Threaded Programming Methodology

Rama Malladi

Application Engineer

Software & Services Group

Optimizing Performance of Parallel Programs on Emerging
Multi-Core Processors & GPUs (OPECG-2009

Objectives

After completion of this module you will

- Learn how to use Intel Software Development Products for multi-core programming and optimizations

- Be able to rapidly prototype and estimate the effort required to thread time consuming regions

Agenda

A Generic Development Cycle

Case Study: Prime Number Generation

Common Performance Issues

What is Parallelism?

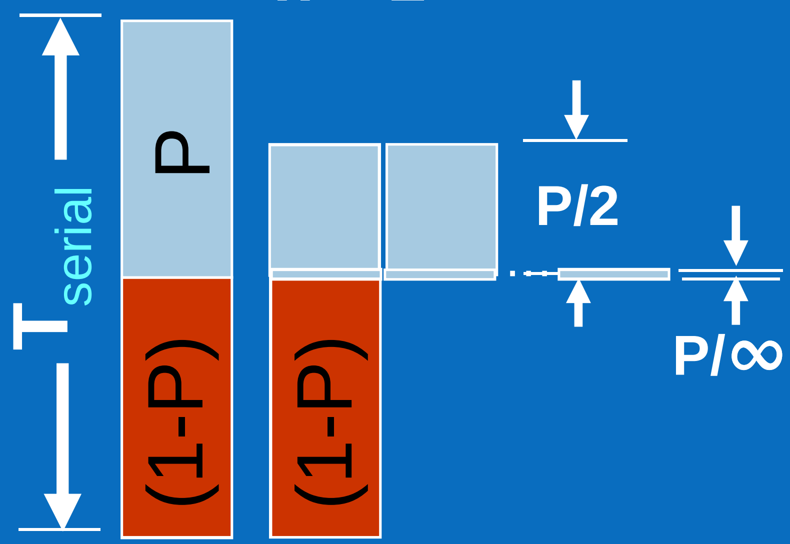
Two or more processes or threads execute at the same time

Parallelism for threading architectures

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

Amdahl's Law

Describes the upper bound of parallel execution speedup



0.5 + 0.25

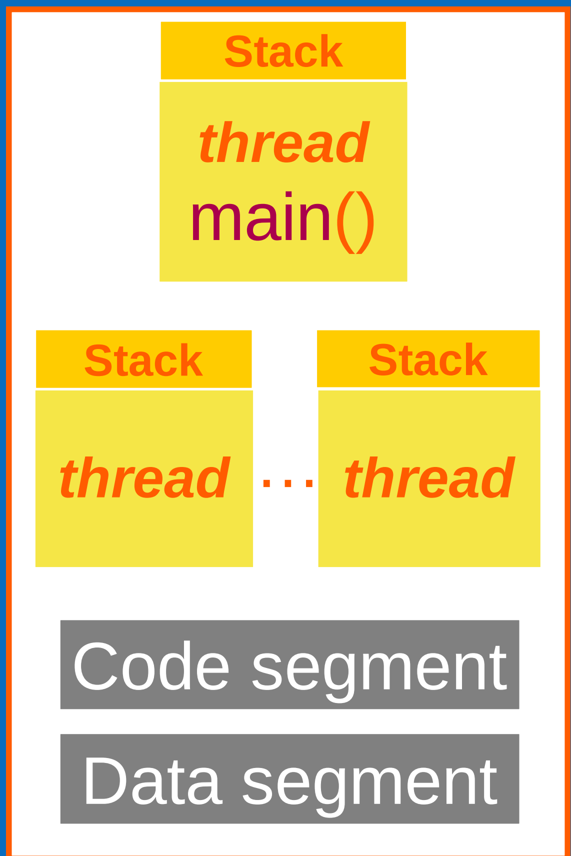
$$T_{parallel} = \{(1-P) + P/n\}\, T_{serial}$$

n = number of processors

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

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 2
    int limit, factor = 2;
    limit = (long)(sqrtf((float)val)+0.5f);
                                    factor) )
}

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);
    }
}
```

```
user17@xeon-linux-production:~/PrimeSingle
[user17@xeon-linux-production PrimeSingle]$ ./PrimeSingle 1 20
100%

      8 primes found between       1 and      20 in     0.00 secs
[user17@xeon-linux-production PrimeSingle]$
```

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 (Pthreads*, Win32*)

Debug for correctness

–Intel® Thread Checker

–Intel Debugger

Tune for performance

–Thread Profiler

–VTune™ Performance Analyzer

Analysis - Sampling

Use VTune Samplin

```
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);
}
```

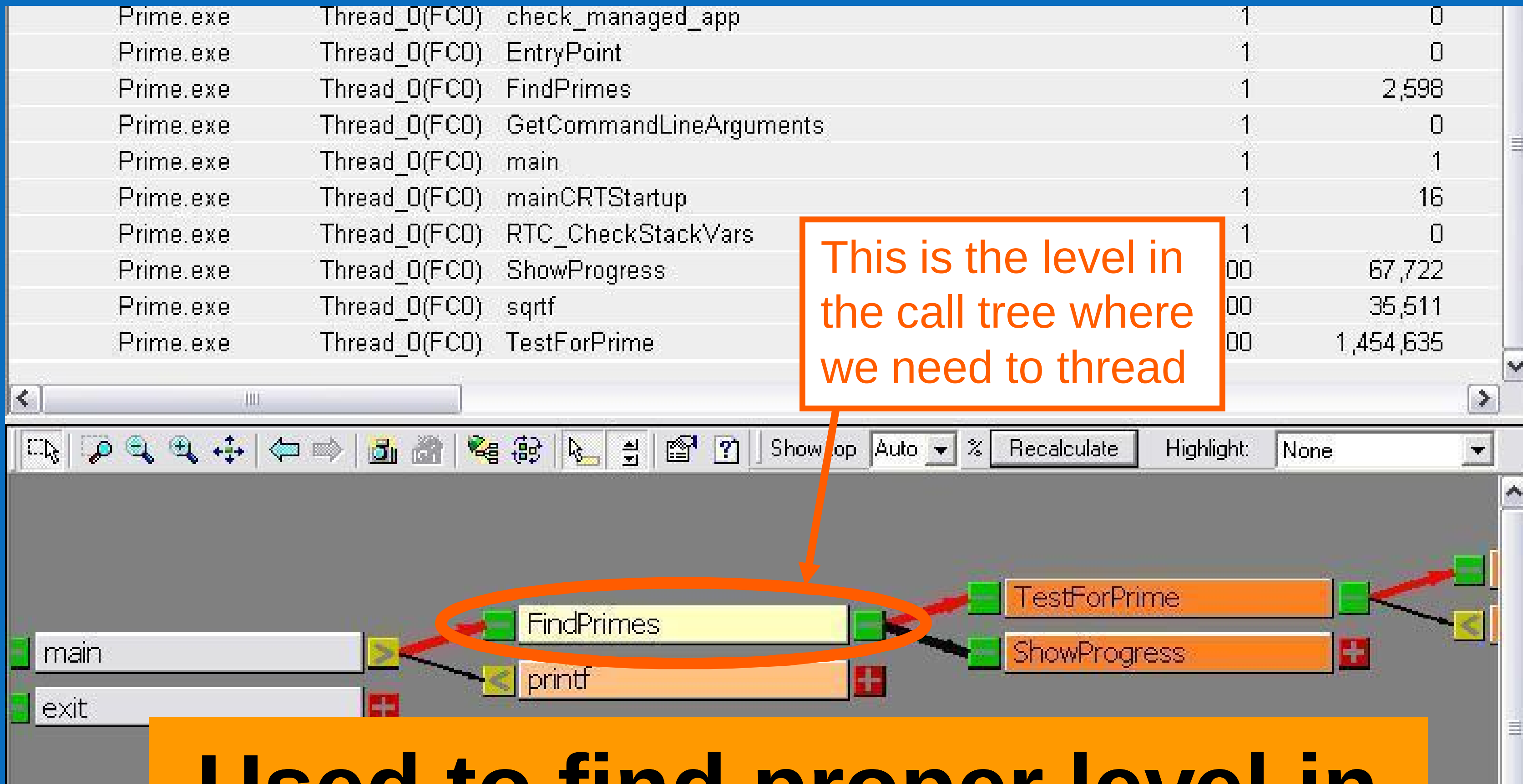
Let'

- P


```
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) )
            globalPrimes[gPrimesFound++] = i;
```

Identifies the time consuming regions

Analysis - Call Graph



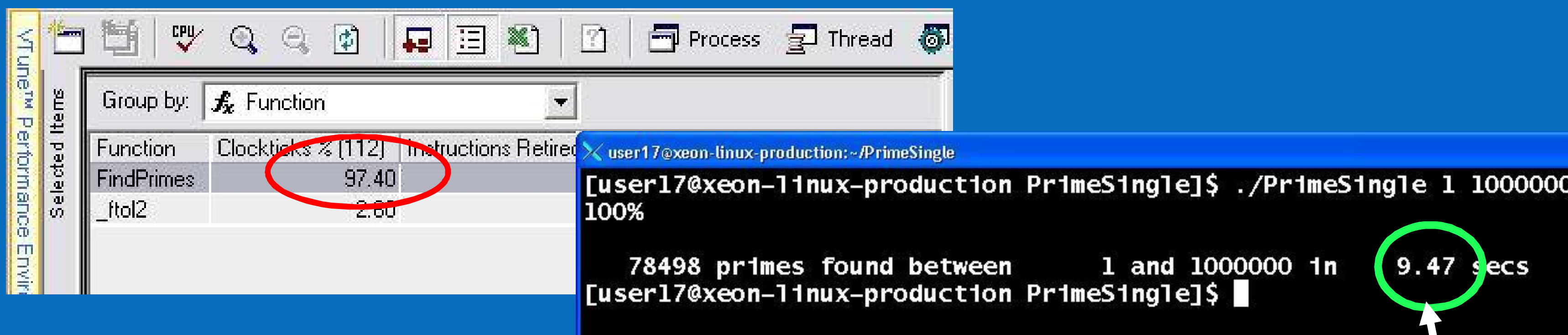
Used to find proper level in the call-tree to thread

Analysis

Where to thread?

- **FindPrimes()**

Is it worth threading a selected region?



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

Baseline measurement

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

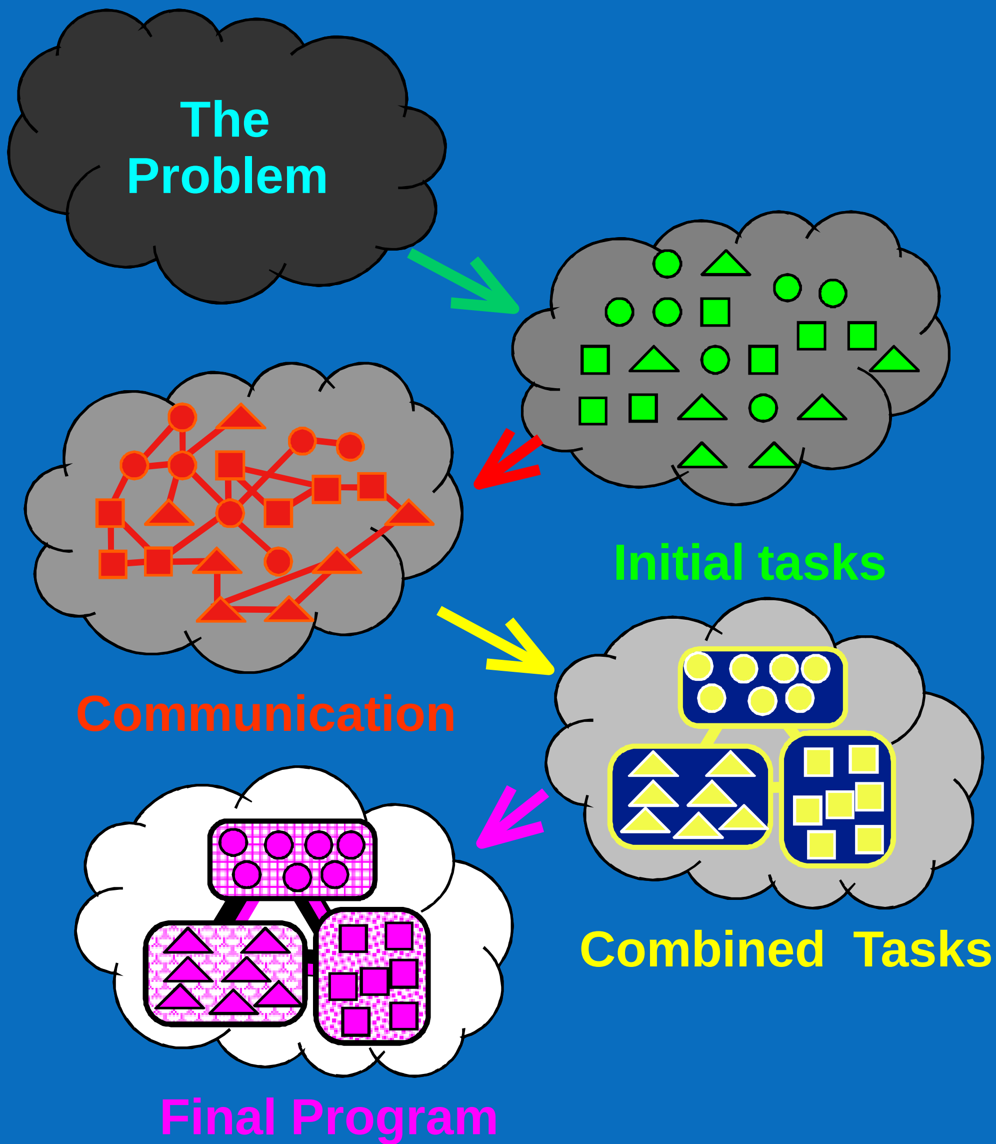
- Determine amount and pattern of communication

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

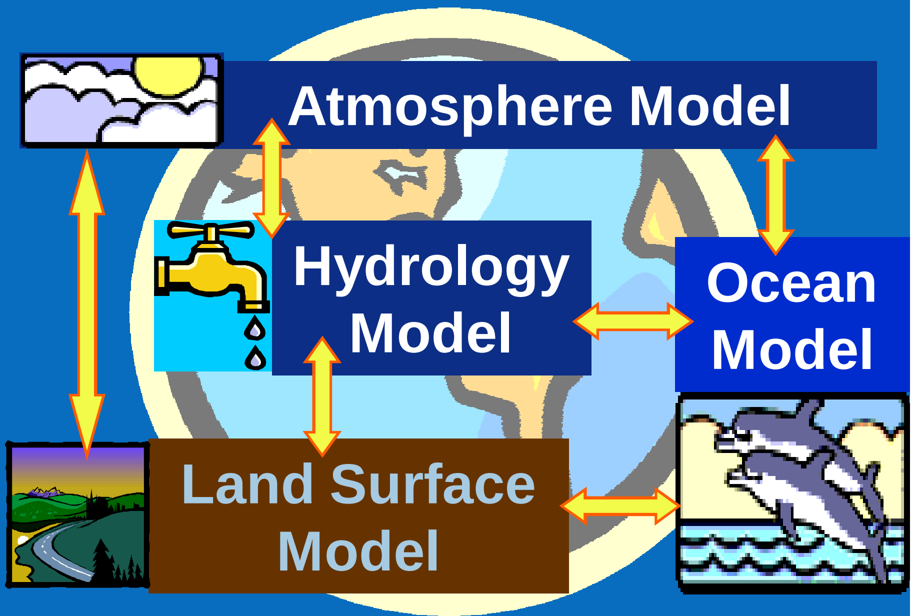
Data Decomposition

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

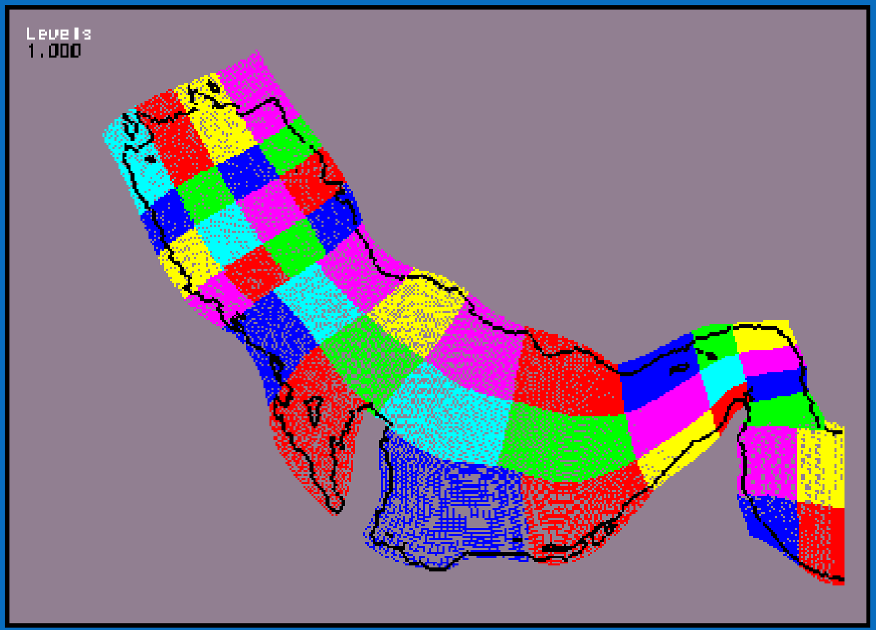
Decomposition Methods

Functional Decomposition

- Focusing on computations can reveal structure in a problem



Grid reprinted with permission of Dr. Phu V. Luong, Coastal and Hydraulics Laboratory, ERDC



Domain Decomposition

- Focus on largest or most frequently accessed data structure
- Data Parallelism
  - Same operation applied to all data

Pipelined Decomposition

Computation done in independent stages

Functional decomposition

- Threads are assigned stage to compute
- Automobile assembly line

Data decomposition

- Thread processes all stages of single instance
- One worker builds an entire car

LAME Encoder Example

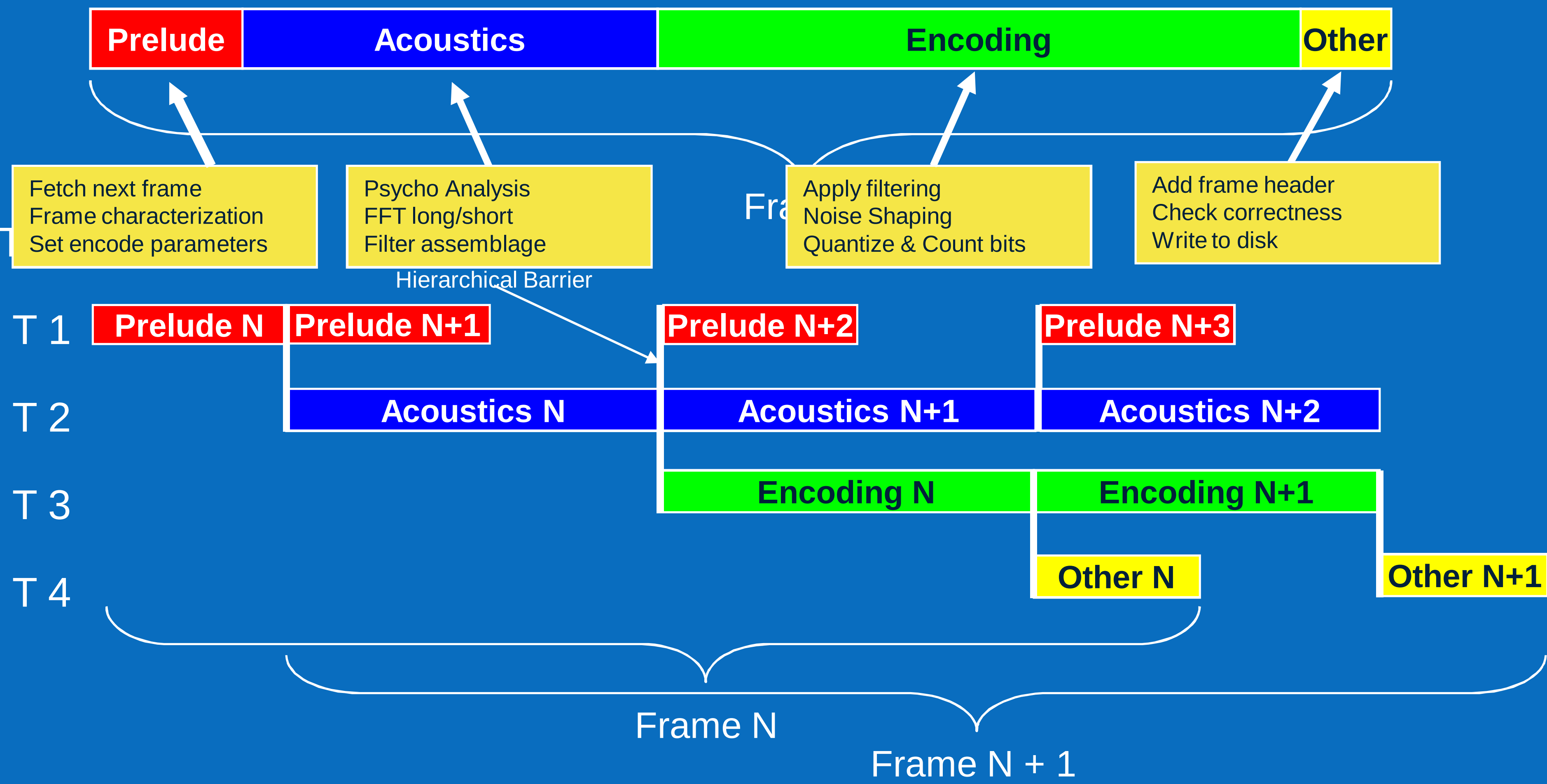
LAME MP3 encoder

- Open source project
- Educational tool used for learning

The goal of project is

- To improve the psychoacoustics quality
- To improve the speed of MP3 encoding

LAME Pipeline Strategy



Design

What is the expected benefit?

Speedup(2P) = 100/(97/2+3) = ~1.94X

How do you determine this with the least effort?

Rapid prototyping with OpenMP

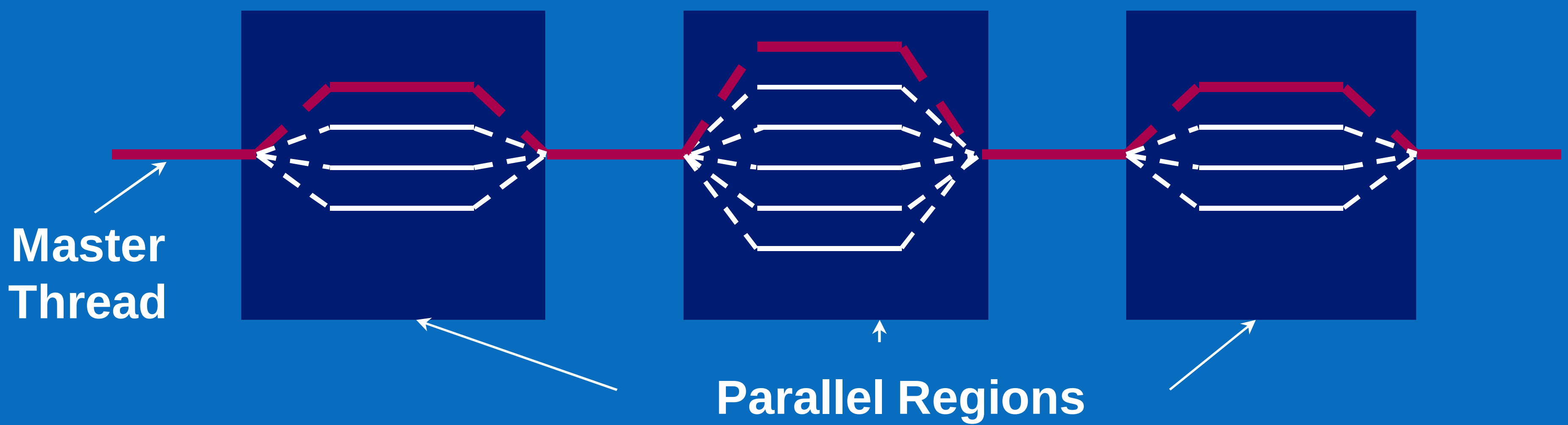
How long would it take to thread?

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 ){
         f( TestForPrime(
              globalPrimes[                      ] =
i;
         Sho
   }
```

OpenMP

Divide iterations of the for loop

Create threads here for this parallel region

user17@xeon-linux-production:~/PrimeOpenMP

```
[user17@xeon-linux-production PrimeOpenMP]$ ./PrimeOpenMP 1 1000000
 90%

   78197 primes found between       1 and 1000000 in    8.48 secs
[user17@xeon-linux-production PrimeOpenMP]$
```


OPECG 2009

Design

What is the expected benefit?

How do you determine this with the least effort?

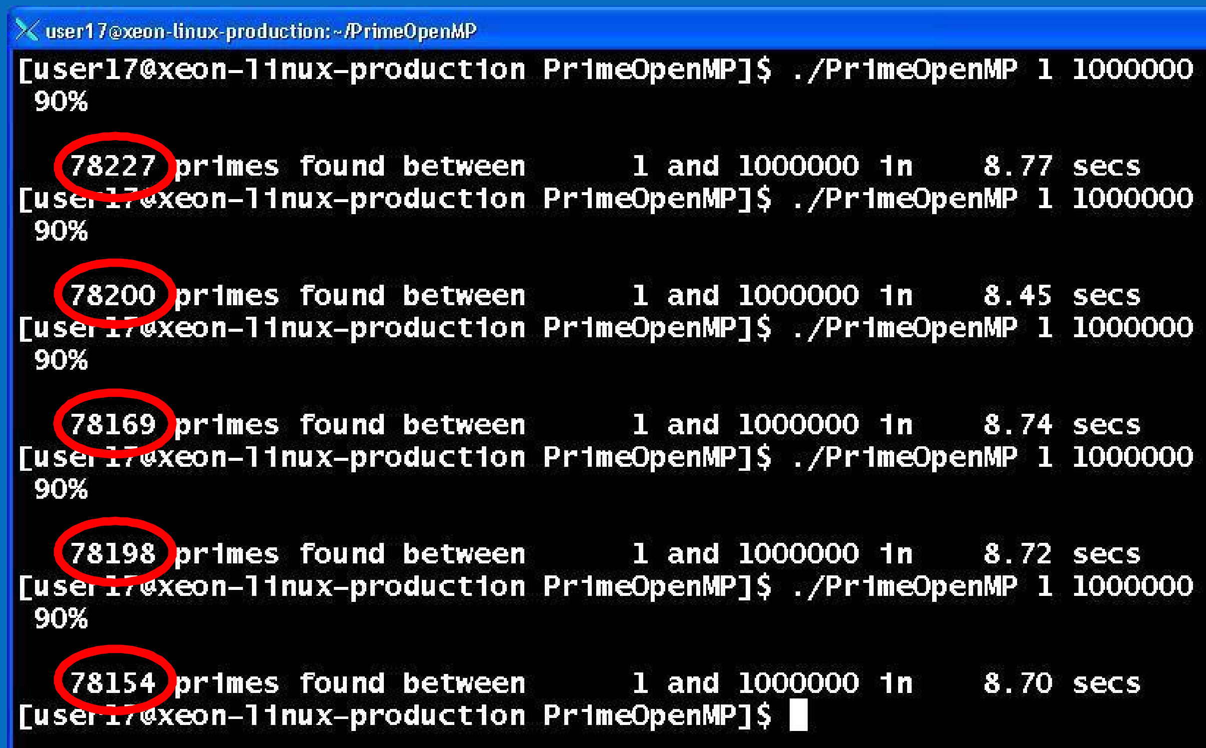
Speedup of 1.12X (less than 1.94X)

How long would it take to thread?

How much re-design/effort is required?

Is this the best scaling possible?

Debugging for Correctness

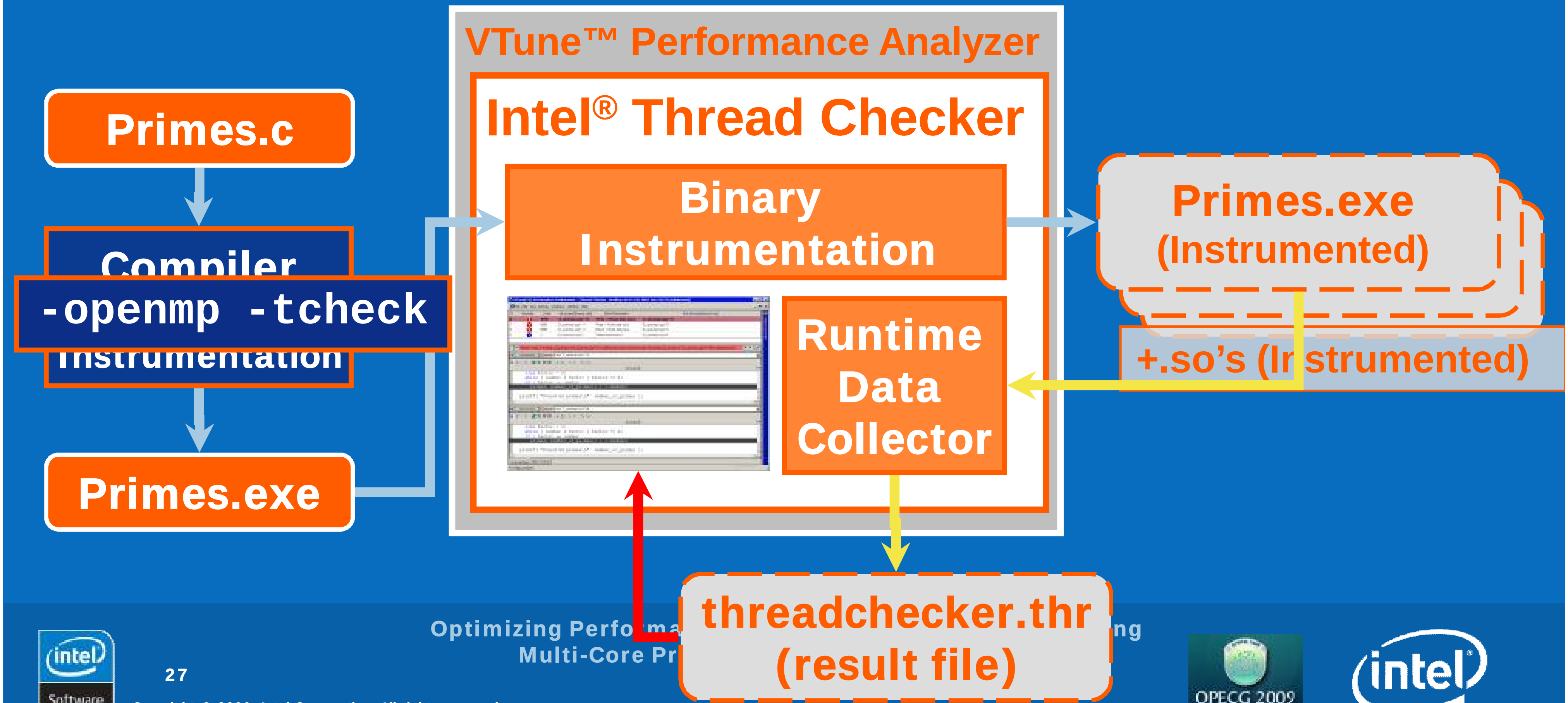


Is this threaded implementation right?

No! The answers are different each time …

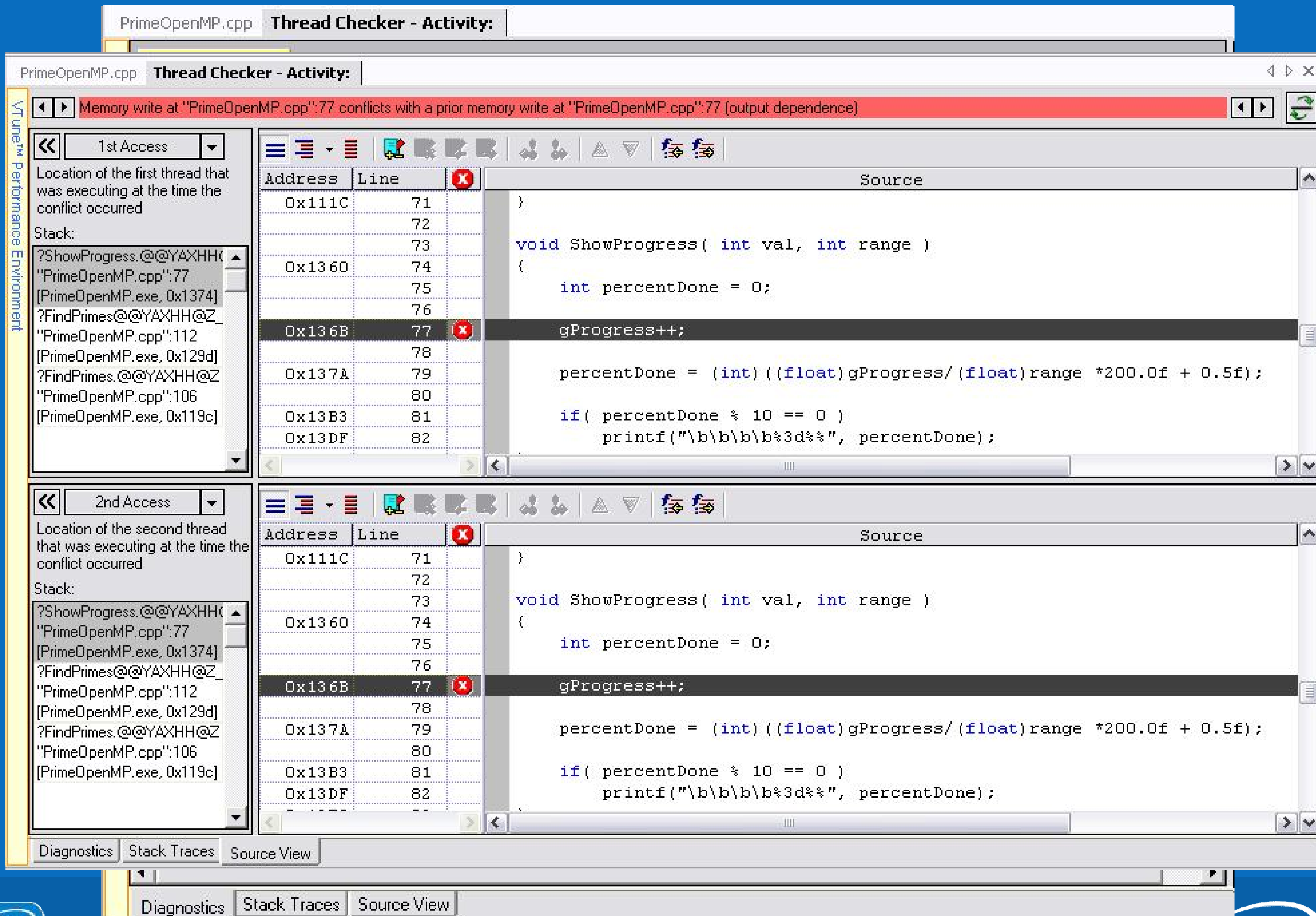
Debugging for Correctness

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



Debugging for Correctness

If application has source instrumentation, it can be run from command prompt


PrimeOpenMP.cpp
Thread Checker - Activity:
VTune™ Performance Environment
Memory write at "PrimeOpenMP.cpp":77 conflicts with a prior memory write at "PrimeOpenMP.cpp":77 (output dependence)
1st Access
Location of the first thread that was executing at the time the conflict occured
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 void ShowProgress(int val, int range)
0x1360 74 {
75 int percentDone = 0;
76
0x136B 77 gProgress++;
78
0x137A 79 percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);
80
0x13B3 81 if(percentDone % 10 == 0)
0x13DF 82 printf("\b\b\b\b%3d%%", percentDone);
2nd Access
Location of the second thread that was executing at the time the conflict occured
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 void ShowProgress(int val, int range)
0x1360 74 {
75 int percentDone = 0;
76
0x136B 77 gProgress++;
78
0x137A 79 percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);
80
0x13B3 81 if(percentDone % 10 == 0)
0x13DF 82 printf("\b\b\b\b%3d%%", percentDone);
Diagnostics
Stack Traces
Source View

Debugging for Correctness

How much re-design/effort is required?

Thread Checker reported only 2 dependencies, so effort required should be low

How long would it take to thread?

Debugging for Correctness

```
#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

Correctness

Correct answer, but performance has slipped to ~1.03X !

```
user17@xeon-linux-production:~/PrimeOpenMP
[user17@xeon-linux-production PrimeOpenMP]$ ./PrimeOpenMP 1 1000000
100%

   78498 primes found between       1 and 1000000 in     9.18 secs
[user17@xeon-linux-production PrimeOpenMP]$
```

Is this the best we can expect from this algorithm?

No! From Amdahl's Law, we expect scaling 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

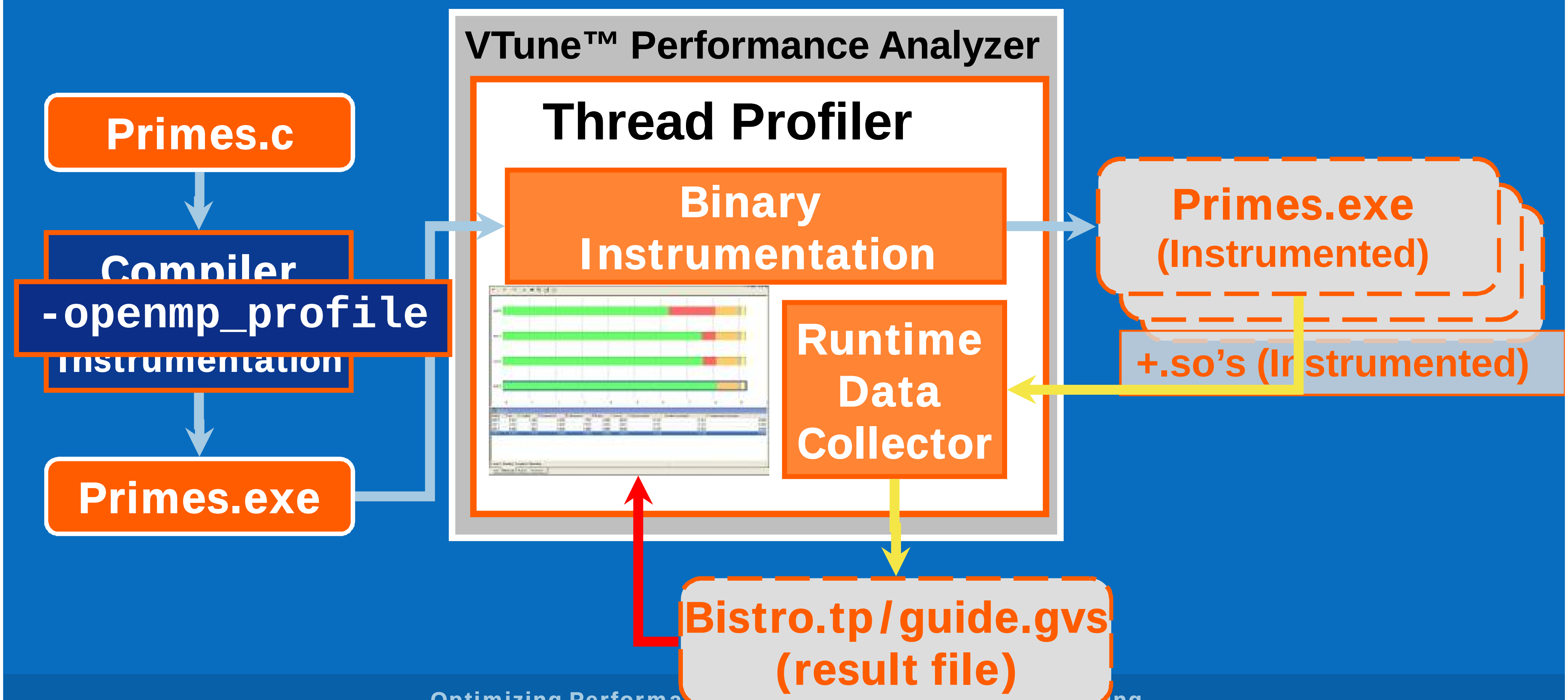
- Improper distribution of parallel work

Granularity

- No sufficient parallel work

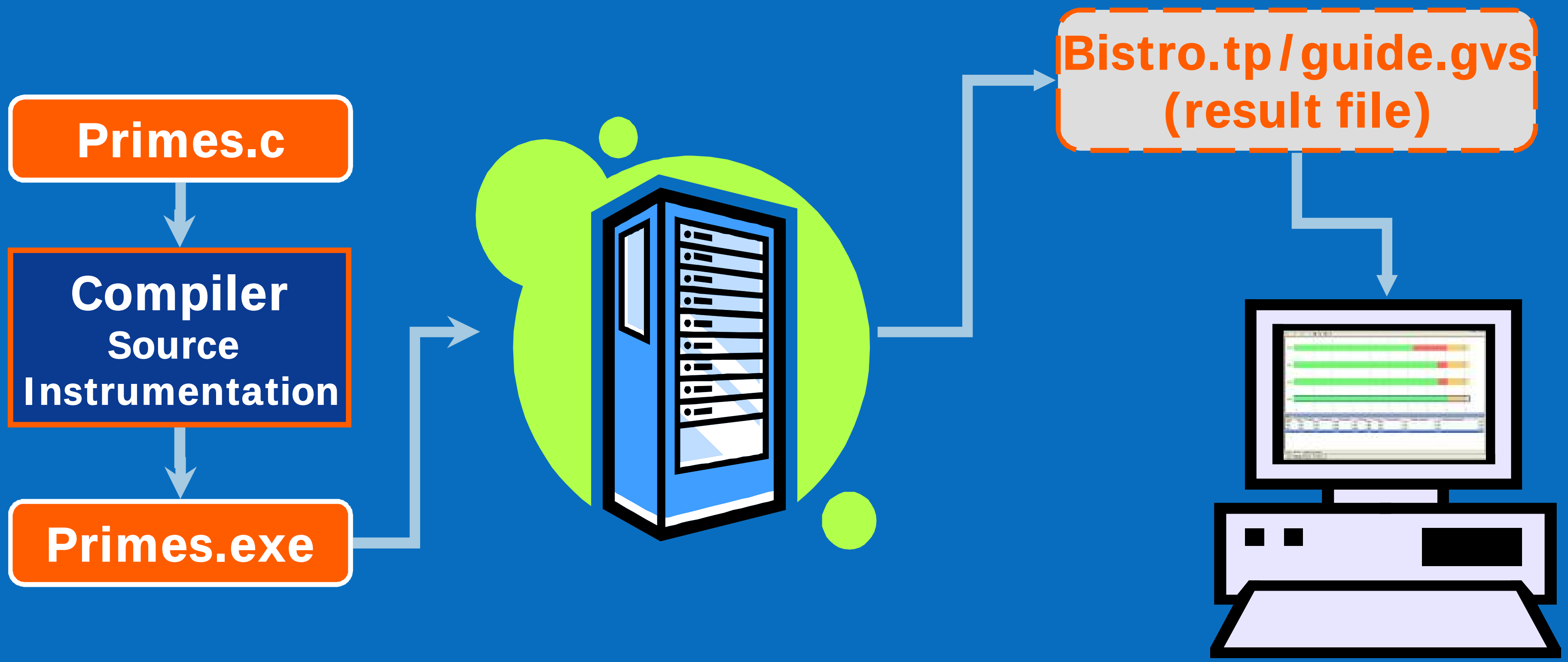
Tuning for Performance

Thread Profiler pinpoints performance bottlenecks in threaded applications

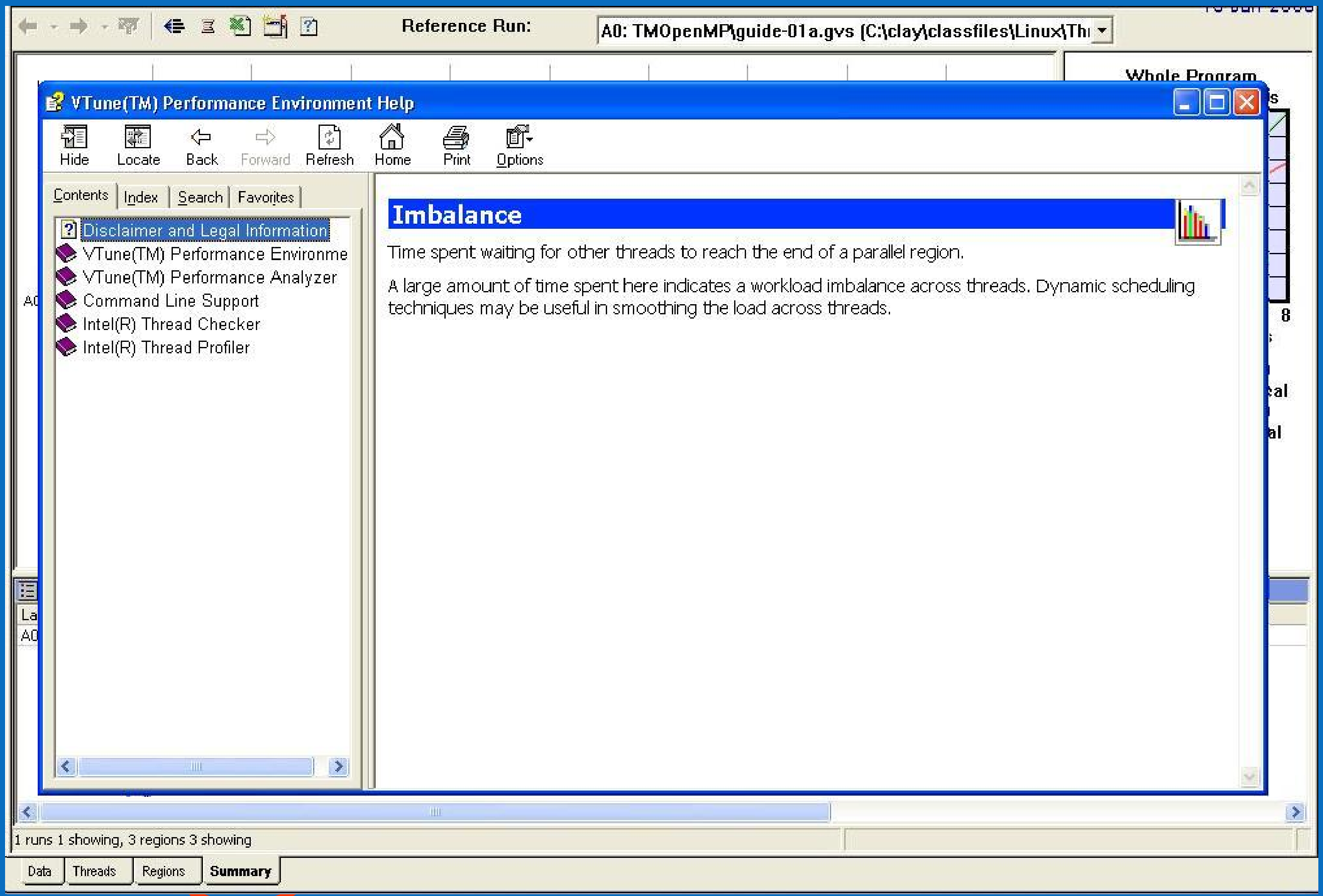


Tuning for Performance

If application has source instrumentation, it can be run from command prompt



Thread Profiler for OpenMP



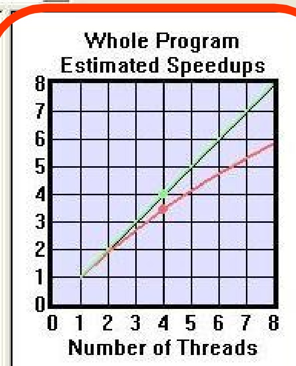
Thread Profiler for OpenMP

Speedup Graph

Estimates threading speedup and potential speedup

- **Based on Amdahl's Law computation**

Gives upper and lower bound estimates



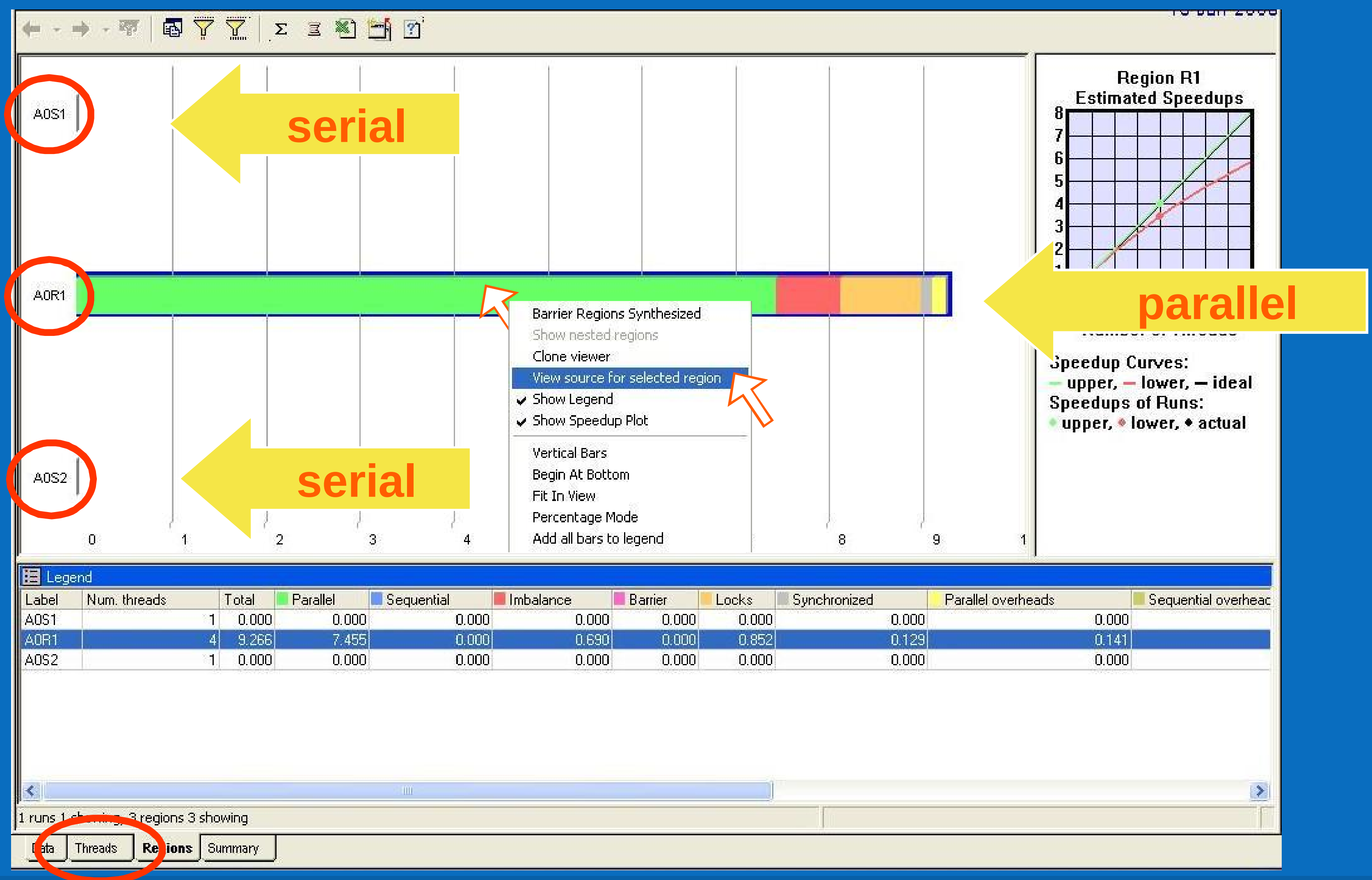
Reference Run: A0: TMOpenMP\guide-01a.gvs (C:\clay\classfiles\Linux\Thr

Label	#threads	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0	4	9.266	7.455	0.000	0.690	0.000	0.852	0.129	0.141	

1 runs 1 showing, 3 regions 3 showing

Data | Threads | Regions | Summary

Thread Profiler for OpenMP



Thread Profiler for OpenMP

Legend

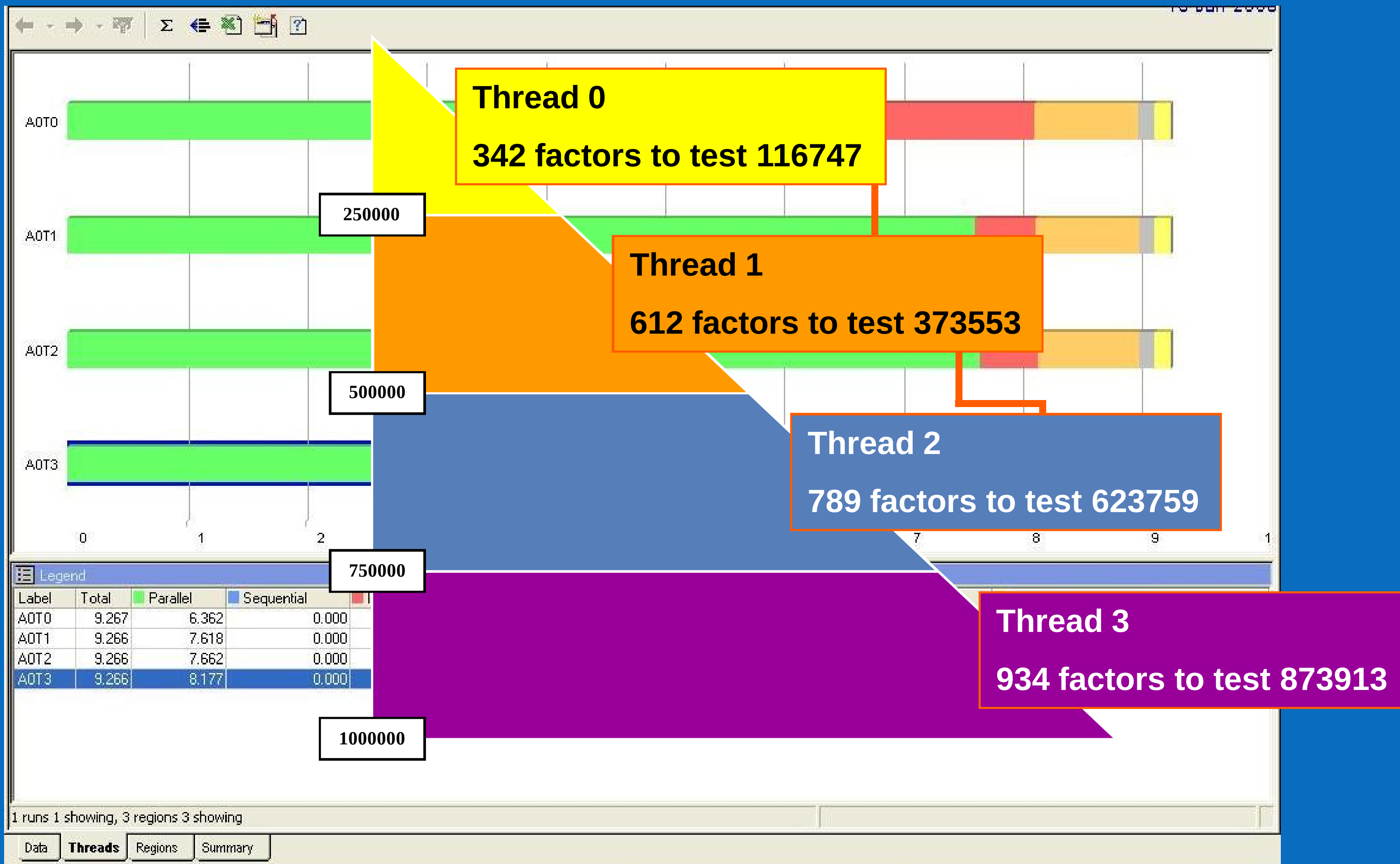
Label	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0T0	9.267	6.362	0.000	1.753	0.000	0.876	0.133	0.143	0.000
A0T1	9.266	7.618	0.000	0.516	0.000	0.861	0.131	0.140	0.000
A0T2	9.266	7.662	0.000	0.490	0.000	0.845	0.129	0.140	0.000
A0T3	9.266	8.177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

1 runs 1 showing, 3 regions 3 showing

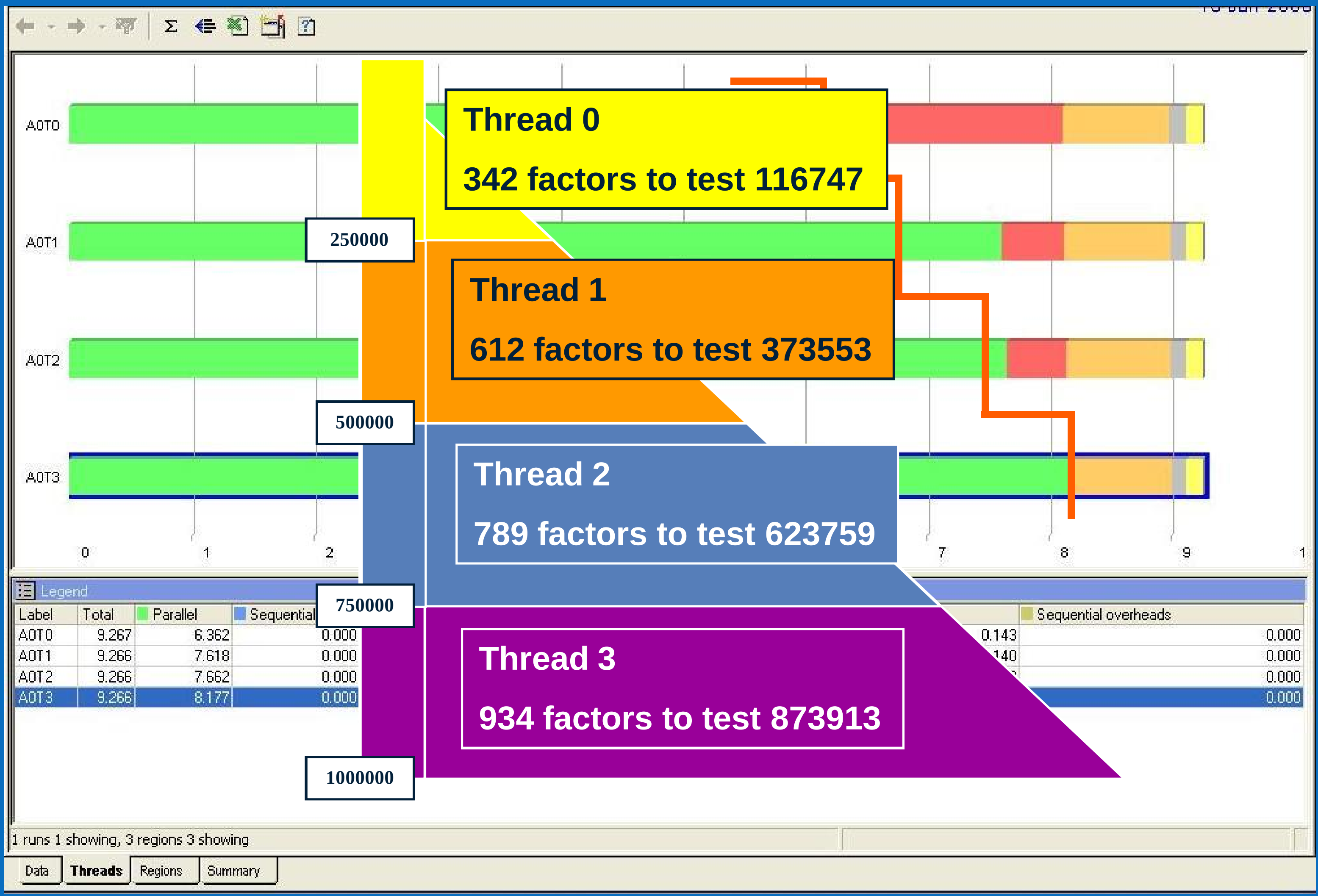
Data | Threads | Regions | Summary

OPECG 2009

Thread Profiler for OpenMP



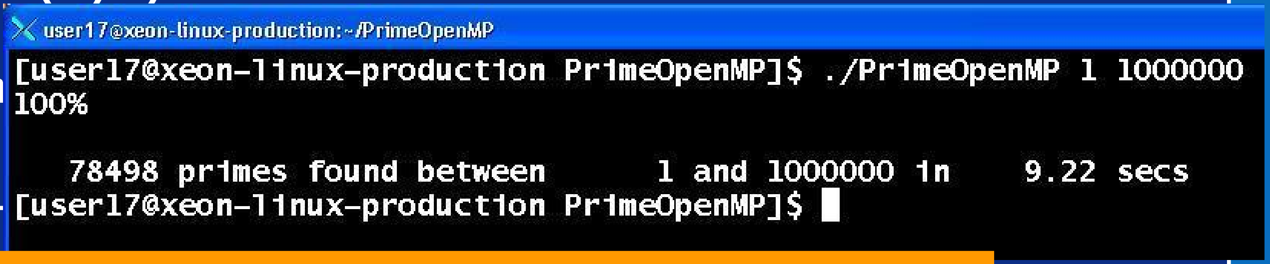
Thread Profiler for OpenMP



Fixing the Load Imbalance

Distribute the work more evenly

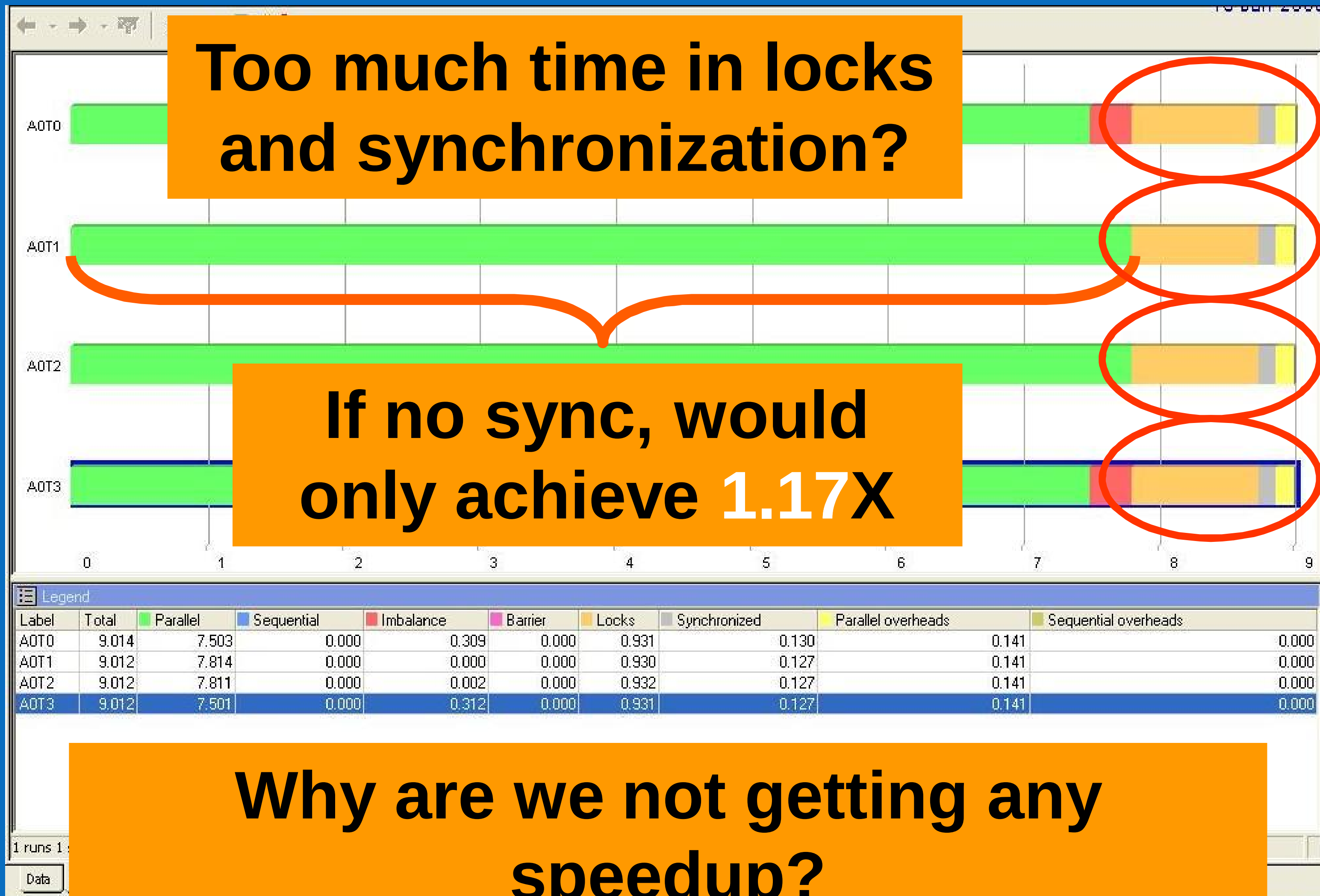
```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;

#pragma omp parallel for schedule(static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
#pragma omp critical
            globalPrim

           ShowProgress(i
    }
}
```



Scaling achieved is 1.02X

Back to Thread Profiler

Too much time in locks and synchronization?



Label	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads
A0T0	9.014	7.503	0.000	0.309	0.000	0.931	0.130	0.141	0.000
A0T1	9.012	7.814	0.000	0.000	0.000	0.930	0.127	0.141	0.000
A0T2	9.012	7.811	0.000	0.002	0.000	0.932	0.127	0.141	0.000
A0T3	9.012	7.501	0.000	0.312	0.000	0.931	0.127	0.141	0.000

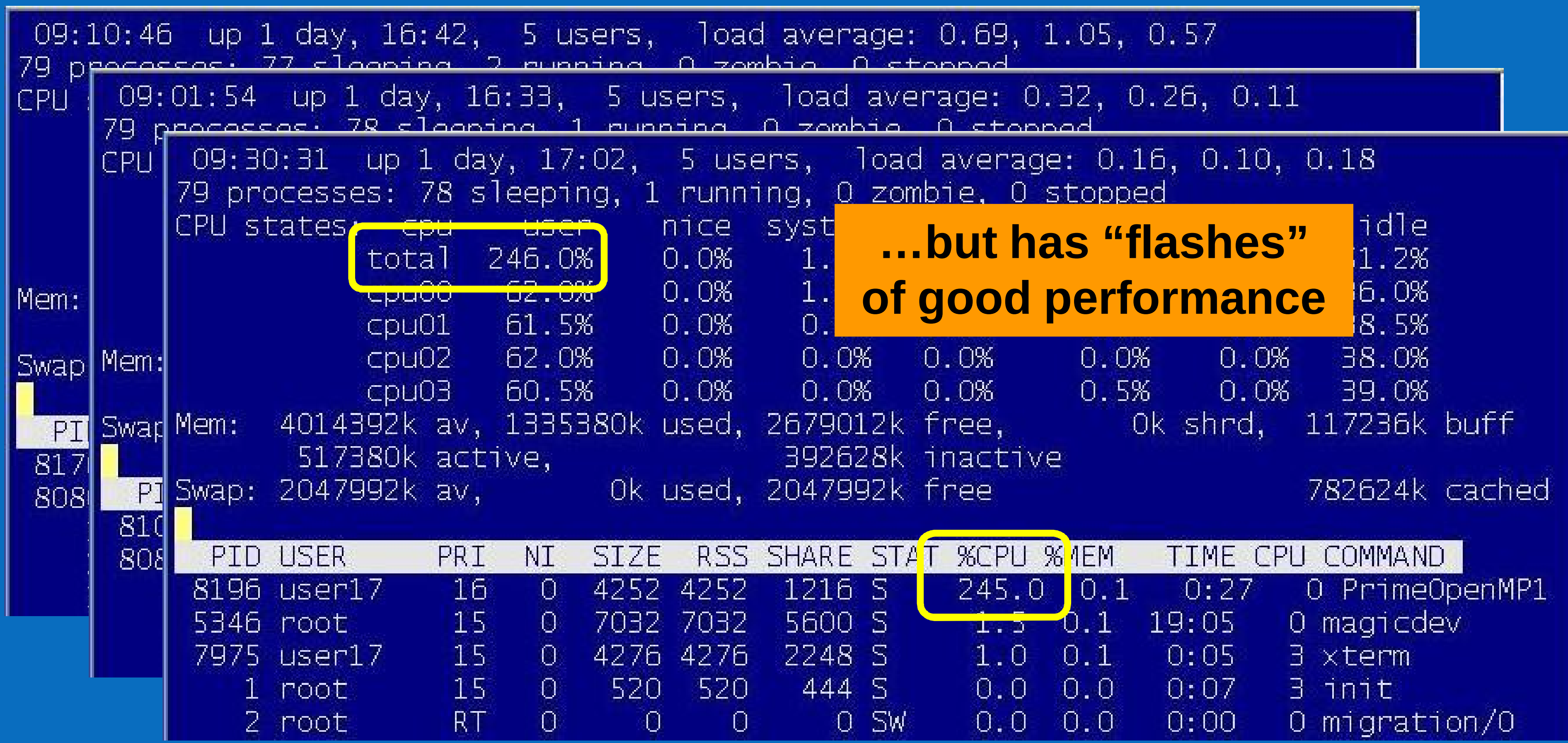
If no sync, would only achieve 1.17X

Why are we not getting any speedup?

OPECG 2009

Is it Possible to do Better?

Using `top` we see



Other Causes to Consider

Problems

Cache Thrashing

False Sharing

Excessive context switching

I/O

Tools

VTune Performance Analyzer

Thread Profiler (explicit threads)

Linux tools

- `vmstat` and `sar`
- Disk i/o, all CPUs, context switches, network traffic, interrupts

Linux sar Output

```
[user17@xeon-linux-production ~]$ sar -f app.sar -w 1 11 | more
Linux 2.6.9-11.ELsmp (xeon-linux-production)    01/16/2006

08:01:41 AM   cswch/s
08:01:42 AM    273.74
08:01:43 AM   1365.00
08:01:44 AM    744.00
08:01:46 AM    576.88
08:01:47 AM    782.00
08:01:48 AM    519.80
08:01:49 AM    521.00
08:01:50 AM    504.00
08:01:51 AM    633.00
08:01:52 AM    279.21
08:01:53 AM    280.81
Average:       588.24
[user17@xeon-linux-production ~]$ sar -f app.sar -W 1 11 | more
Linux 2.6.9-11.ELsmp (xeon-linux-production)    01/16/2006

08:01:41 AM  pswpin/s pswpout/s
08:01:42 AM      0.00      0.00
08:01:43 AM      0.00      0.00
08:01:44 AM      0.00      0.00
08:01:46 AM      0.00      0.00
08:01:47 AM      0.00      0.00
08:01:48 AM      0.00      0.00
08:01:49 AM      0.00      0.00
08:01:50 AM      0.00      0.00
08:01:51 AM      0.00      0.00
08:01:52 AM      0.00      0.00
08:01:53 AM      0.00      0.00
Average:         0.00      0.00
[user17@xeon-linux-production ~]$
```

Acceptable context switching

Less than 5000 context switches per second should be acceptable

Little paging activity

Memory doesn't seem a problem

Performance

This implementation has implicit synchronization calls

- Thread-safe libraries may have “hidden” synchronization to protect shared resources
- Will serialize access to resources

I/O libraries share file pointers

- Limits number of operations in parallel
- Physical limits

Performance

How many times is `printf()` executed?

```
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++;
    }
}
```

This change should fix the contention issue

Design

Eliminate the contention due to implicit synchronization

```
user17@xeon-linux-production:~/PrimeOpenMP
[user17@xeon-linux-production PrimeOpenMP]$ ./PrimeOpenMP 1 1000000
100%

   78498 primes found between       1 and 1000000 in     0.53 secs
[user17@xeon-linux-production PrimeOpenMP]$
```

Speedup is 17.87X!

Can that be right?

Performance

Our original baseline measurement had the "flawed" progress update algorithm

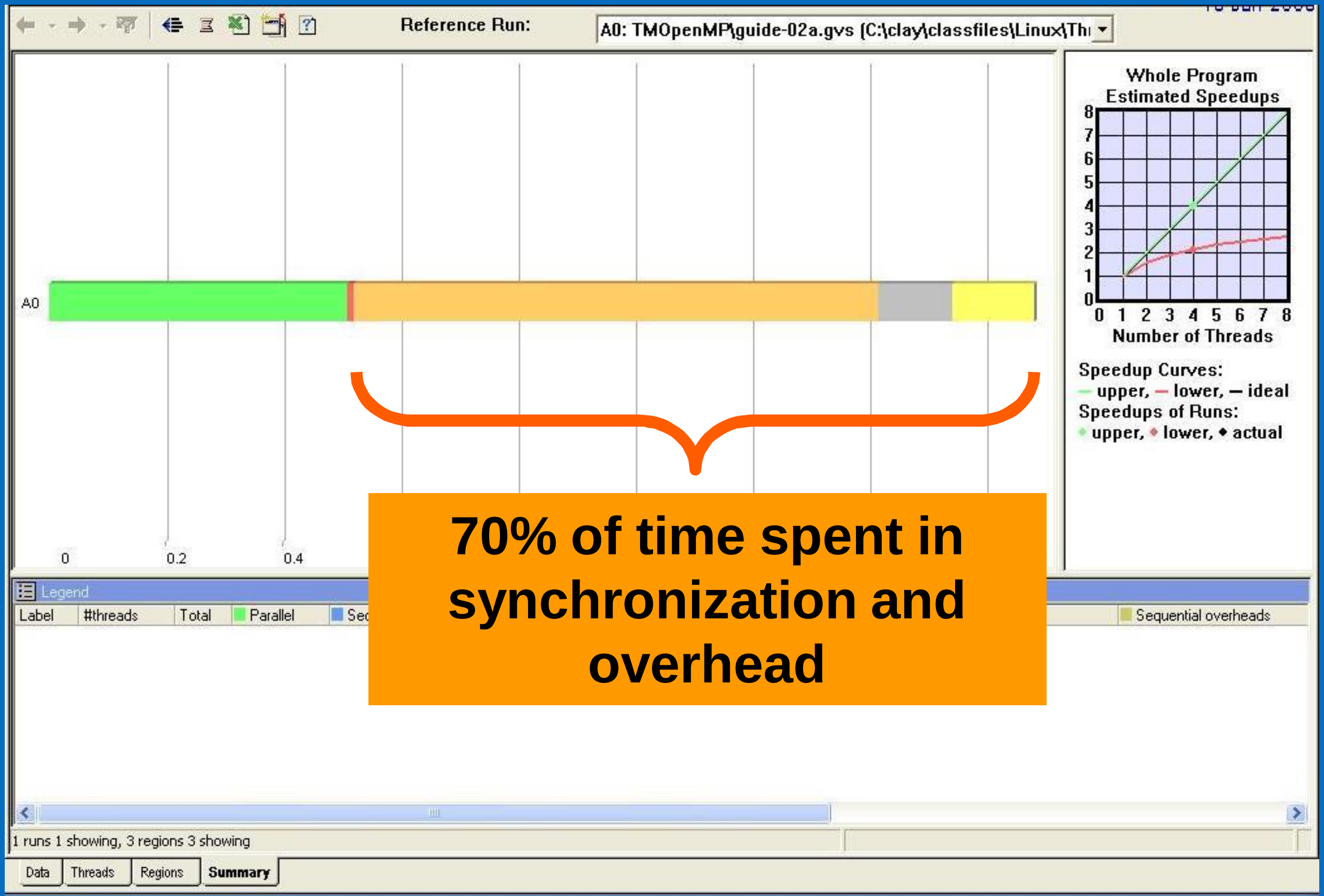
```
user17@xeon-linux-production:~/PrimeSingle
[user17@xeon-linux-production PrimeSingle]$ ./PrimeSingle 1 1000000
100%

   78498 primes found between      1 and 1000000 in    0.87 secs
[user17@xeon-linux-production PrimeSingle]$
```

Is this the best we can expect from this algorithm?

Speedup achieved is 1.49X (<1.9X)

Performance Re-visited

Let's use Thread Profiler again…



OpenMP Critical Regions

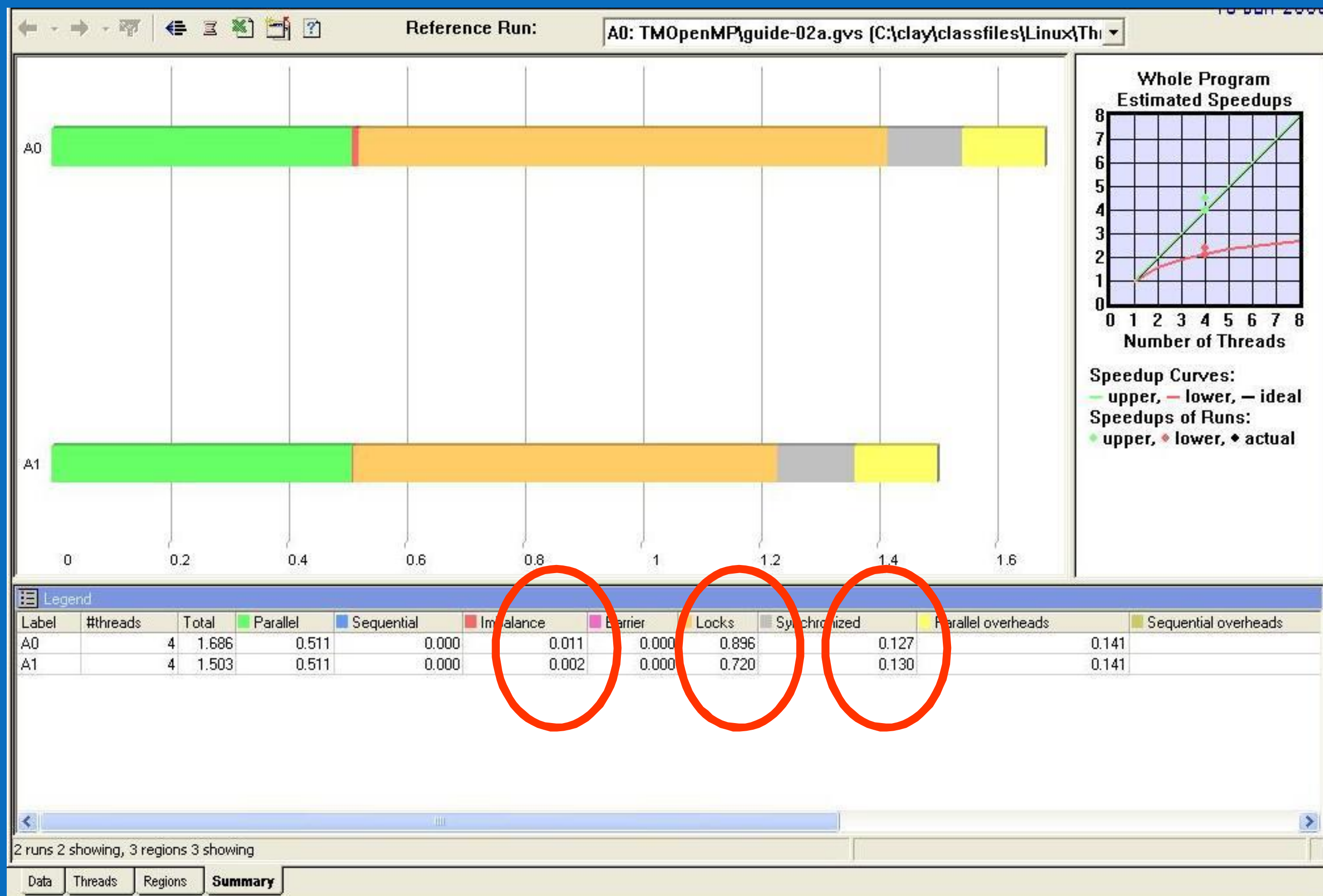
```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
       if( TestForPrime(i) )
  #pragma omp critical (one)
           globalPrimes[gPrimesFound++] = i;
       ShowProgress(i, range);
   }
```

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

Naming regions will create a different critical section for code regions

Performance Re-visited

Any better?



Reducing Critical Region Size

```
#pragma omp
    for( in
        if(
 #prag
    }
```

Two operations (read & write)

Only `gPrimesFound` update needs protection

Use local variables for read access outside critical region

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

Write and read are separate

Reducing Critical Region Size

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
       if( TestForPrime(i) ) {
  #pragma omp critical (one)
             lPrimesFound = gPrimesFound++;
```

```
user17@xeon-linux-production:~/PrimeOpenMP
[user17@xeon-linux-production PrimeOpenMP]$ ./PrimeOpenMP 1 1000000
100%

   78498 primes found between      1 and 1000000 in    0.46 secs
[user17@xeon-linux-production PrimeOpenMP]$
```

```
#pragma omp critical (two)
   lProgress = gProgress++;
 percentD
```

Speedup is up to 1.89X

Scaling Performance

user17@xeon-linux-production:~/finalruns

```
[user17@xeon-linux-production finalruns]$ ./PrimeSingle 1 10000000
100%

  664579 primes found between       1 and 10000000 in  22.04 secs
[user17@xeon-linux-production finalruns]$ ./PrimeOpenMP 1 10000000
100%

  664579 primes found between       1 and 10000000 in  11.23 secs
[user17@xeon-linux-production finalruns]$
```

1.96X

user17@xeon-linux-production:~/finalruns

```
[user17@xeon-linux-production finalruns]$ ./PrimeSingle 1 100000000
100%

 5761455 primes found between       1 and 100000000 in  581.71 secs
[user17@xeon-linux-production fin                  eOpenMP 1 100000000
100%

 5761455 primes found between       1 and 100000000 in  294.02 secs
[user17@xeon-linux-production finalruns]$
```

1.97X

Summary

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

Use tools to improve productivity

Unleash the power of dual-core and multi-core processors with multithreaded applications intel® Leap ahead™

Threading for Performance

- Other Threading Issues

Synchronization is an expensive, but necessary "evil" – Ways to minimize impact

- **Heap contention**
  - Allocation from heap causes implicit synchronization
  - Use local variable for partial results, update global after local computations
  - Allocate space on thread stack (`alloca`)
  - Use thread-local storage API (TlsAlloc)

- **Atomic updates versus Critical Sections**
  - Some global data updates can use atomic operations (Interlocked family)
  - Use atomic updates whenever possible

- **Critical Sections versus Mutual Exclusion**
  - Critical Section objects reside in user space
  - Introduces lesser overhead

False Sharing is a common problem when using data parallel threading

- False sharing can occur when 2 threads access distinct or independent data that fall into the same cache line

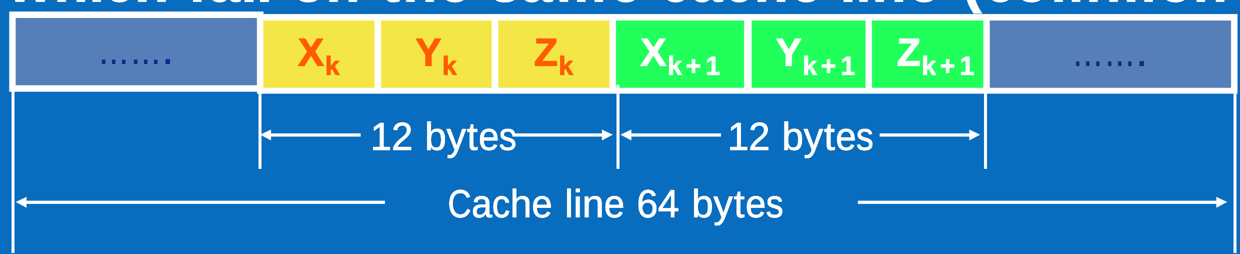
- Care should be taken to duplicate private buffers and counter variables be thread local

- Split dataset in such a manner as to avoid cache conflicts

False sharing Example

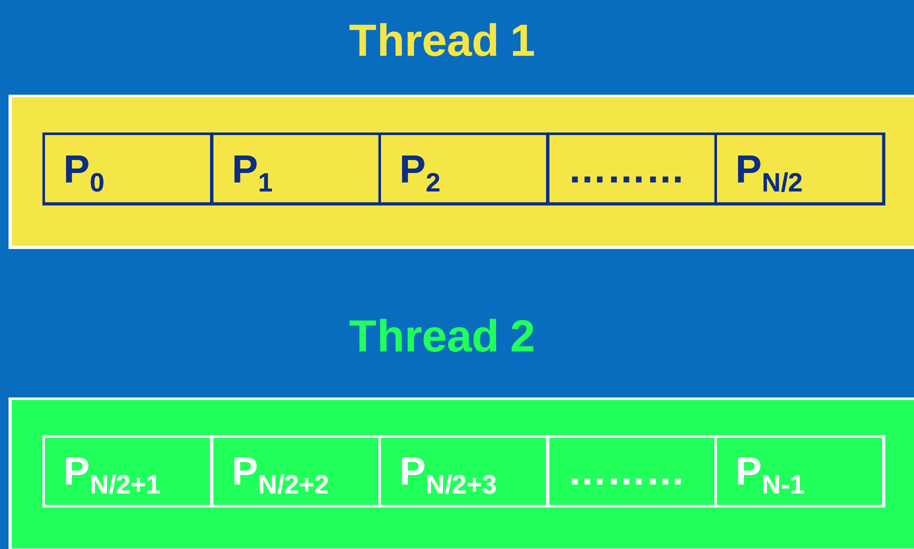
- Two threads divide the work by every other 3-component vertex

- Two threads update the contiguous vertices, v[k] and v[k+1], which fall on the same cache line (common case)



False sharing Example: Problem Fixed

- Let each thread handle half of the vertices by dividing the data into equal halves



Granularity of data decomposition is important to dynamically scale for the system

- **Finding the right sized "chunks" can be challenging**
  - Too large can lead to load imbalance
  - Too small can lead to synchronization overhead

- **What is the optimal # of concurrent threads? Depends on**
  - Total size of working set
  - Thread synchronization overhead

- **Adjust dynamically to help keep the balance right and reduce synchronization**

```
GetSystemInfo(&lpSystemInfo);

N = lpSystemInfo->dwNumberOfProcessors;
```

Threading with OpenMP

- Simple, portable, scalable SMP Specification
  - Compiler directives
  - Library routines

- Quick and Easy
  - *#pragma omp parallel for*
  - Thread scheduling, synchronization, sections, and more

- OpenMP is supported in Visual Studio® 2005

- Fork / Join programming Model
  - Pool of Sleeping threads
  - Single threaded until a Fork is reached

Intel® Threading Building Blocks
A Parallel Programming Model for C++

Generic Parallel Algorithms

ParallelFor
ParallelWhile
ParallelReduce
Pipeline
ParallelSort
ParallelScan

Concurrent Containers

ConcurrentHashTable
ConcurrentQueue
ConcurrentVector

TaskScheduler

Low-Level Synchronization Primitives

SpinMutex
QueuingMutex
ReaderWriterMutex
Mutex

Native Win32 Threads is the most commonly used currently

- _beginthread() / _endthread()
  - Basic thread creation function
  - Lacks configurability with potential for errors

- CreateThread() / ExitThread()
  - Provides more flexibility during thread creation
  - No automatic TLS (Thread Local Storage) created

- _beginthreadex() / _endthreadex()
  - Best native thread creation function to use
  - Automatically creates TLS to correctly execute multi-threaded C Runtime libraries

Atomic Updates

Use Win32 `Interlocked*` intrinsics in place of synchronization object

```
static long counter;

// Fast
InterlockedIncrement (&counter);

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

Lock free algorithms

See Game Programming Gems 6 and other recent literature

Compare and Swap:

- implemented by InterlockedCompareAndExchange
- Atomically implemented in hardware in x86 CPUs

Parallel Overhead

Thread Creation overhead

- Overhead increases rapidly as the number of active threads increases

Solution

- Use of re-usable threads and thread pools
  - Amortizes the cost of thread creation
  - Keeps number of active threads relatively constant

Synchronization

Heap contention

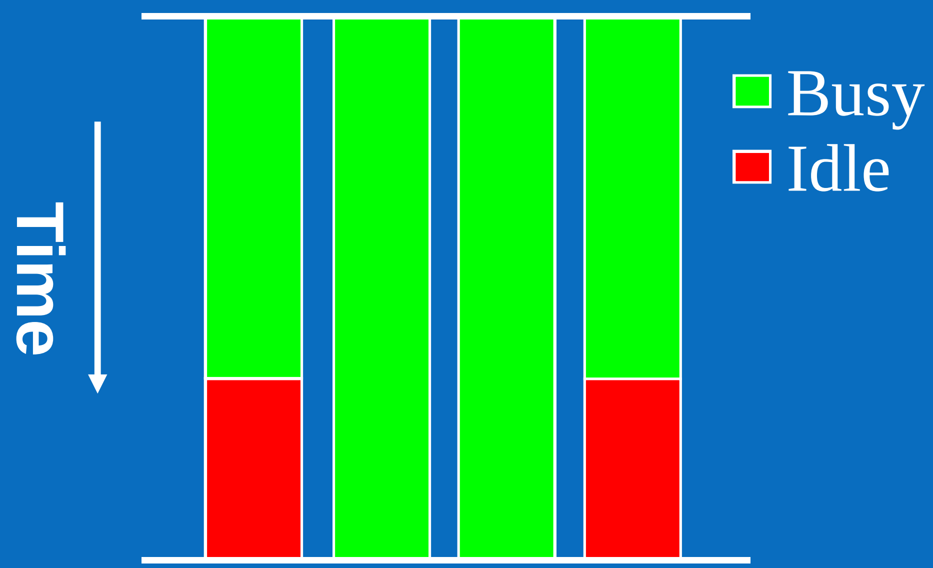
- Allocation from heap causes implicit synchronization
- Allocate on stack or use thread local storage

Atomic updates versus critical sections

- Some global data updates can use atomic operations (Interlocked family)
- Use atomic updates whenever possible

Load Imbalance

Unequal work loads lead to idle threads and wasted time



Granularity

Loosely defined as the ratio of computation to synchronization

Be sure there is enough work to merit parallel computation

Example: Two farmers divide a field. How many more farmers can be added?