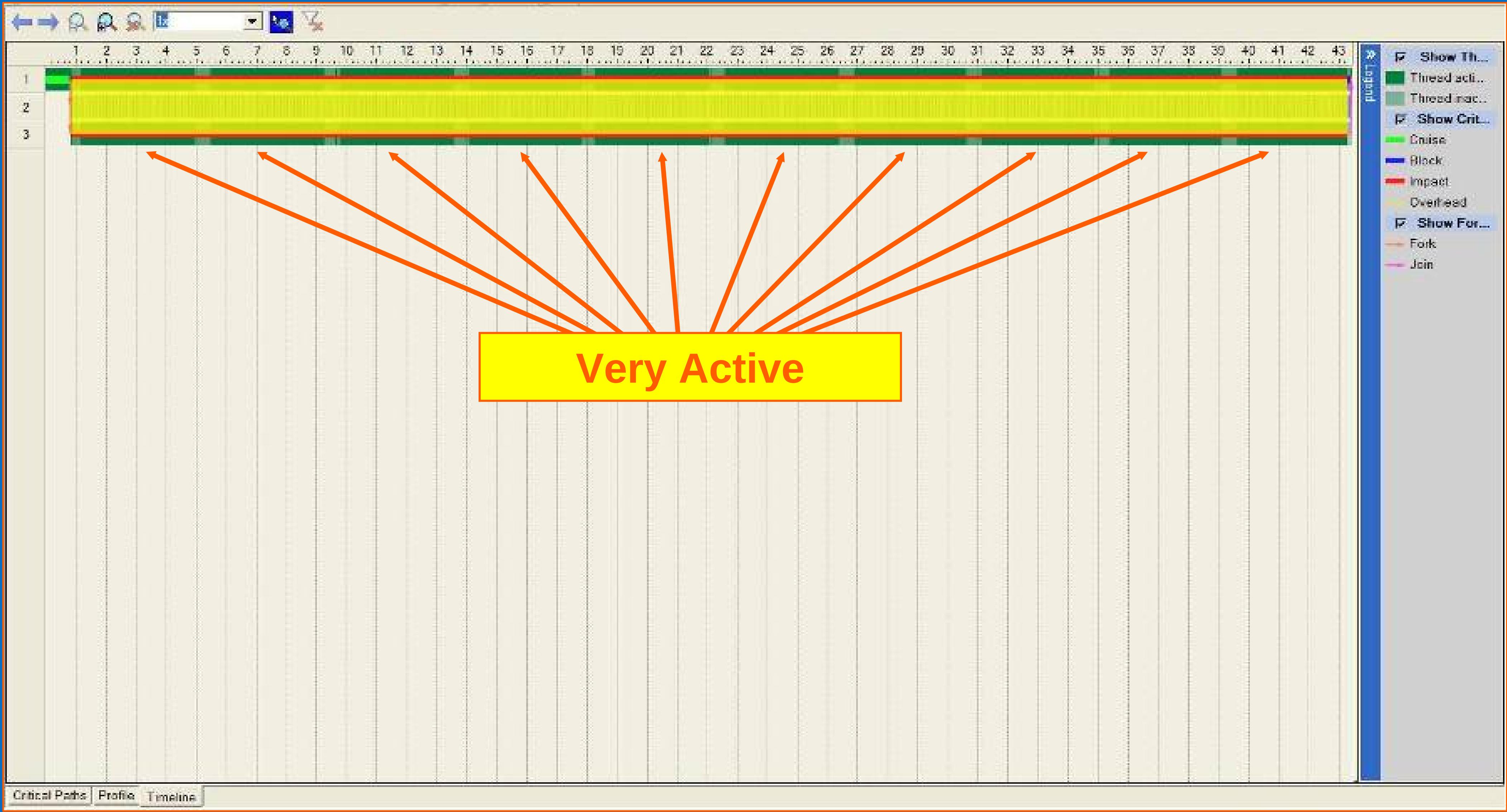
Data | Threads | Regions | Summary

Thread Profiler (Explicit Threads)

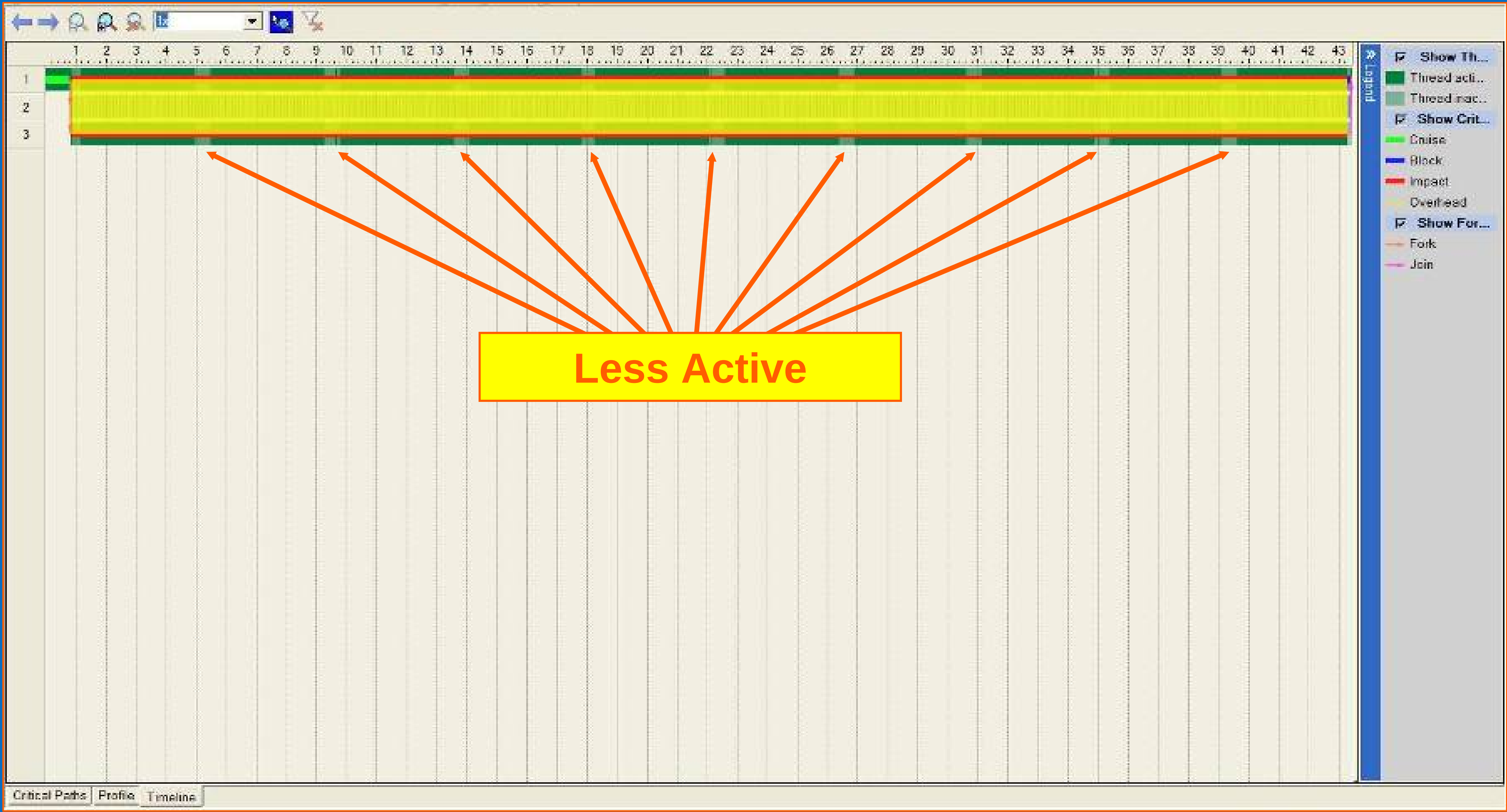


Gives a high level summary of execution

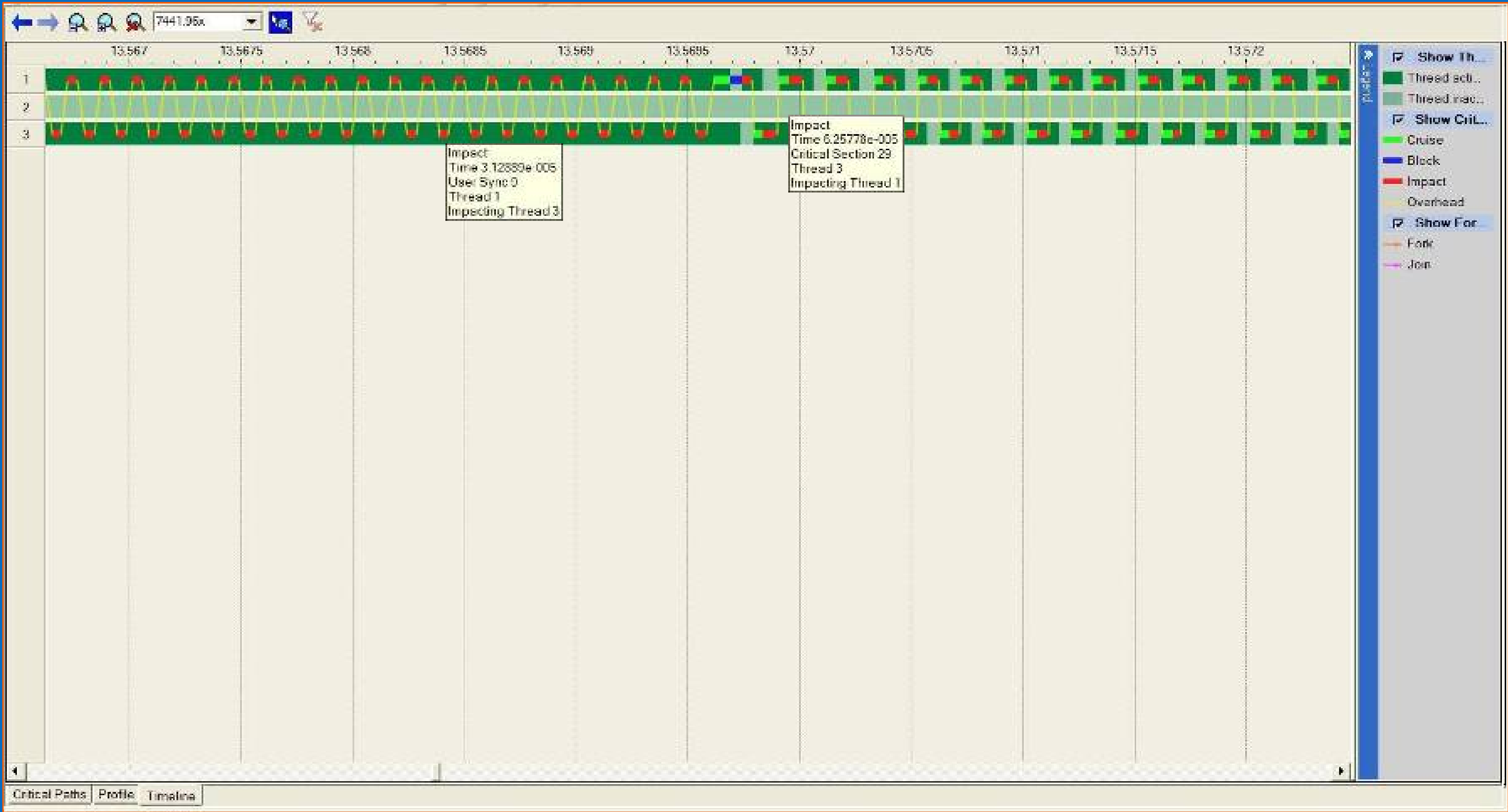
Thread Profiler (Explicit Threads)



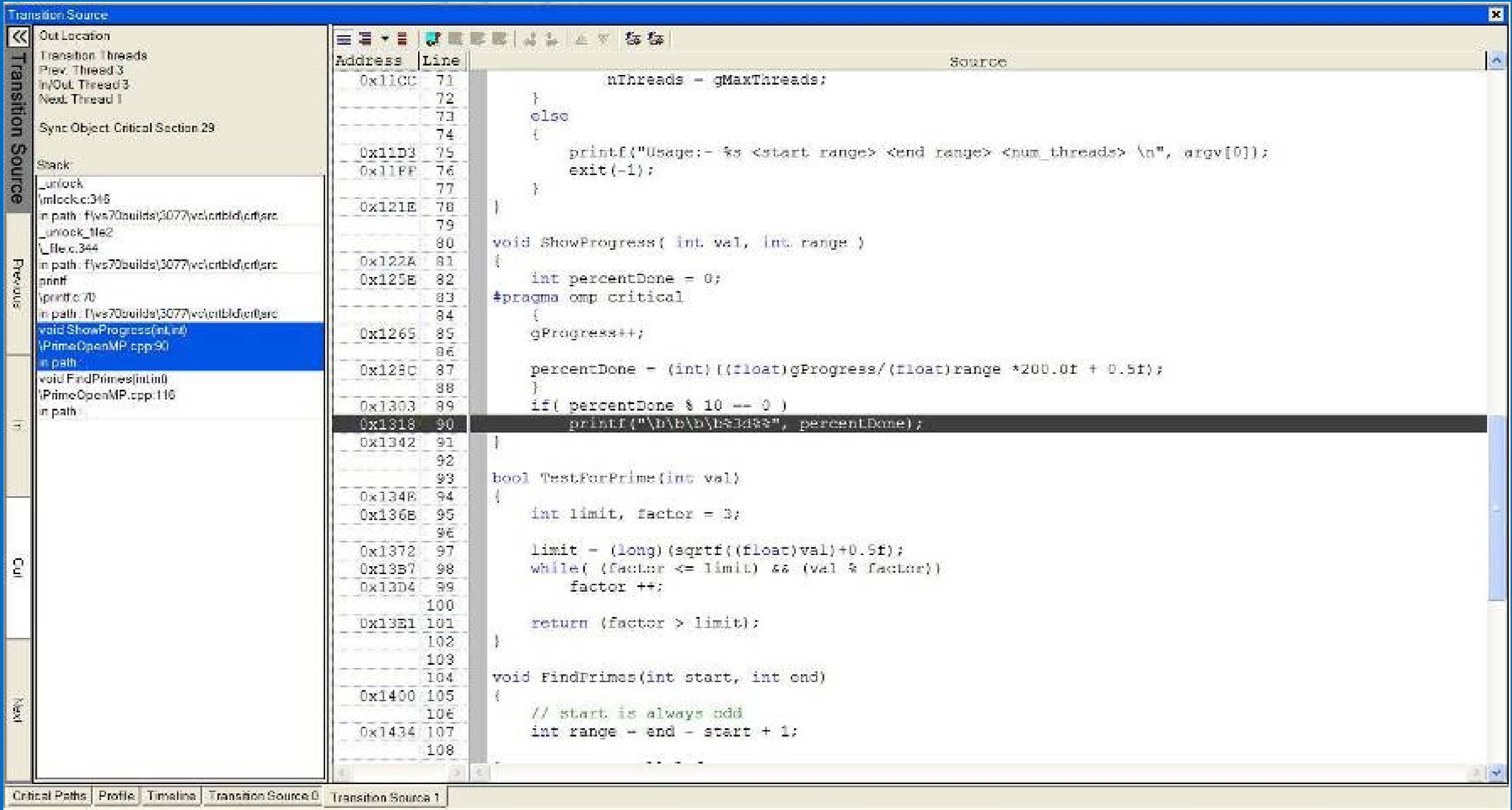
Thread Profiler (Explicit Threads)



Thread Profiler (Explicit Threads)



Thread Profiler (Explicit Threads)



Performance

This implementation has implicit synchronization calls

This limits scaling performance due to the resulting context switches


```
    if( percentDone % 10 == 0 )
        printf("\b\b\b\b%3d%%", percentDone);
}
```

Back to the design stage

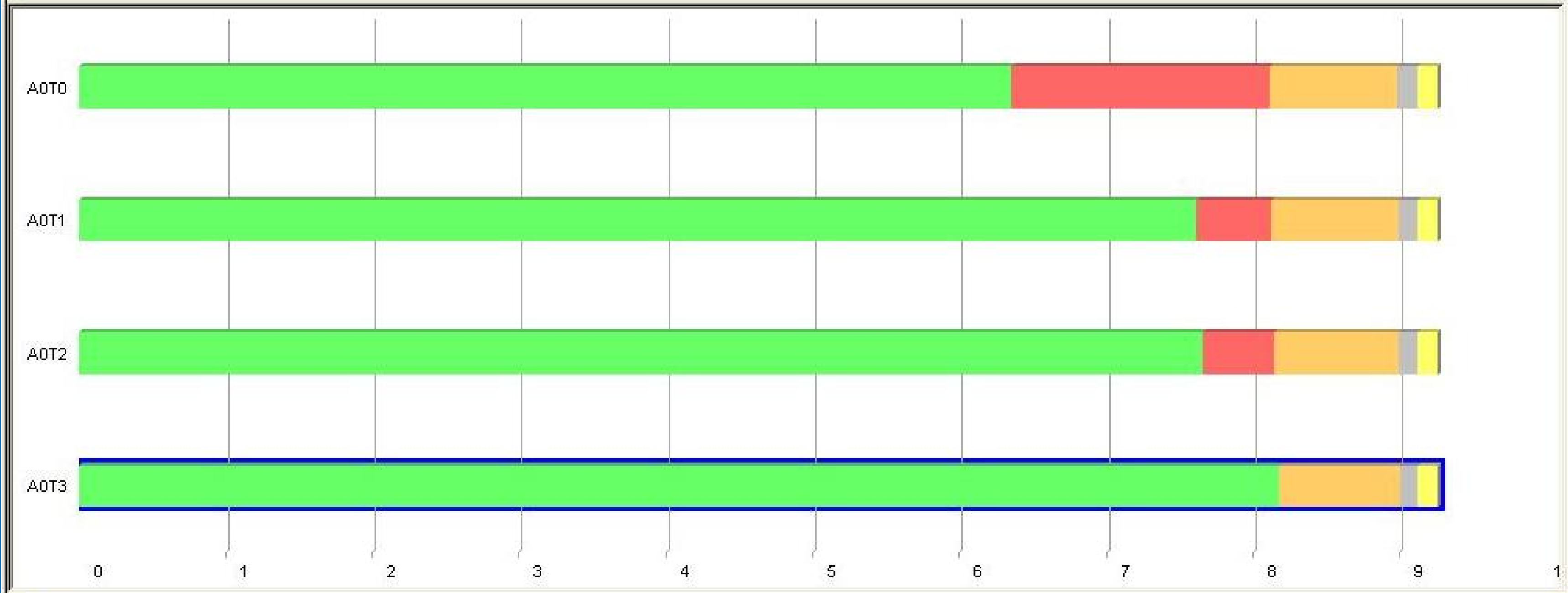
Demo 6

Use Thread Profiler to analyze threaded

application

- Use **/Qopenmp_profile** to compile and link
- Create Thread Profiler Activity (for explicit threads)
- Run application in Thread Profiler
- Find the source line that is causing the threads to be inactive

Four Thread Example



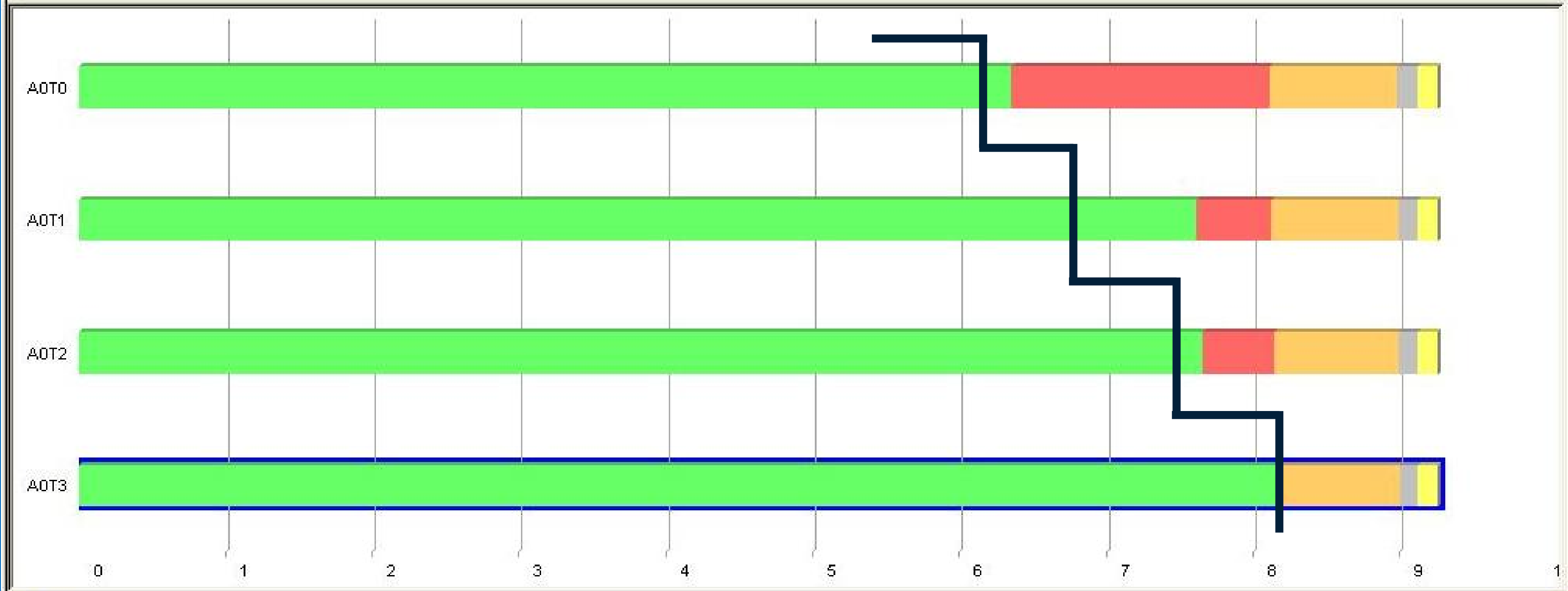
Legend

Label	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0T0	9.267	6.362	0.000	1.753	0.000	0.876	0.133	0.143	0.000
A0T1	9.266	7.618	0.000	0.516	0.000	0.861	0.131	0.140	0.000
A0T2	9.266	7.662	0.000	0.490	0.000	0.845	0.129	0.140	0.000
A0T3	9.266	8.177	0.000	0.000	0.000	0.827	0.123	0.139	0.000

1 runs 1 showing, 3 regions 3 showing

Data | **Threads** | Regions | Summary

Four Thread Example



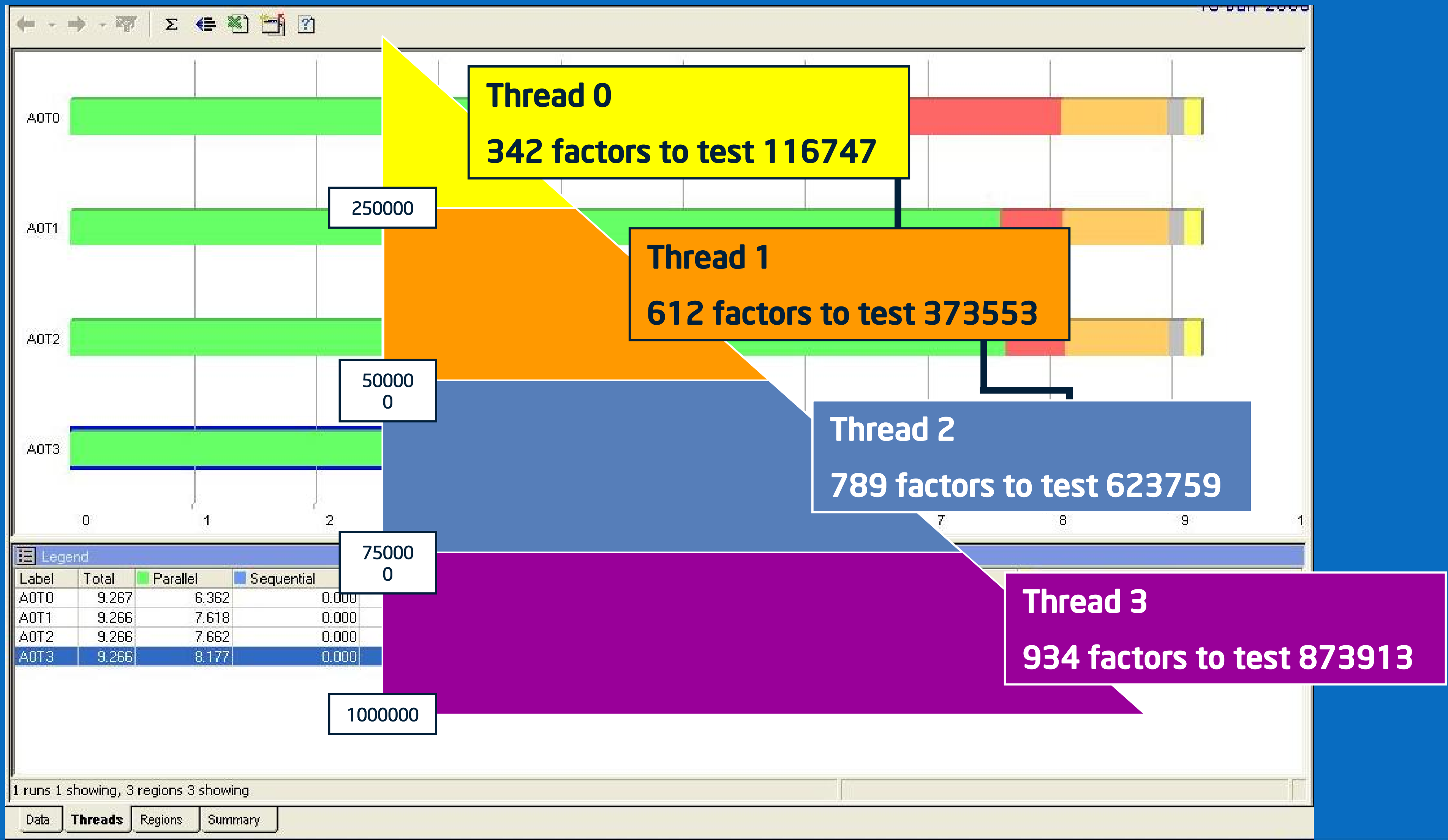
Legend

Label	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0T0	9.267	6.362	0.000	1.753	0.000	0.876	0.133	0.143	0.000
A0T1	9.266	7.618	0.000	0.516	0.000	0.861	0.131	0.140	0.000
A0T2	9.266	7.662	0.000	0.490	0.000	0.845	0.129	0.140	0.000
A0T3	9.266	8.177	0.000	0.000	0.000	0.827	0.123	0.139	0.000

1 runs 1 showing, 3 regions 3 showing

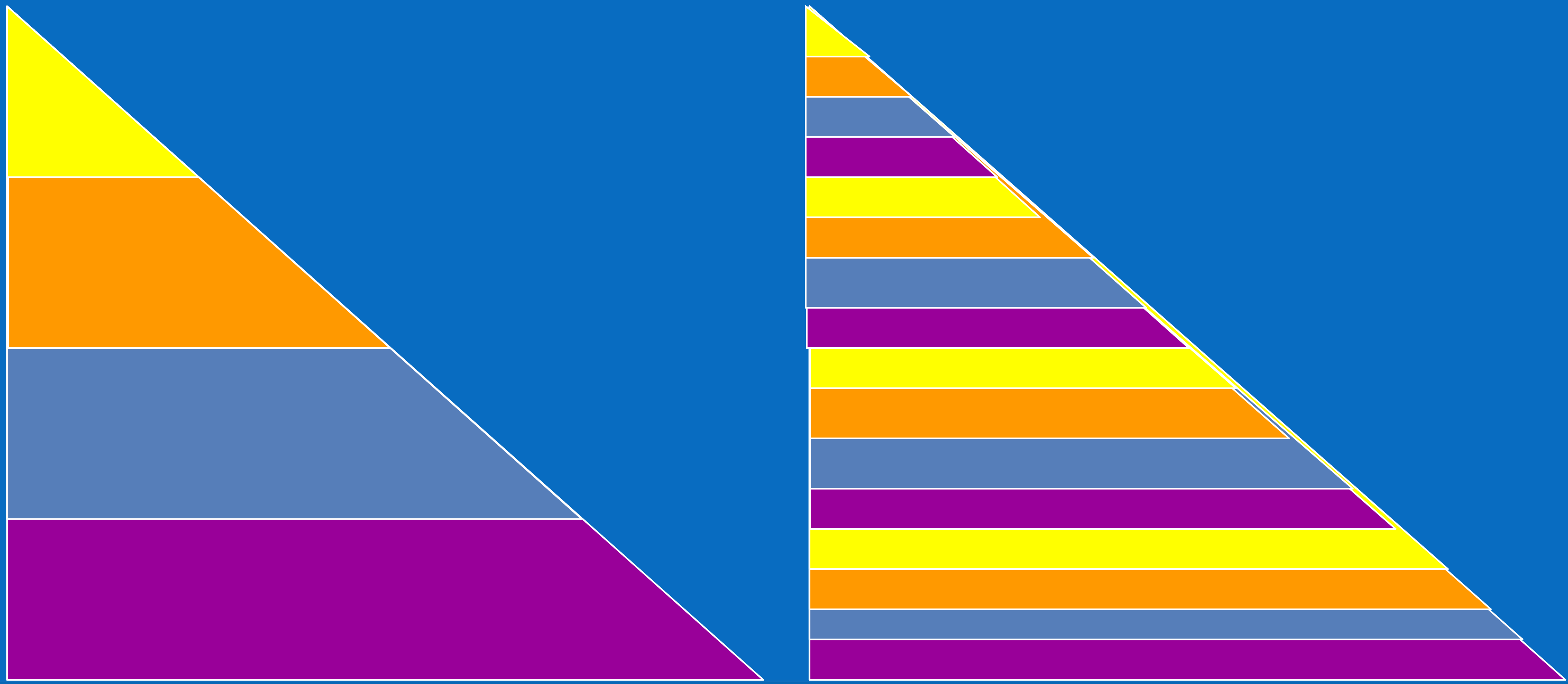
Data | **Threads** | Regions | Summary

Four Thread Example



Fixing the Load Imbalance

Distribute the work more evenly



Fixing the Load Imbalance

Distribute the work more evenly

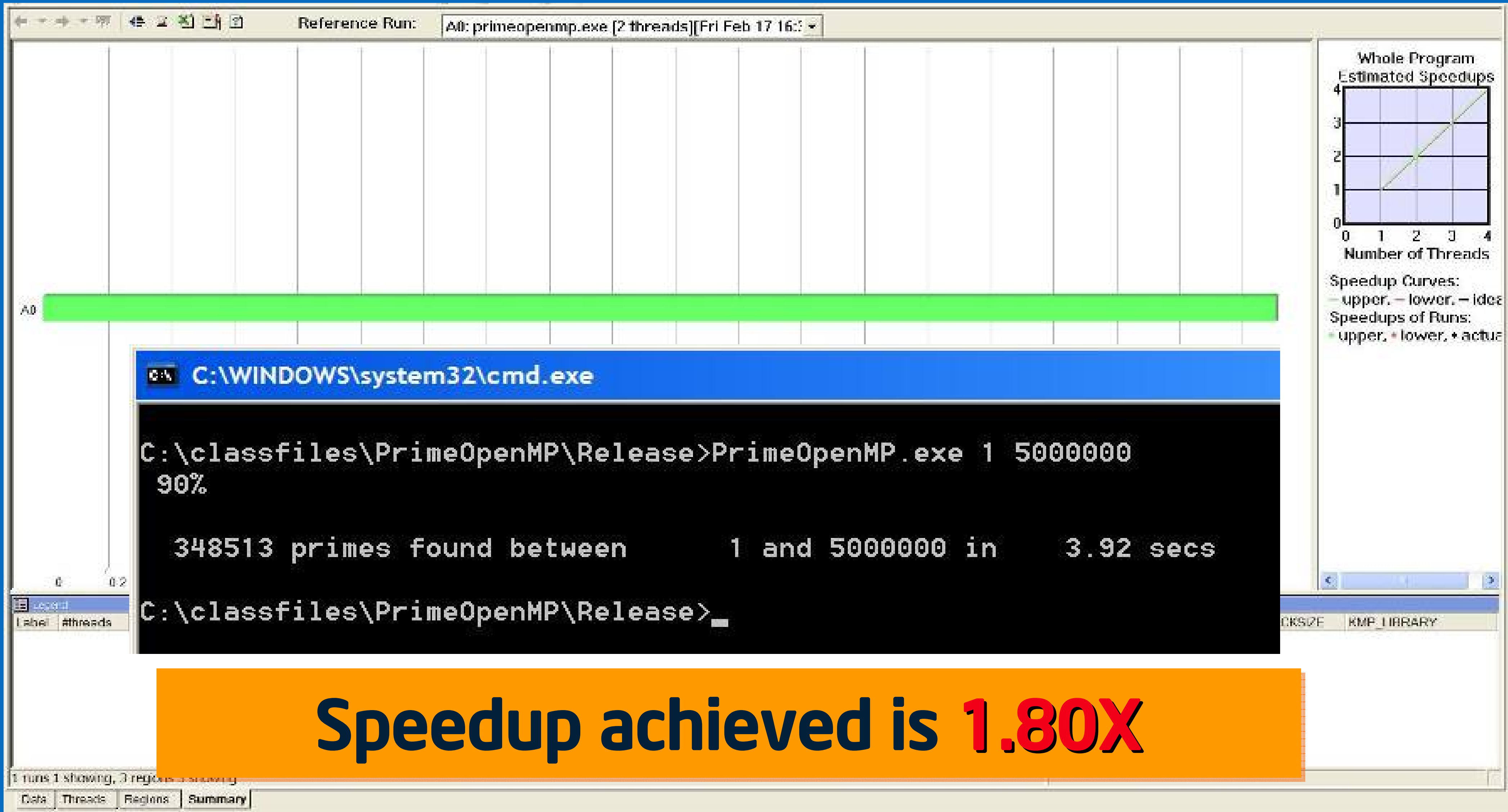
```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;

#pragma omp parallel for schedule(static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;
        ShowProgress(i, range);
    }
}
```

Demo 7

Modify code for better load balance

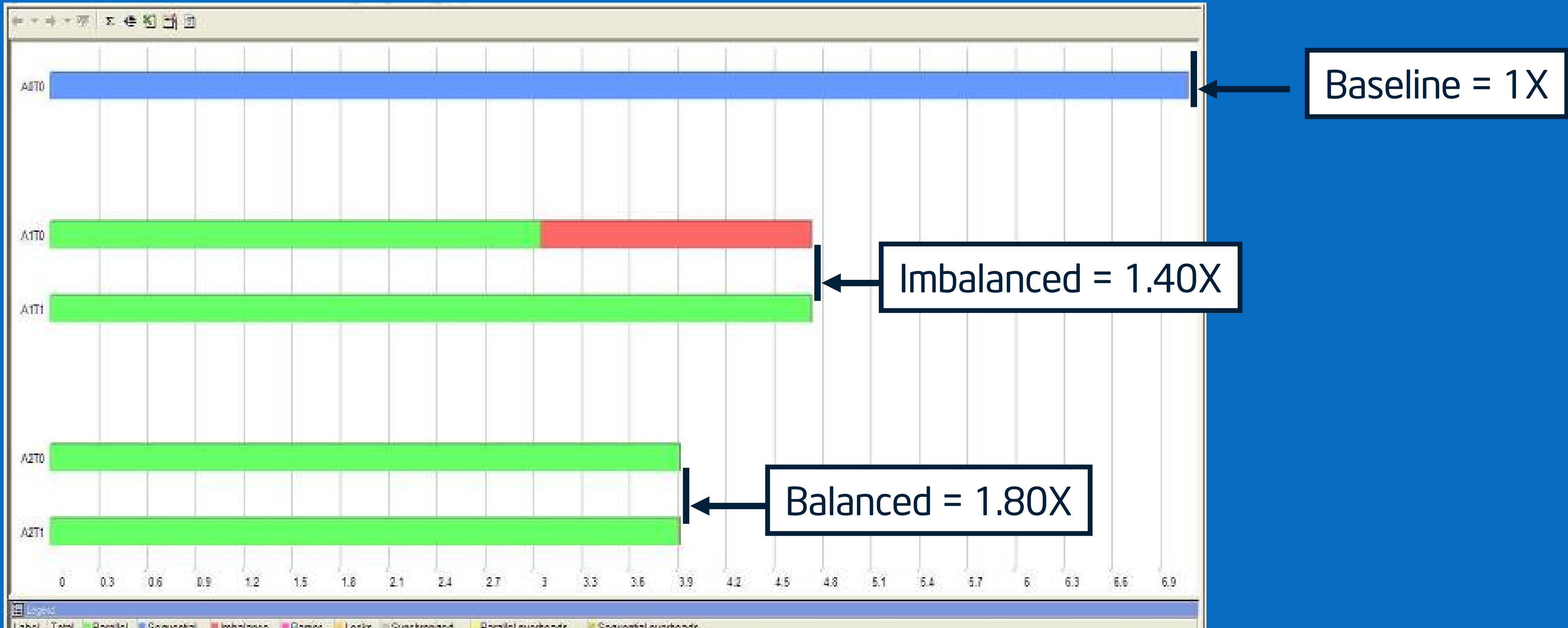
- Add **schedule (static, 8)** clause to OpenMP **parallel for** pragma
- Re-compile and run code
- What is speedup from serial version now?

Final Thread Profiler Run



Speedup achieved is 1.80X

Comparative Analysis



Threading applications require multiple iterations of going through the software development cycle

Threading Methodology
What's Been Covered

Four step development cycle for writing threaded code from serial and the Intel® tools that support each step

Analysis

Design (Introduce Threads)

Debug for correctness

Tune for performance

Threading applications require multiple iterations of designing, debugging and performance tuning steps

Use tools to improve productivity

Leap ahead™