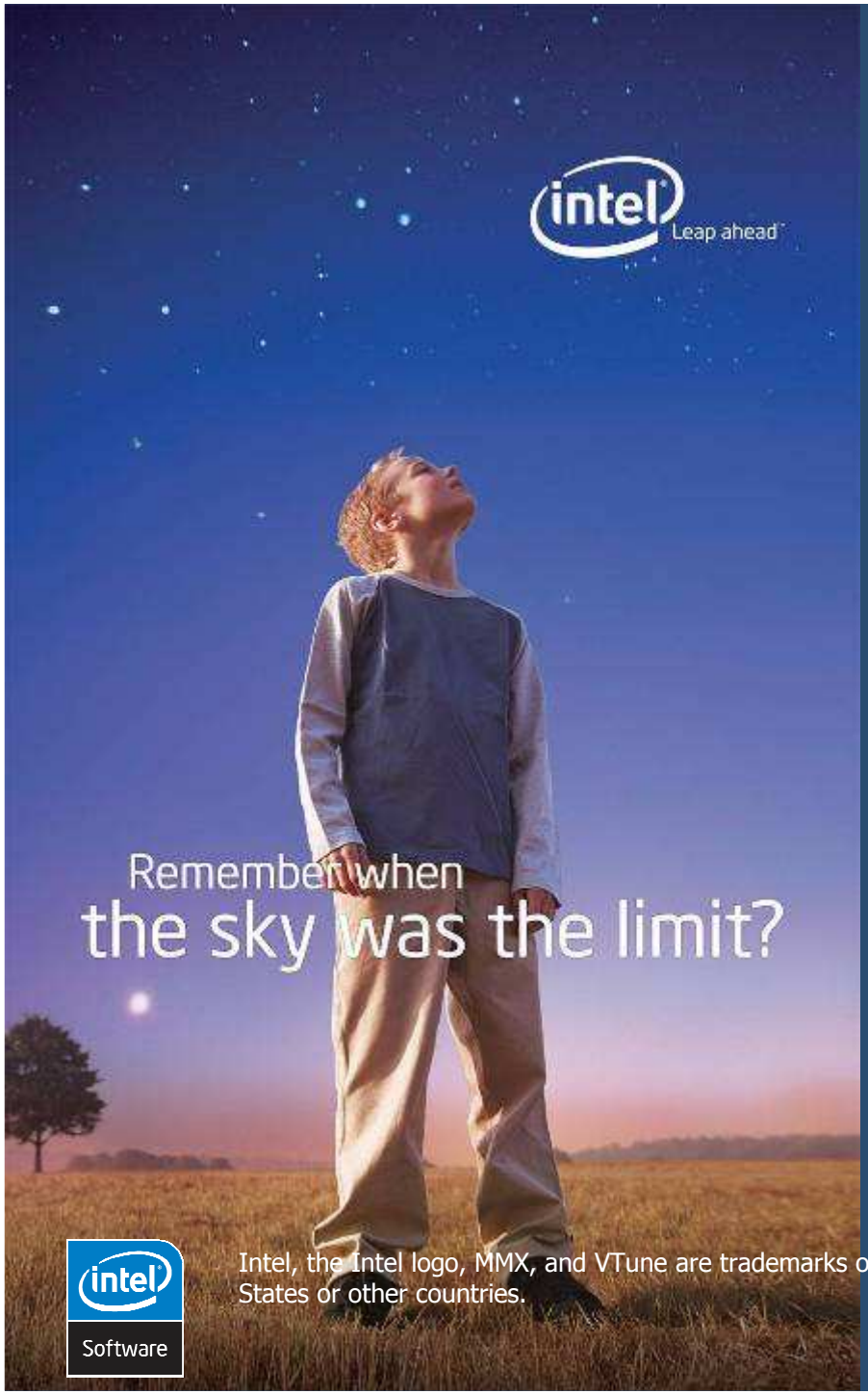


Best Practices for Developing Multi-threaded Programs

Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

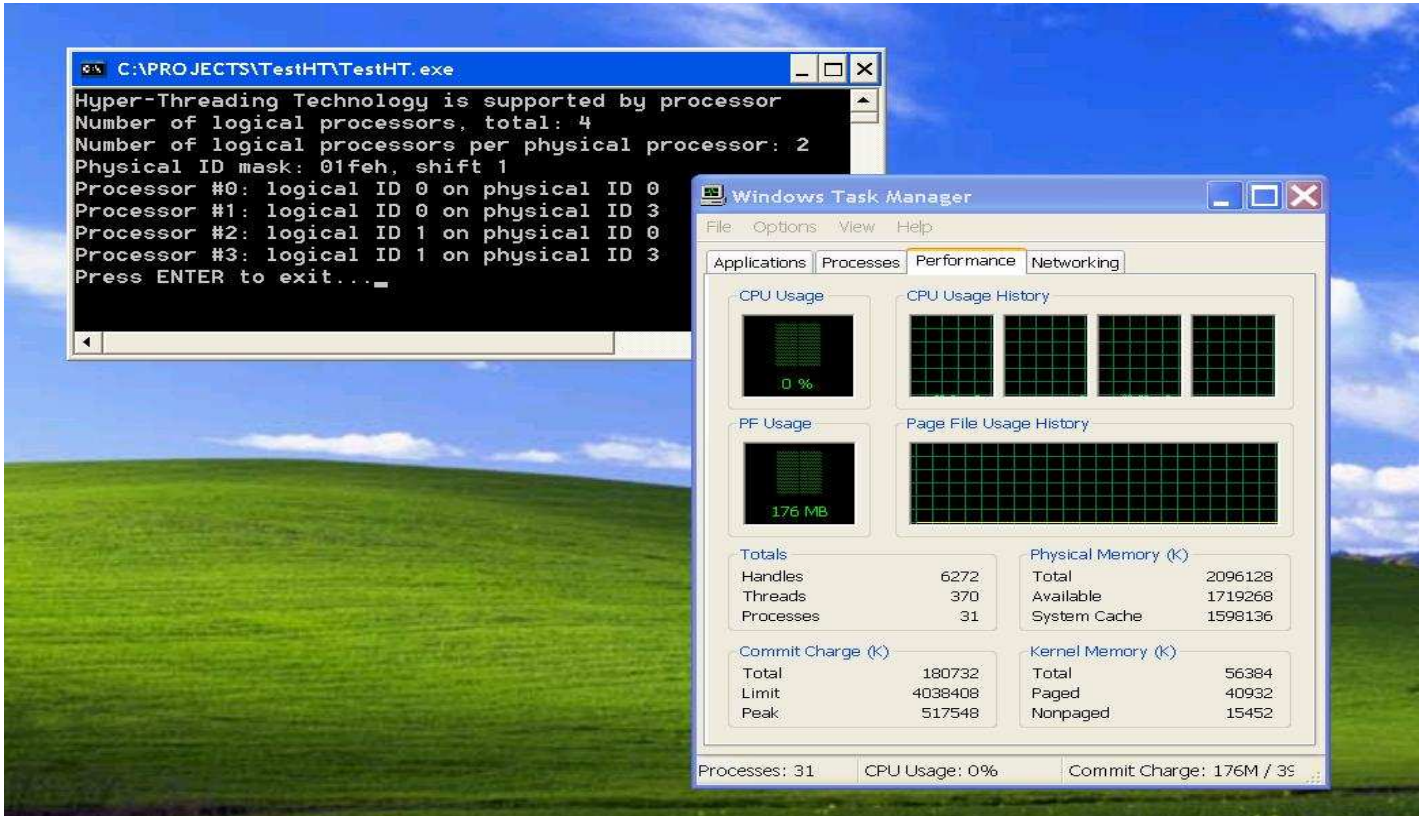
Common Threading Challenges

Intel® Software Tools for Multi-threading

Summary

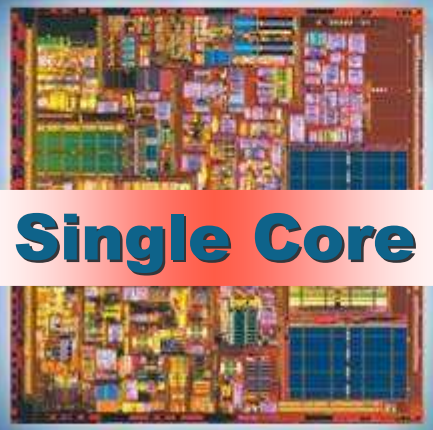
Hardware Architecture

The trend toward multi-core mobile, desktop, and server processors is expected to continue into the foreseeable future, and software must be threaded to take full advantage of it.



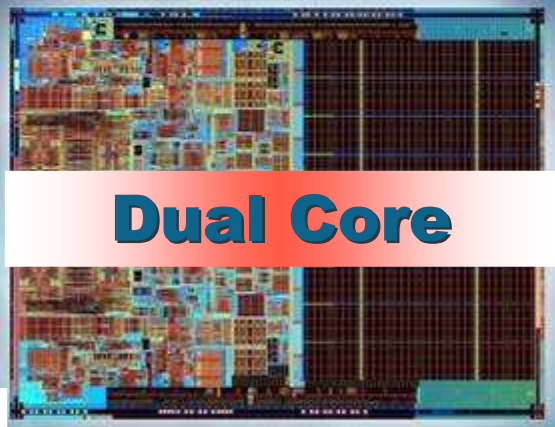
Processor Evolution

Intel® Xeon® processor



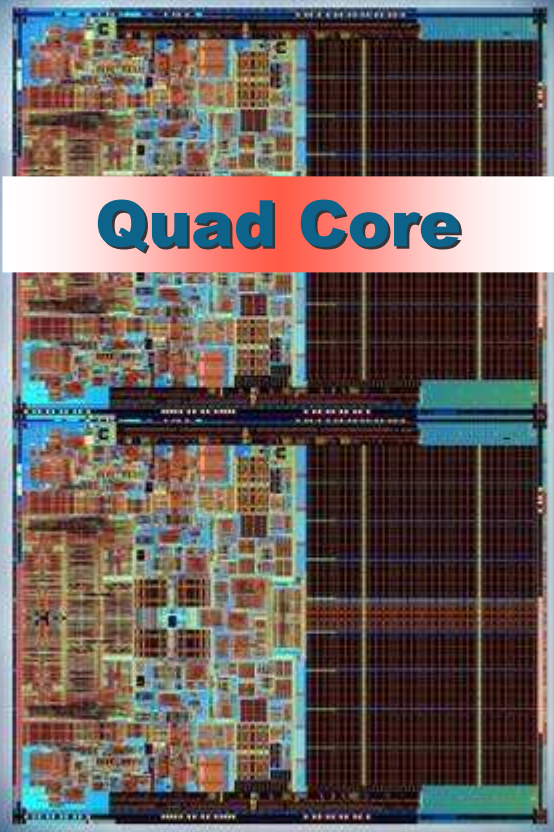
Single Core

Dual-Core Intel® Xeon® processor 5100 series



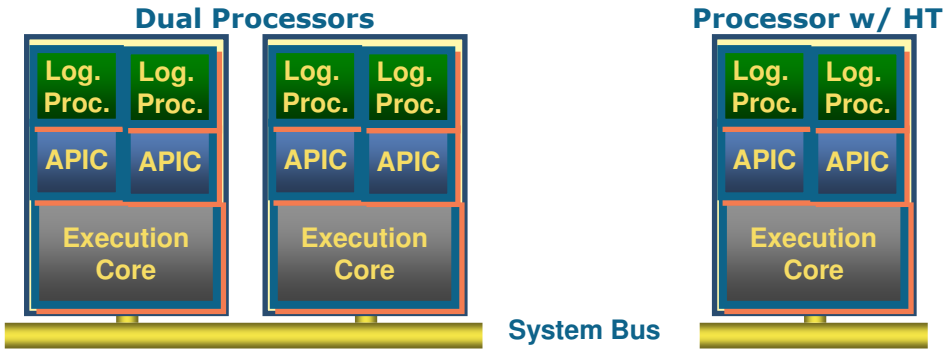
Dual Core

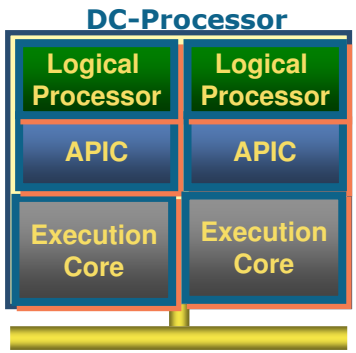
New Quad-Core Intel® Xeon® 5300 for 2006

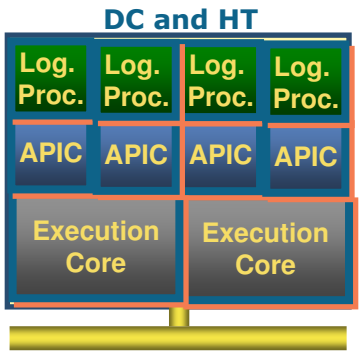


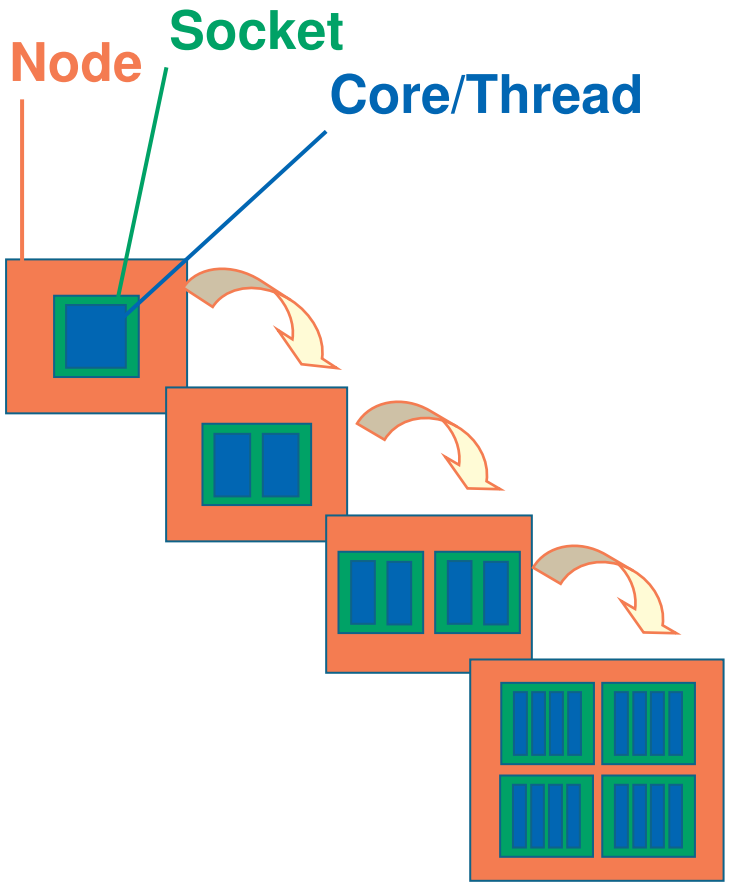
Quad Core

HT Technology, Dual-Core and Multi-Core

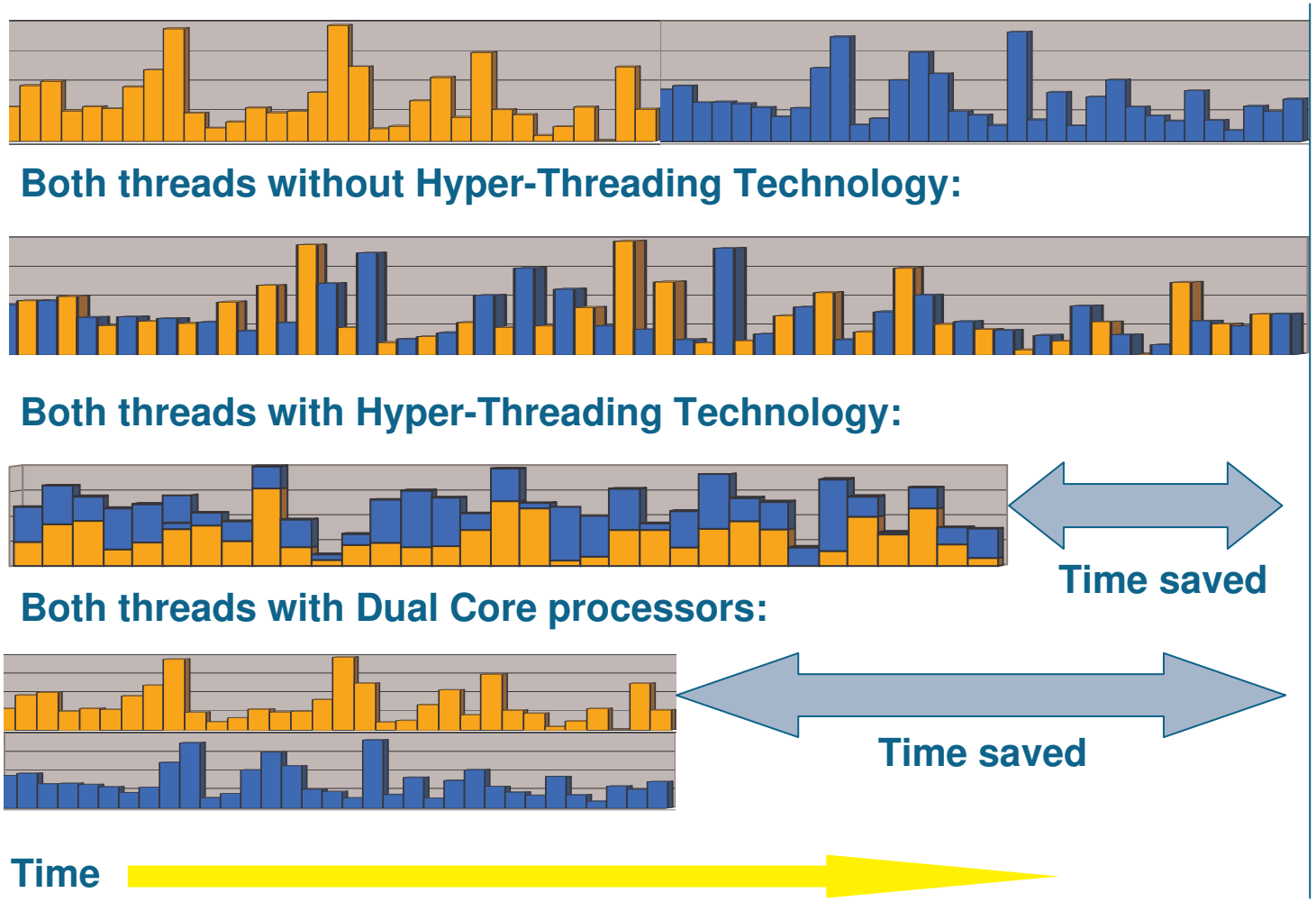




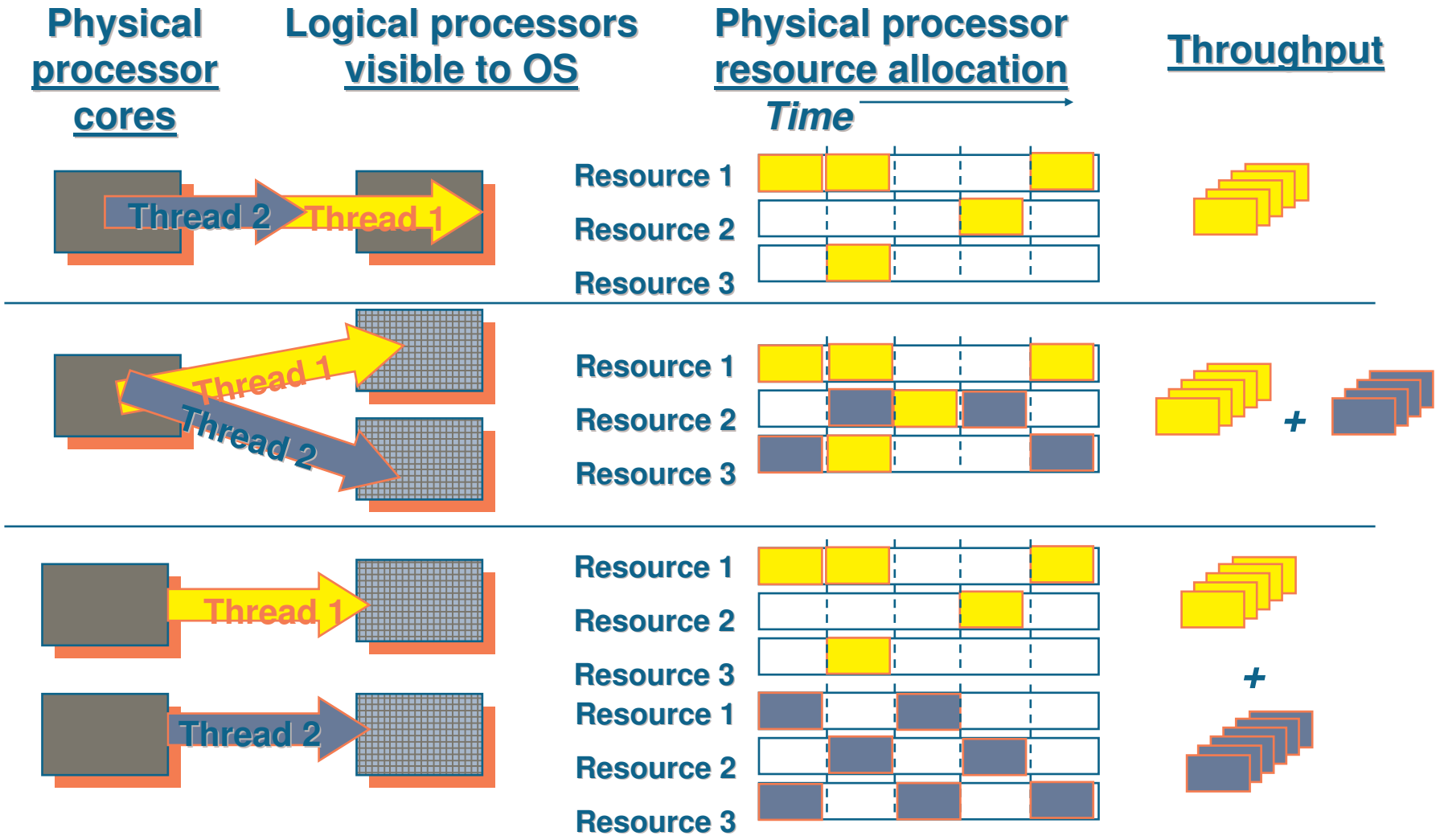




Hyper-Threading vs. Dual Core



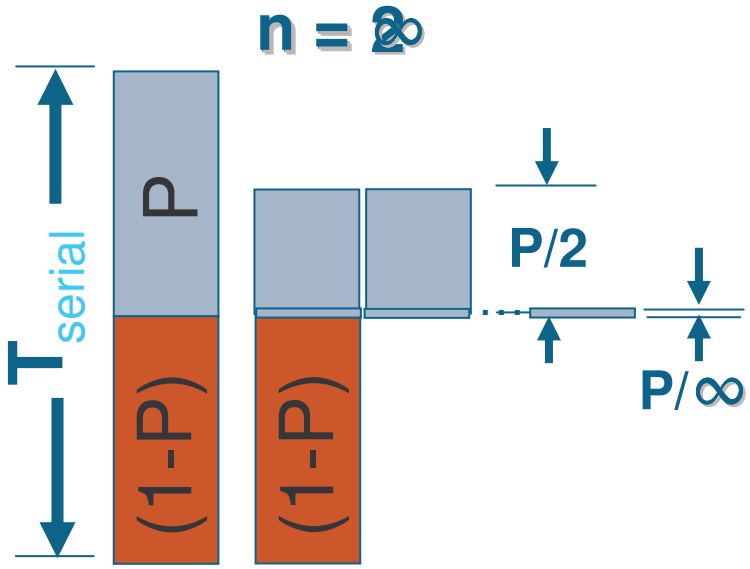
How HT Technology and Dual-Core Work



Why use parallelism? Amdahl's Law

- Describes the upper bound of parallel execution speedup

Serial code limits speedup



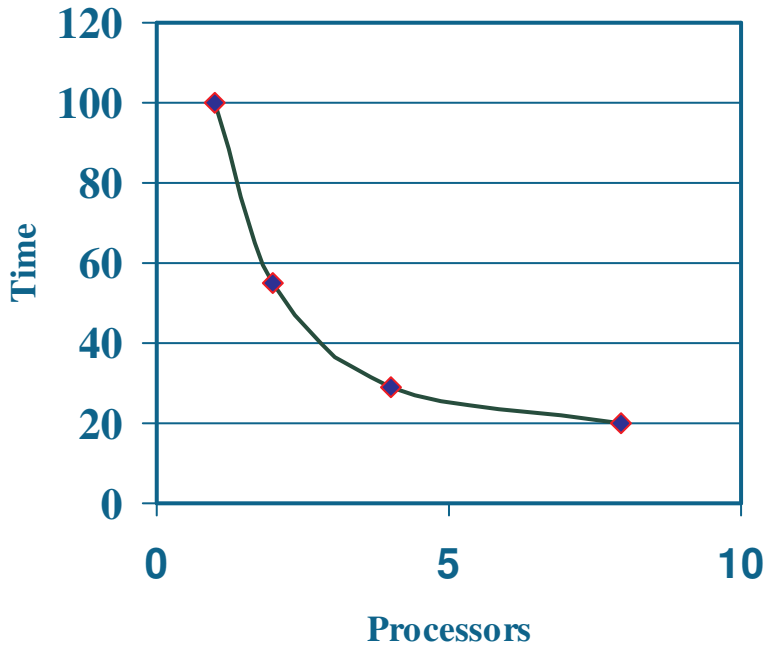
$$T_{parallel} = \{(1-P) + P/n\} T_{serial}$$

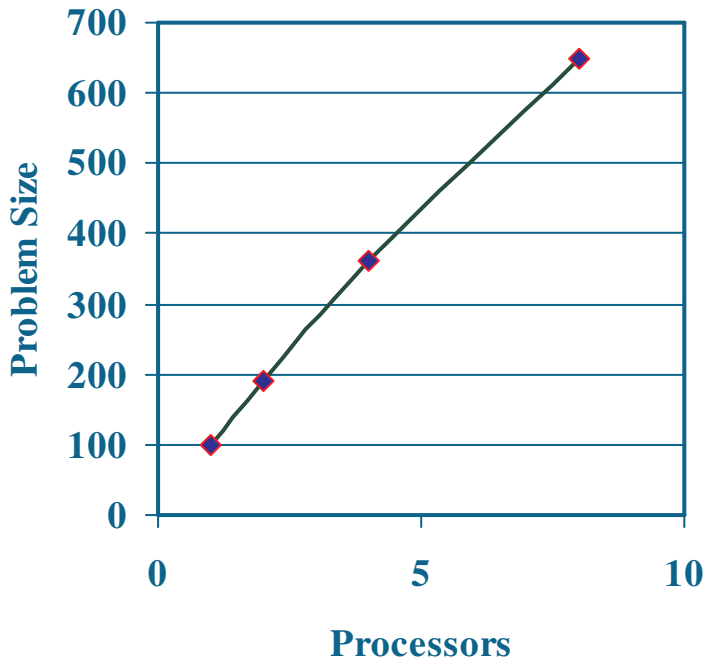
n = number of processors

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

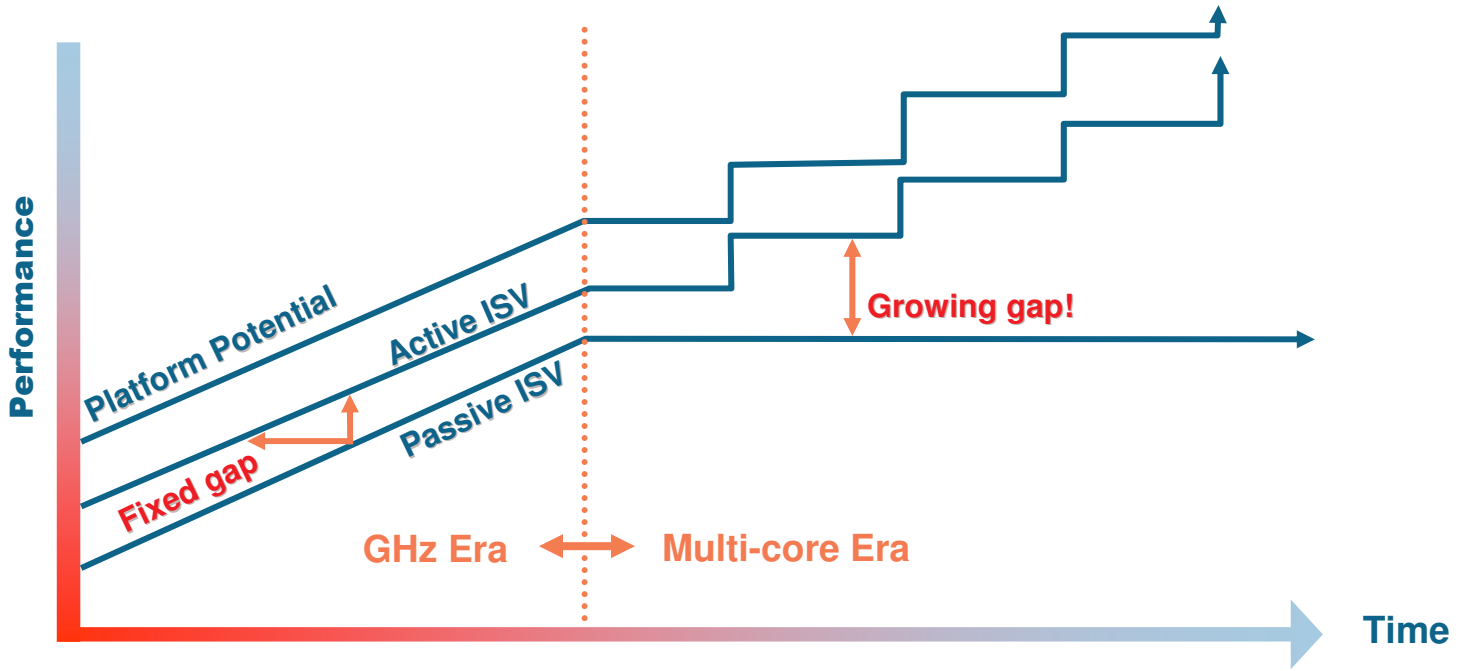
Why use parallelism ?

- Speed up a given problem
- Scale up to a larger problem with fixed run-time





Multi-core Processors change the rules



Get your software ready for multi-core using Intel® Tools

Why Thread Your Application?

Increased responsiveness and worker productivity

- Increased application responsiveness when different tasks run in parallel

Improved performance in parallel environments

- When running computations on multiple processors

More computation per cubic foot of data center

- Web-based apps are often multi-threaded by nature

Taking full advantage of Multi-Core requires multi-threaded software

Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

Common Threading Challenges

Intel® Software Tools for Multi-threading

Summary

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) – Message Passing
- Single process, multiple threads
  - Communication through shared memory

Different levels of parallelism

- Hardware parallelism
  - Processor pipeline, execution units, SSEx
  - Two logical processors sharing the resources of one physical processor
  - SMP systems, shared memory
- Software parallelism
  - Data parallelism
  - Task parallelism

Intel® Architecture – Hardware level

Instruction Level Parallelism (1993)

- Out-of-order instruction pipeline
- Multiple execution units

Data Level Parallelism (1997)

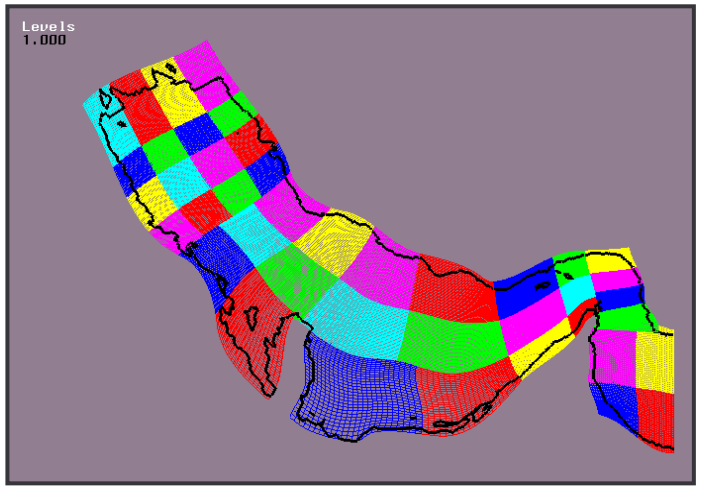
- MMX™
- Streaming SIMD Extensions (SSE)
- Streaming SIMD Extensions 2 (SSE2)
- Streaming SIMD Extensions 3 (SSE3)

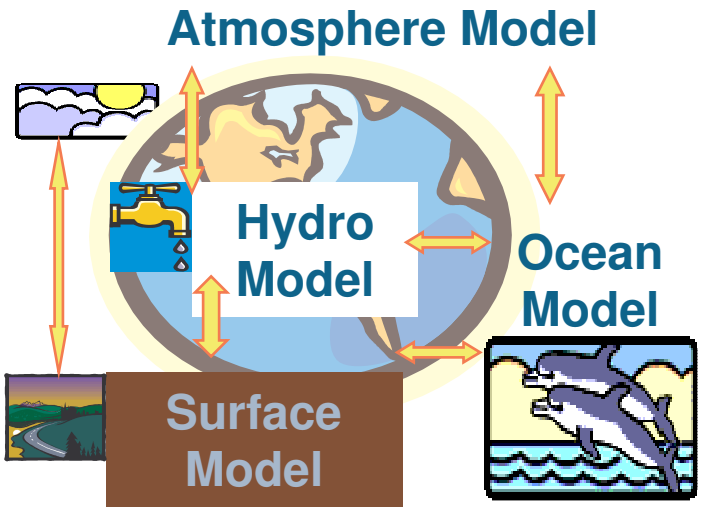
Thread Level Parallelism (2002)

- Hyper-Threading Technology (HT Technology)
- Multi-core Intel processor architecture

Partitioning Methods

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





Domain Decomposition

- **Data Parallelism:**

  Same operation applied to all data

Functional Decomposition

- **Task Parallelism**

  Each Thread performs a unique job

GOAL: Identify independent computations / primitive tasks

Most Code Contains Parallelism

Task parallelism:

Independent subprograms

```
call fluxx(fv,fx)
call fluxy(fv,fy)
call fluxz(fv,fz)
```

Data parallelism:

Independent loop iterations

```
for (y=0; y<nLines; y++)
   genLine(model,im[y]);
```

Options for Adding Parallelism

Explicitly thread your program using Win32*/POSIX* threading APIs

Use a Compiler to automatically parallelize code

Use a Programming Language API (C#*, Java*, etc.)

Programming Language Extension (OpenMP*)

- Use OpenMP directives to tell the compiler how to decompose parts of a serial program for parallel execution

Use an internally-threaded runtime library for common tasks

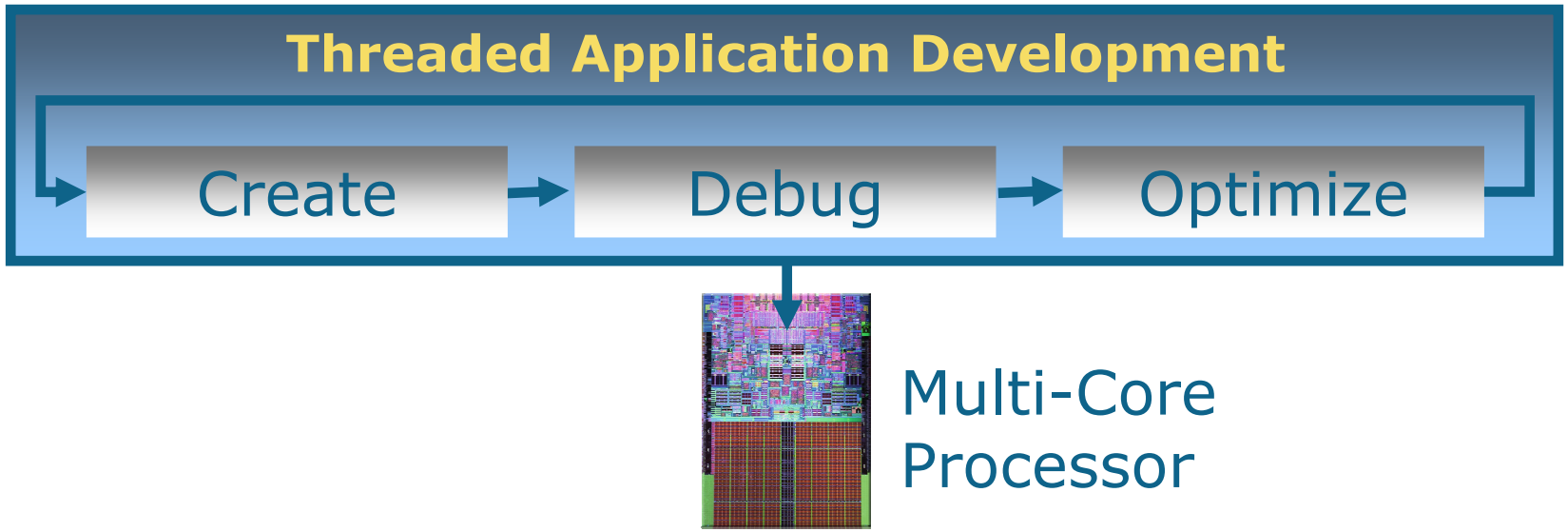
- Intel® Threading Building Blocks (TBB), Intel® Integrated Performance Primitives (Intel® IPP) and Intel® Math Kernel Library (Intel® MKL)
- Parallel memory managers like MicroQuill SmartHeap* and Hoard*

Maximize Multi-Core Performance By Threading Your Software

Software threading needed at the application level

- Breaks problem into pieces that can be solved in parallel
- Performance can scale with number of processors

Need tools to create, debug and optimize multi-threaded applications



Developer + Tools

Use your knowledge of the algorithm to identify opportunities for parallelism

Verify with tools

- Tools will identify dependencies you overlooked
- Tools will help identify regions for greatest ROI
- Tools will improve your productivity

> "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason …"
>
> — **William M. Wulf**

Developer knowledge of algorithm is important!

Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

Common Threading Challenges

Intel® Software Tools for Multi-threading

Summary

Start Small

If full-blown threading looks too hard…

Look at the code where you spend the most time

Identify code regions that would benefit from parallelism

Try to use the Intel® Compiler to parallelize tight inner loops

- /Qparallel* and /Qx* options
- OpenMP* directives and /Qopenmp* option

Use an OpenMP*, Intel® Thread Checker, and Intel® Thread Profiler to prototype possible threading implementations

- Once you have a good algorithm, you can rewrite in a native threading API like Win32* or pThreads* as desired.

Start Small (continued)

If full-blown threading looks too hard (continued)

Replace calls to large common functions with calls to internally parallel libraries such as:

- Intel® Threading Building Blocks
- Intel® Integrated Performance Primitives
- Intel® Math Kernel Library

Make your libraries thread-safe in anticipation of their being called by threaded code

- Use a simple multi-threaded test harness

Consider "functional" parallelism

- Try to separate computation from unrelated tasks (such as the GUI, printing, etc.)

Threading Methods

For performance hungry apps – parallel threads use all cores

Use OpenMP* compiler for automated thread creation

	OpenMP	Native Threads	Intel® TBB
Portable	✓	somewhat	Intel only
Performance Oriented	✓		✓
High-Level	✓		✓
No Special Compiler		✓	✓
Serial Code Intact	✓		
Works well with C++	if C-like	✓	✓
Data Parallel	✓		✓
Functional Decomposition		✓	✓
Nested Parallelism Support	some		✓

Example: Non-Threaded Application

```
public class Report
{
     public void Compute(int x)
     {
          x = x * x; ... // long task
     }
}

public static void main()
{
     Report r = new Report();
     r.Compute(5);
}
```

Example: Worker Thread

```
public class Report
{
      public void Compute(int x)
      {
          this.x = x;
          mi = new MethodInvoker(AsyncCompute);
          mi.BeginInvoke(null, null);
      }
      void AsyncCompute()
      {
          x = x * x; ... // long task
      }
      MethodInvoker mi;
      int x;
}

public static void main()
{
      Report r = new Report();
      r.Compute(5);
}
```

POSIX* pthreads*: Example

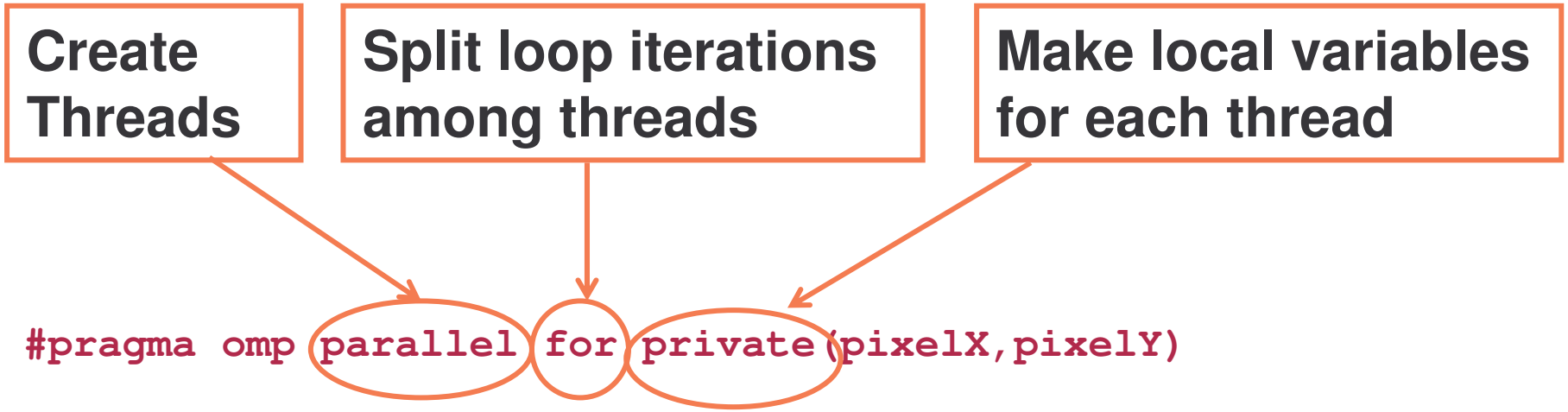
```
#include <stdio.h>
#include <pthread.h>
const int num_threads = 4 ;

void* thread_func(void* arg) {
  do_work();
  return NULL;
}
main() {
  pthread_t threads[num_threads];
  for( int i = 0; i < num_threads; i++)
       pthread_create(&threads[i], NULL, thread_func, NULL);

  for(int j = 0; j < num_threads; j++)
         pthread_join (threads[j], NULL);
}
```

Parallel processing

OpenMP* Threads: Implicit



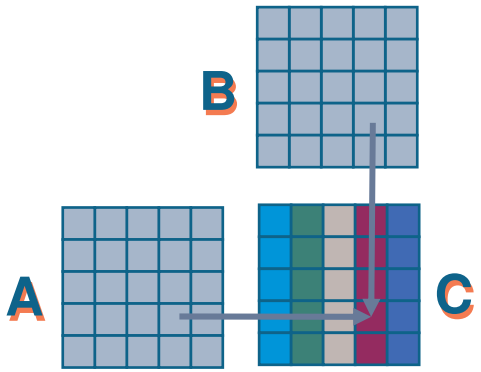

```
#pragma omp parallel for private(pixelX,pixelY)

for (pixelX = 0; pixelX < imageWidth; pixelX++)
{
   for (pixelY = 0; pixelY < imageHeight; pixelY++)
   {
      newImage[pixelX,pixelY] =
                  ProcessPixel(pixelX, pixelY, image);
   }
}
```

Example – OpenMP* Threads

Number of threads is determined at initialization time (number of processors).

If disabled, code looks and runs like single threaded code




```
…
// Divide work of outer loop between all
processors on the system
#pragma omp parallel for private(x,y)
for (x=0; x<width; x++)
        for (y=0; y<height; y++)
                C(x,y) = F(A(x,y),B(x,y));
…
```

Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

Common Threading Challenges

Intel® Software Tools for Multi-threading

Summary

Challenges to Implementing Parallelism

Correctness

- Shared resource identification
- Threading the right code correctly
- Difficult to debug
  - Data Races
  - Deadlocks

Performance

- Program decomposition (Functional task/Data)
  - Load Imbalance
  - Granularity
- Resource utilization (load balancing)
  - Task Priorities
  - Adequate Memory and I/O bandwidth
- Parallel Overhead (Thread management and synchronization)
  - Due to thread creation, scheduling …
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- All these affect scaling

Program Gives Incorrect Results

Some possible explanations:

- Race condition or storage conflicts
  - More than one thread accesses memory without synchronization
  - Locking used, but is too local to be effective
- Other components (such as 3rd-party APIs) may not be "thread safe" in certain use cases

Debug via:

- Intel® Thread Checker
- A tool like Rational Purify* or Compuware BoundsChecker*

Data Race Example

Suppose you have global variables a=1, b=2

Thread1	Thread2
x = a + b	b = 42

The end result can be different if

- Thread**1** runs before Thread**2**:
- Thread**2** runs before Thread**1**:

x = 3

x = 43

Execution order is not guaranteed unless updates to "b" and its use are protected by a synchronization mechanism.

Threaded Program Hangs

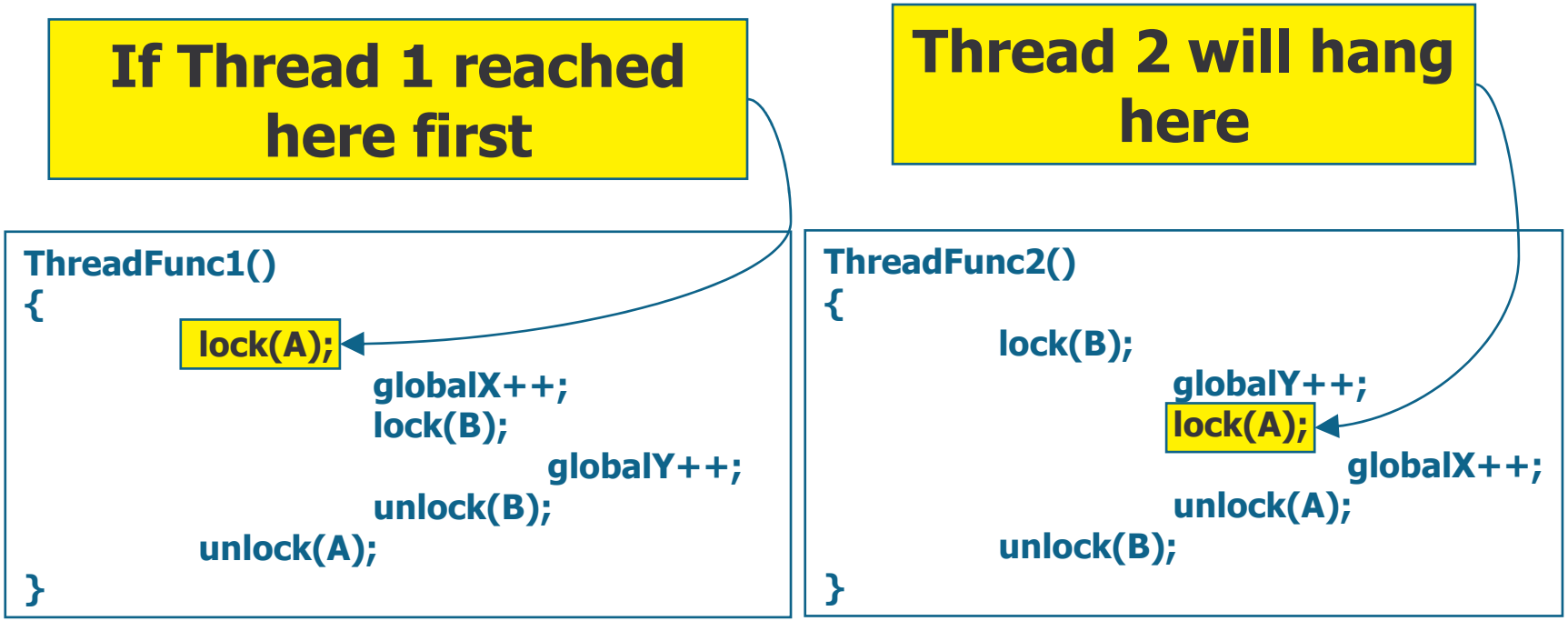
Possible explanations:

- Deadlocks
  - Actual (A waits on B, which is waiting on A)
  - Potential (will happen under the "right" conditions)
- Thread Stalls and Waits
  - Dangling locks (thread exits holding a lock requested by another thread)
    - Thread exit does not automatically release held locks
    - Must release lock from the same thread where you obtained (entered) it

Debug using:

- Intel® Thread Checker
- A standard debugger
- Print statements (be sure to identify the thread)

Deadlock Example



Deadlock - Both threads are now frozen

To fix: both functions must acquire and release locks in the same order

Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

Common Threading Errors

Intel® Software Tools for Multi-threading

Summary

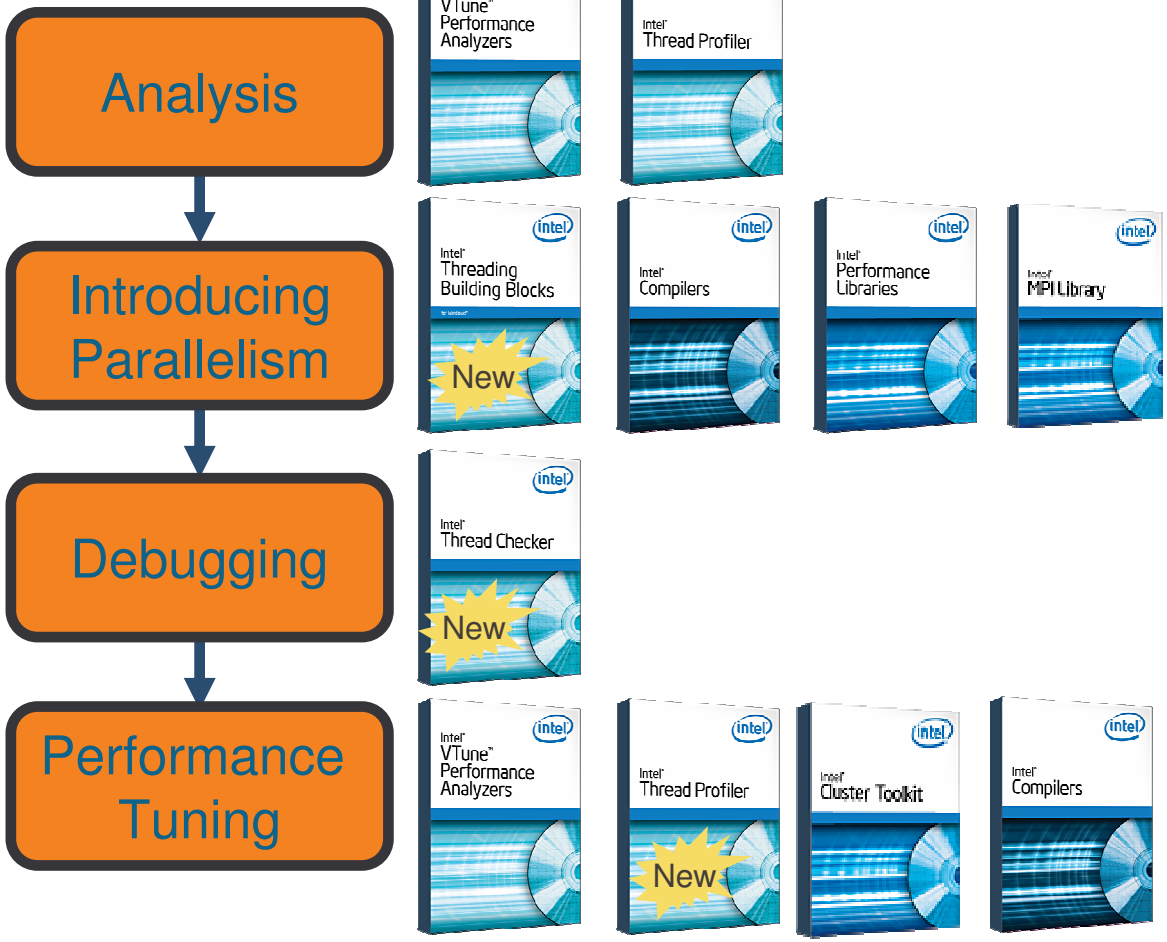
Comprehensive, industry leading solutions for parallelized software development

Analyze your application and identify multi-core performance bottlenecks and hotspots

Highly optimizing compilers delivering scalable solutions

Detect latent programming to address unique challenges

Tune for performance and scalability



Intel has a broad toolset to help develop fast, reliable threaded applications

intel

Intel® Software Development Products

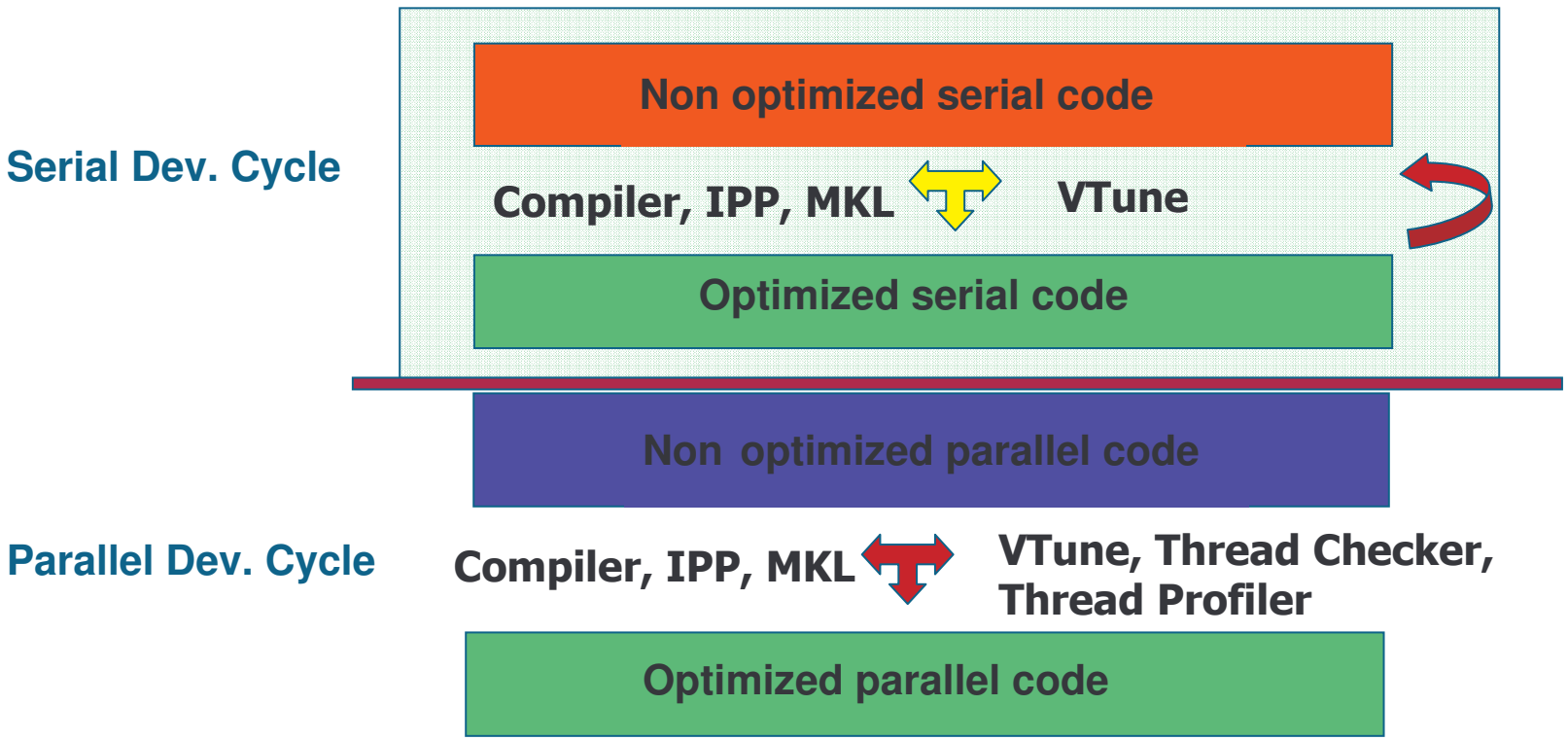
Intel® VTune™ Performance Analyzer

- Identify "hot spots" of code that may benefit from threading
- Shows callgraph to help identify threading candidates

Intel® Threading Tools

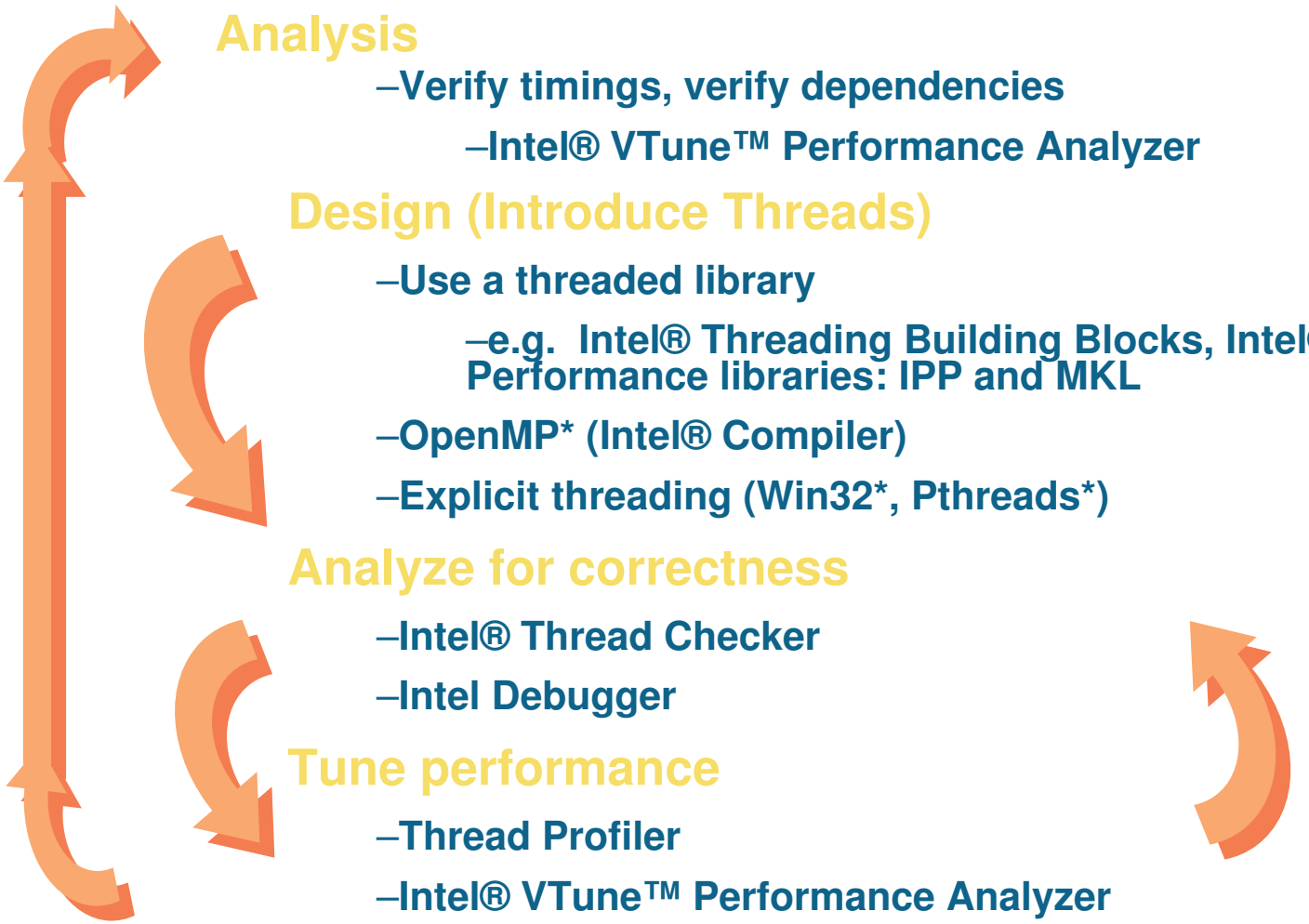
- Locate thread performance bottlenecks
- Estimate achievable/available performance
- Quickly validate designs and create prototypes
- Locate positions of data race conditions (read/write, write/read, write/write)
  - Isolate deadlock
  - Identify inappropriate API arguments

Development Cycle



Serial and parallel performance problems are different and need extra tools !

Development Cycle



Analysis

- –Verify timings, verify dependencies
  - –Intel® VTune™ Performance Analyzer

Design (Introduce Threads)

- –Use a threaded library
  - –e.g. Intel® Threading Building Blocks, Intel® Performance libraries: IPP and MKL
- –OpenMP* (Intel® Compiler)
- –Explicit threading (Win32*, Pthreads*)

Analyze for correctness

- –Intel® Thread Checker
- –Intel Debugger

Tune performance

- –Thread Profiler
- –Intel® VTune™ Performance Analyzer

Example – Prime Number Generation

```
main(int argc, char *argv[])
{
  int i, j, limit;
  clock_t time_1, time_2;
  int start = 1000000;
  int end = 4000000;        /* start:end == range of numbers to search */
  int number_of_primes=0;  /* number of primes found */
```

```
for (i = start; i <= end; i += 2)
  {
    limit = (int) sqrt((double)i) + 1;
    prime = 1; /* assume number is prime */

    for (j=3; prime && (j <= limit); j+=2)
      if (i % j == 0)
        prime = 0;
    if (prime)
                  number_of_primes++;
  }
```

```
        prime = 1; /* assume number is prime */

        for (j=3; prime && (j <= limit); j+=2)
```

2
3 3
5 3,5
7 3,5,7
9 3
11 3,5,7,9,11
13 3,5,7
15 3
17 3,5,7

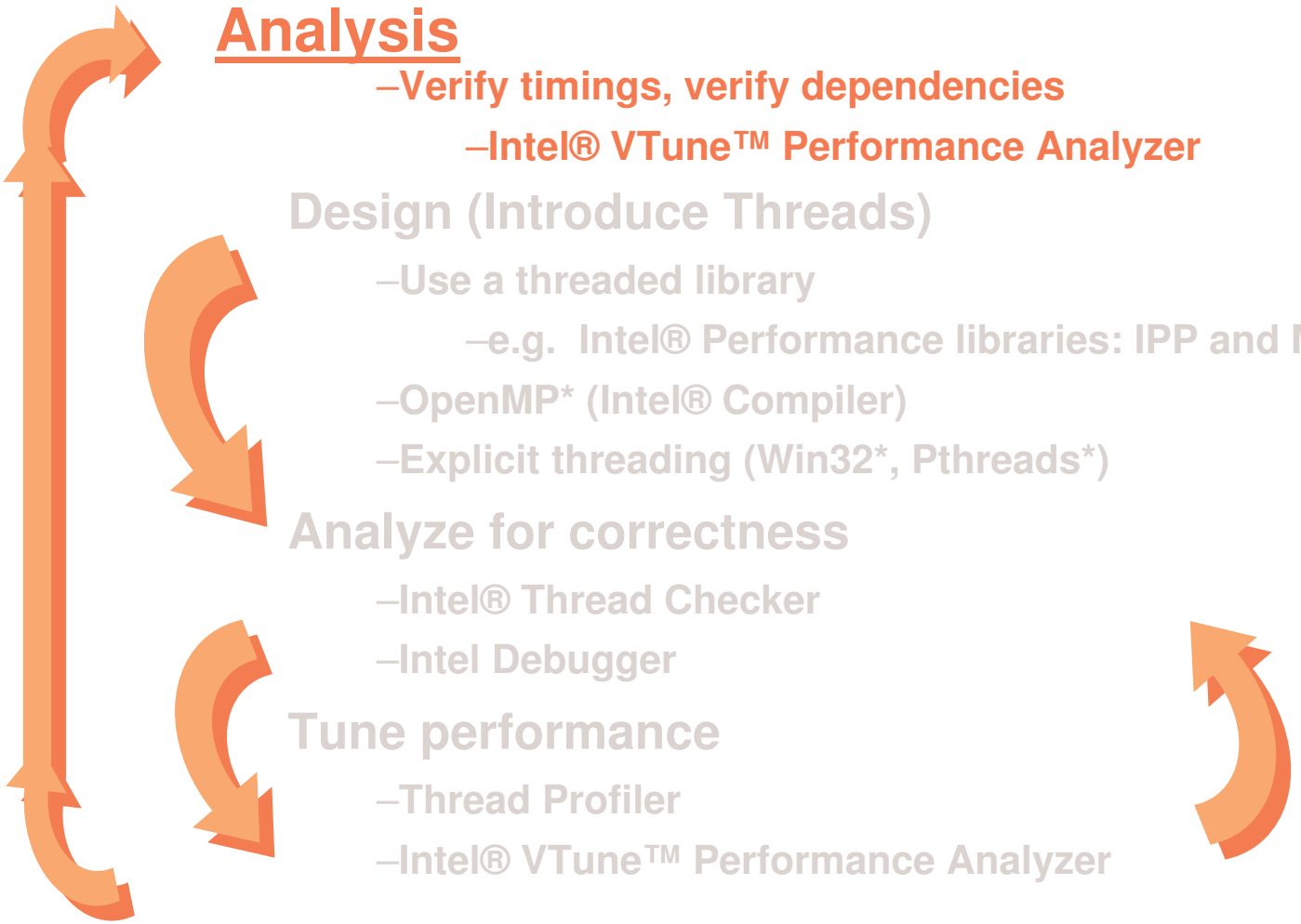

```
        number_of_primes,
          (double) (time_2-time_1)/CLOCKS_PER_SEC);
            (double) (time_2-time_1)/CLOCKS_PER_SEC);
}
```

Computing Primes

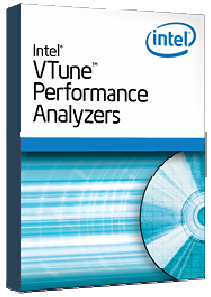
- Compute the number of primes between *start* and *end*
  - Not the fastest way, really a 'brute force approach'
- Numbers are checked independently of each others

```
for(i = start; i <= end; i += 2){
  limit = (int) sqrt((float)i) + 1;
  prime = 1; /* assume number is prime */
  for ( j=3;  (prime && (j <= limit); j+=2 )
    if (i%j == 0) prime = 0;
  if (prime)
    number_of_primes++;
}
```

Development Cycle



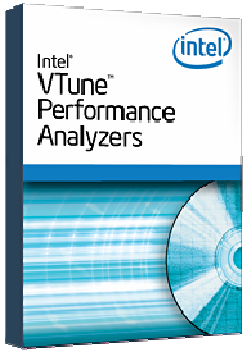
VTune™ Performance Analyzer



Helps you identify and characterize performance issues by:

- Collecting performance data from the system running your application
  - Sampling: Event-based or Time-based
  - Call Graph
  - Counter Monitor
- Organizing and displaying the data in a variety of interactive views
  - From system-wide down to source code or processor instruction perspective
  - GUI and CLI
- Identifying potential performance issues and suggesting improvements

Intel® VTune™ Performance Analyzer



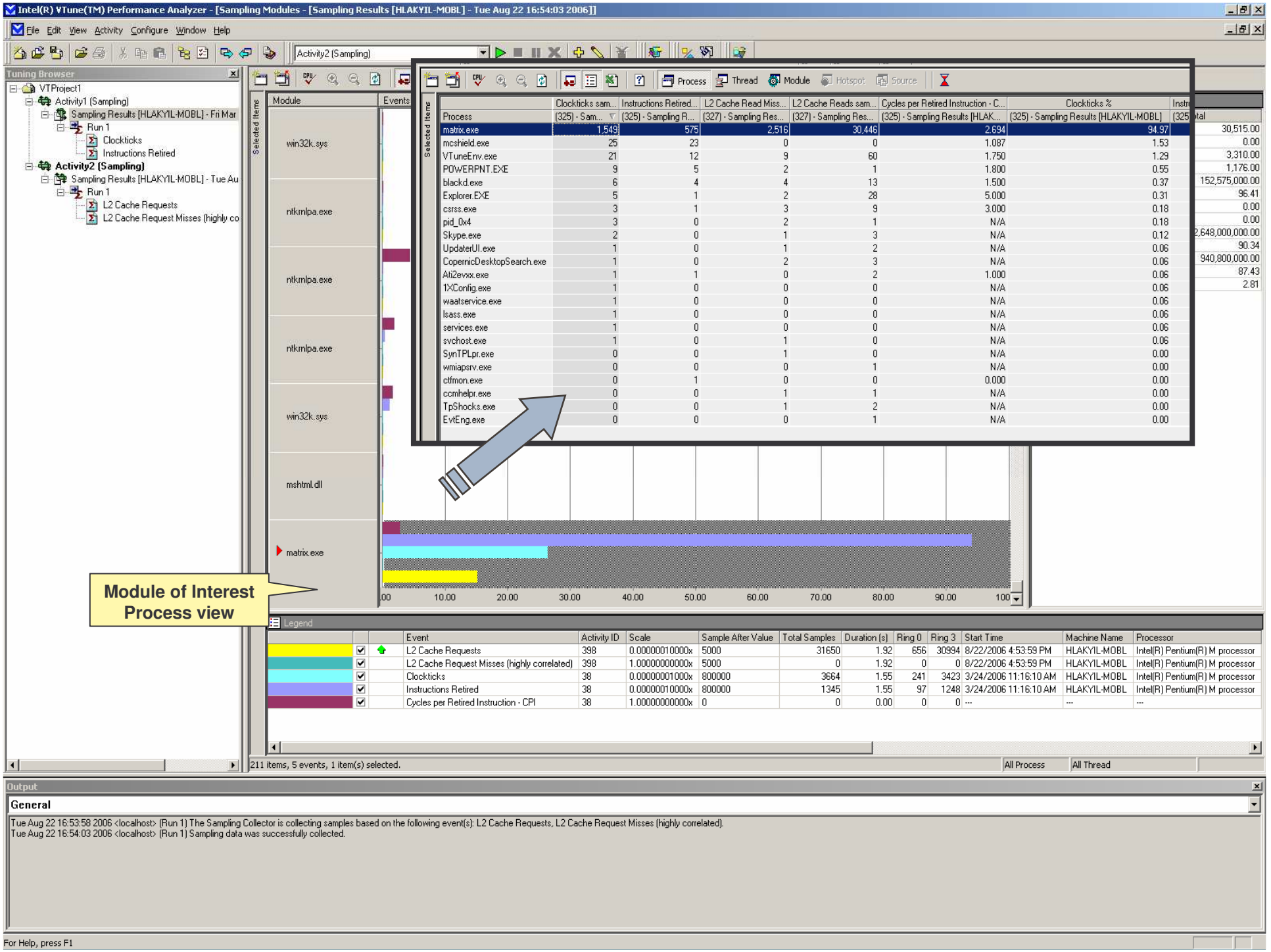
Identifies hard to find performance bottlenecks

Via Call Graph

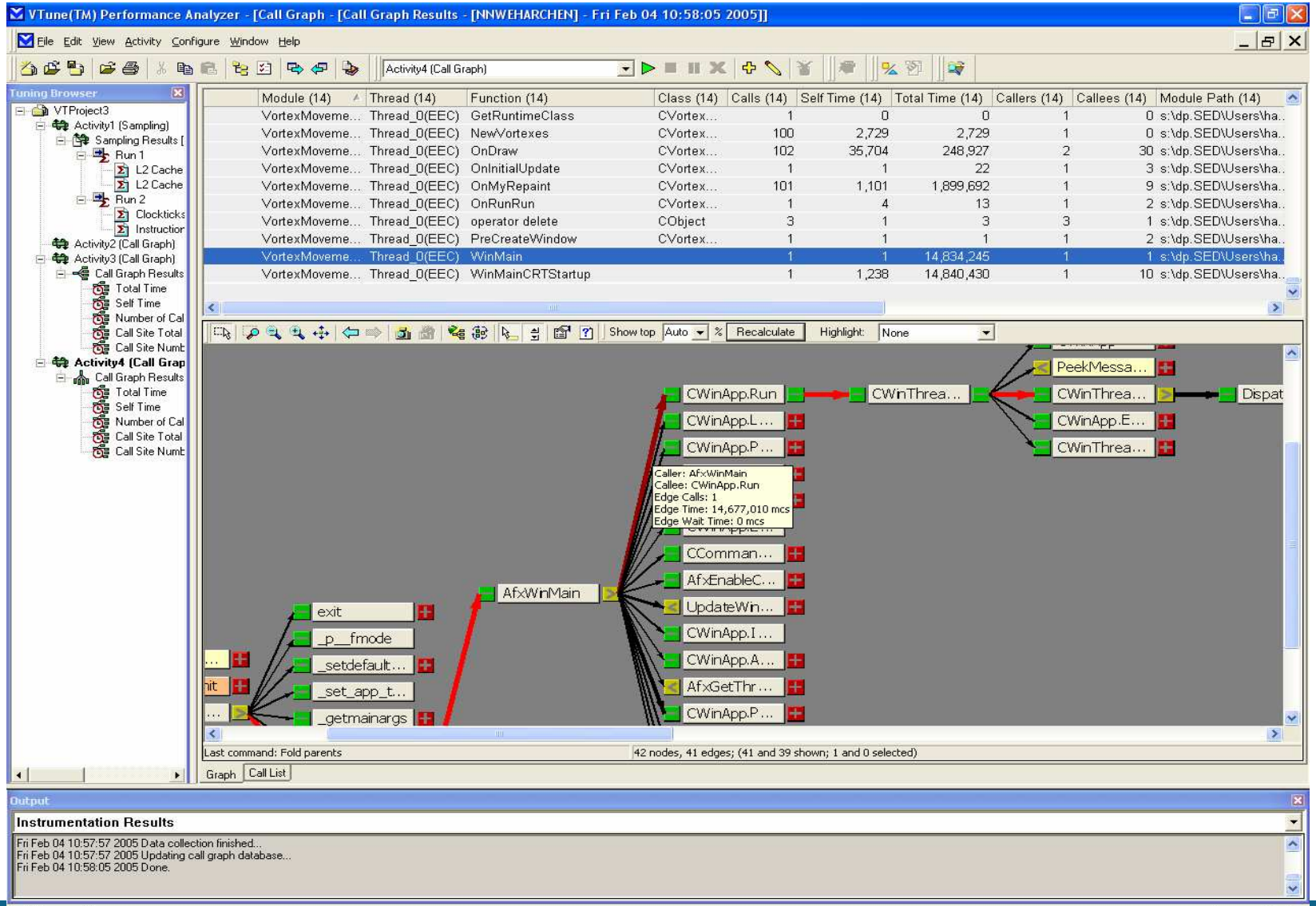
- Using instrumented application
- Some impact on execution time
- All details related to function calls collected at run-time
- Provides call relationships, frequency of calls, time spent in functions
- Quickly identifies critical path in call-tree

Via Sampling

- Using statistical sampling approach to analyze where time (CPU cycles) is spent or micro-architectural event like cache misses are occurring
- Processor is interrupted in fixed small intervals; execution context is saved for later analysis
- Almost not intrusive at all – less then 5%
- GUI provide hierarchical display of sampling data in many ways like e.g. 'Sampling over execution time'


Intel(R) VTune(TM) Performance Analyzer - [Sampling Modules - [Sampling Results [HLAKYIL-MOBL] - Tue Aug 22 16:54:03 2006]]
File Edit View Activity Configure Window Help
Activity2 (Sampling)
Tuning Browser
VTProject1
Activity1 (Sampling)
Sampling Results [HLAKYIL-MOBL] - Fri Mar
Run 1
Clockticks
Instructions Retired
Activity2 (Sampling)
Sampling Results [HLAKYIL-MOBL] - Tue Au
Run 1
L2 Cache Requests
L2 Cache Request Misses (highly co
Module
Events
win32k.sys
ntkrnlpa.exe
ntkrnlpa.exe
ntkrnlpa.exe
win32k.sys
mshtml.dll
matrix.exe
Process
Thread
Module
Hotspot
Source
Selected Items
Process
Clockticks sam... (325) - Sam...
Instructions Retired... (325) - Sampling R...
L2 Cache Read Miss... (327) - Sampling Res...
L2 Cache Reads sam... (327) - Sampling Res...
Cycles per Retired Instruction - C... (325) - Sampling Results [HLAK...
Clockticks % (325) - Sampling Results [HLAKYIL-MOBL]
matrix.exe 1,549 575 2,516 30,446 2.694 94.97
mcshield.exe 25 23 0 0 1.087 1.53
VTuneEnv.exe 21 12 9 60 1.750 1.29
POWERPNT.EXE 9 5 2 1 1.800 0.55
blackd.exe 6 4 4 13 1.500 0.37
Explorer.EXE 5 1 2 28 5.000 0.31
csrss.exe 3 1 3 9 3.000 0.18
pid_0x4 3 0 2 1 N/A 0.18
Skype.exe 2 0 1 3 N/A 0.12
UpdaterUI.exe 1 0 1 2 N/A 0.06
CopernicDesktopSearch.exe 1 0 2 3 N/A 0.06
Ati2evxx.exe 1 1 0 2 1.000 0.06
1XConfig.exe 1 0 0 0 N/A 0.06
waatservice.exe 1 0 0 0 N/A 0.06
lsass.exe 1 0 0 0 N/A 0.06
services.exe 1 0 0 0 N/A 0.06
svchost.exe 1 0 1 0 N/A 0.06
SynTPLpr.exe 0 0 1 0 N/A 0.00
wmiapsrv.exe 0 0 0 1 N/A 0.00
ctfmon.exe 0 1 0 0 0.000 0.00
ccmhelpr.exe 0 0 1 1 N/A 0.00
TpShocks.exe 0 0 1 2 N/A 0.00
EvtEng.exe 0 0 0 1 N/A 0.00
30,515.00
0.00
3,310.00
1,176.00
152,575,000.00
96.41
0.00
0.00
2,648,000,000.00
90.34
940,800,000.00
87.43
2.81
00 10.00 20.00 30.00 40.00 50.00 60.00 70.00 80.00 90.00 100
Module of Interest
Process view
Legend
Event Activity ID Scale Sample After Value Total Samples Duration (s) Ring 0 Ring 3 Start Time Machine Name Processor
L2 Cache Requests 398 0.00000010000x 5000 31650 1.92 656 30994 8/22/2006 4:53:59 PM HLAKYIL-MOBL Intel(R) Pentium(R) M processor
L2 Cache Request Misses (highly correlated) 398 1.00000000000x 5000 0 1.92 0 0 8/22/2006 4:53:59 PM HLAKYIL-MOBL Intel(R) Pentium(R) M processor
Clockticks 38 0.00000001000x 800000 3664 1.55 241 3423 3/24/2006 11:16:10 AM HLAKYIL-MOBL Intel(R) Pentium(R) M processor
Instructions Retired 38 0.00000010000x 800000 1345 1.55 97 1248 3/24/2006 11:16:10 AM HLAKYIL-MOBL Intel(R) Pentium(R) M processor
Cycles per Retired Instruction - CPI 38 1.00000000000x 0 0 0.00 0 0 --- --- ---
211 items, 5 events, 1 item(s) selected.
All Process
All Thread
Output
General
Tue Aug 22 16:53:58 2006 <localhost> (Run 1) The Sampling Collector is collecting samples based on the following event(s): L2 Cache Requests, L2 Cache Request Misses (highly correlated).
Tue Aug 22 16:54:03 2006 <localhost> (Run 1) Sampling data was successfully collected.
For Help, press F1

VTune™ Call graph: Application Workflow

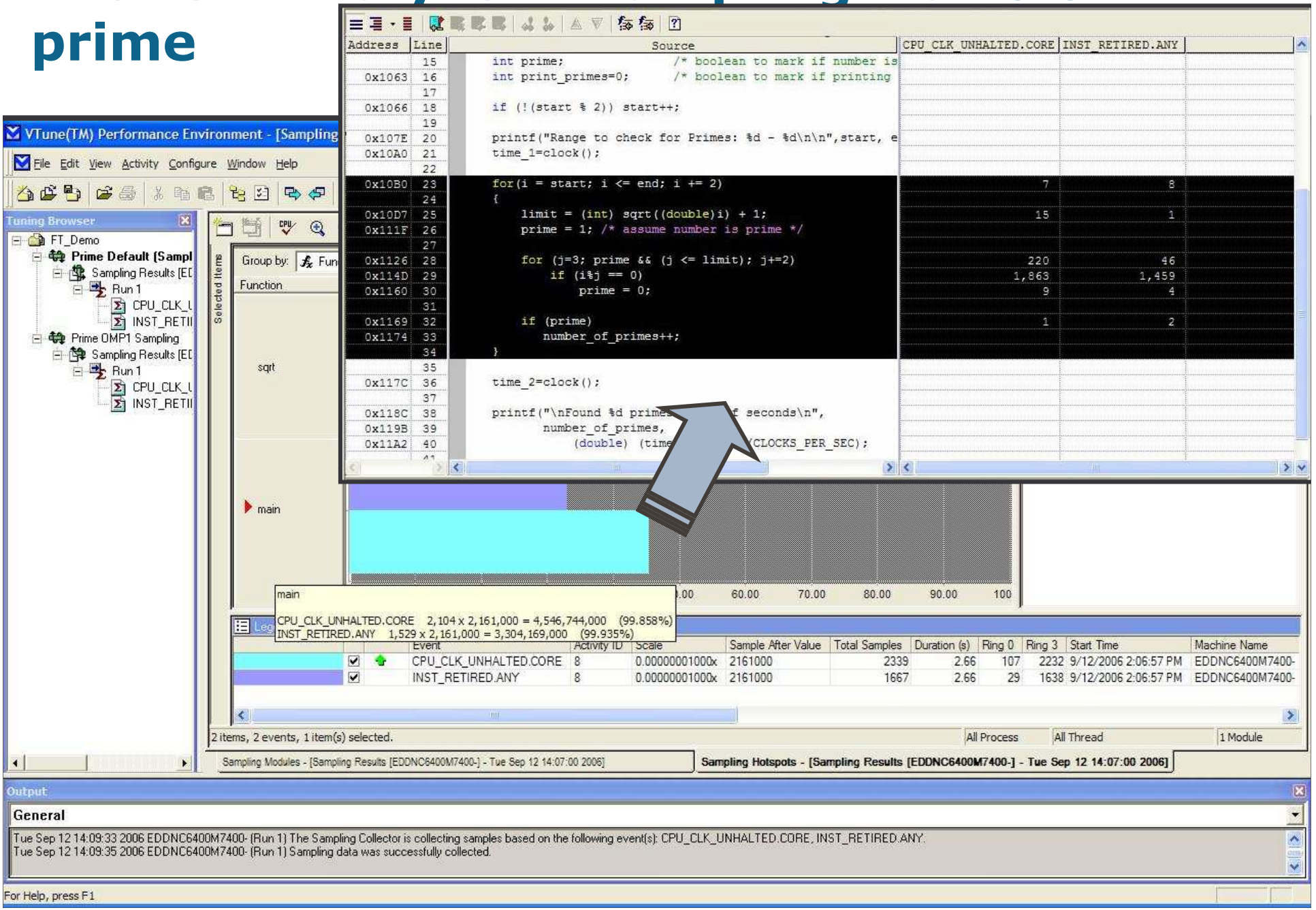


Software

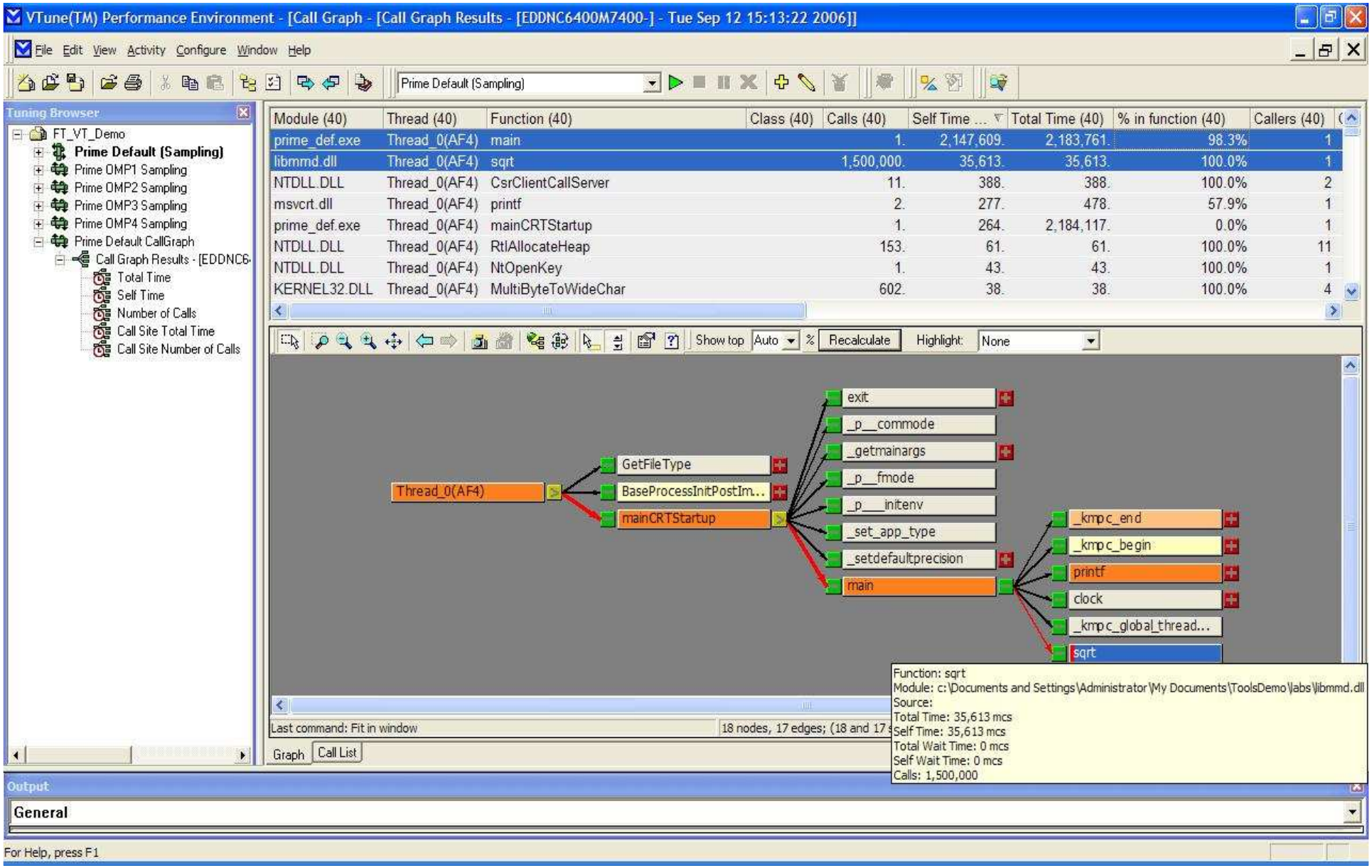
Demo #1

- Analyze the simple Prime application
- Identify hotspots and opportunities for parallelism

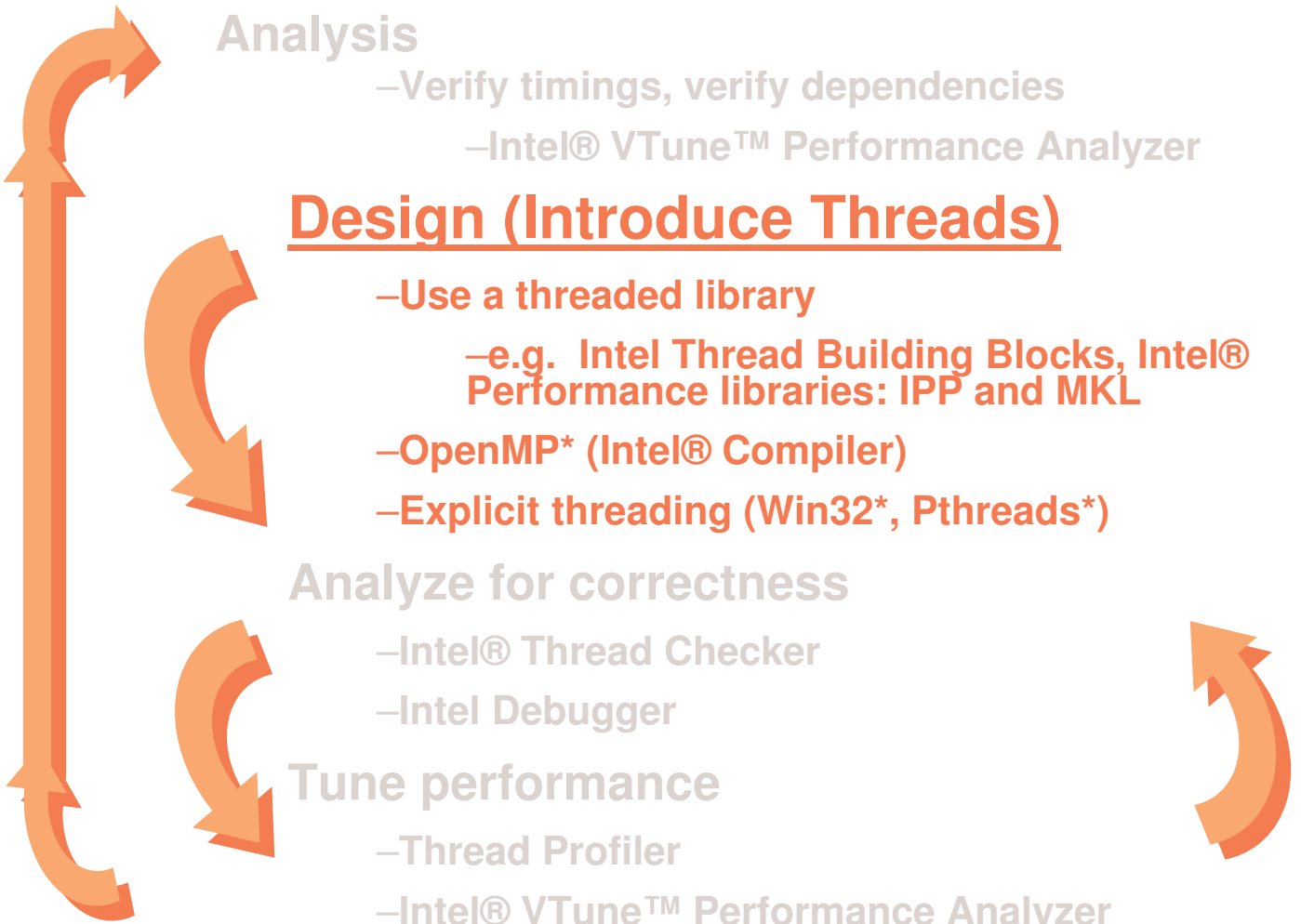
VTune™ Analyzer - Sampling Demo on prime



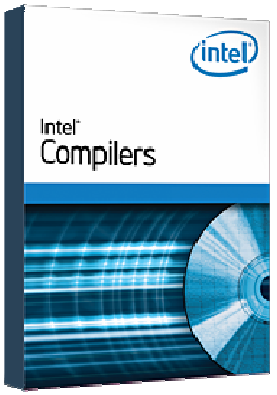
VTune™ Analyzer – Call Graph Demo on prime



Development Cycle



Intel® Compilers: OpenMP* Support



Full support of latest OpenMP* 2.5 standard

- /Qopenmp
- /Qopenmp_report{0|1|2|3}
- /Qopenmp_profile

Automatic parallelization

- /Qparallel
- /Qparthreshold<n> // control over which loops to parallelize

Environment variables to control run time behavior

- E.g. OMP_NUM_THREADS

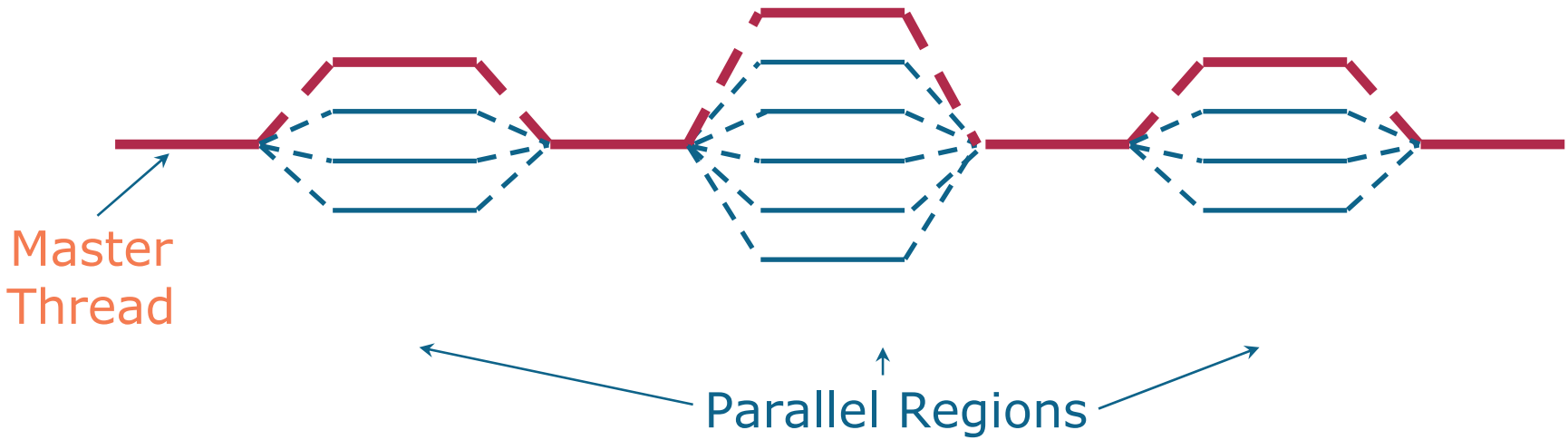
OpenMP™

OpenMP* Architecture

- Fork-join model
- Work-sharing constructs (for, section, single)
- Data environment constructs (private, shared)
- Synchronization constructs (master, critical, barrier, atomic)
- Extensive Application Program Interface (API) for finer control

Programming Model

- Master thread spawns a team of threads
- Parallelism is added incrementally: the sequential program evolves into a parallel program



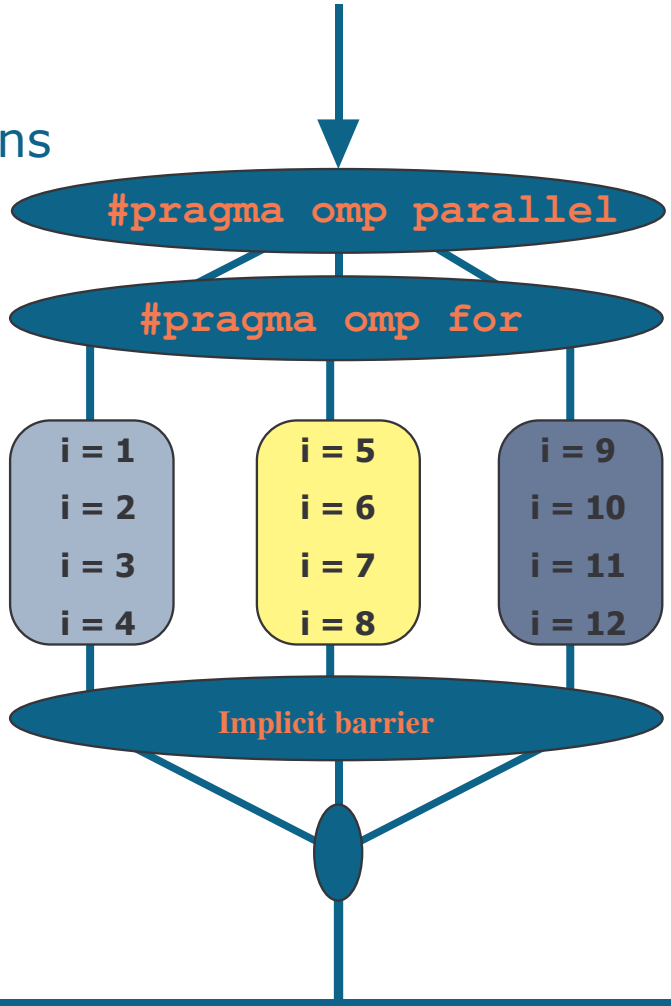
OpenMP* Parallel Loop Model

Threads are created – typically one per core

Each threads executes a subset of all iterations

Data is classified as shared or private

```
#pragma omp parallel for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```

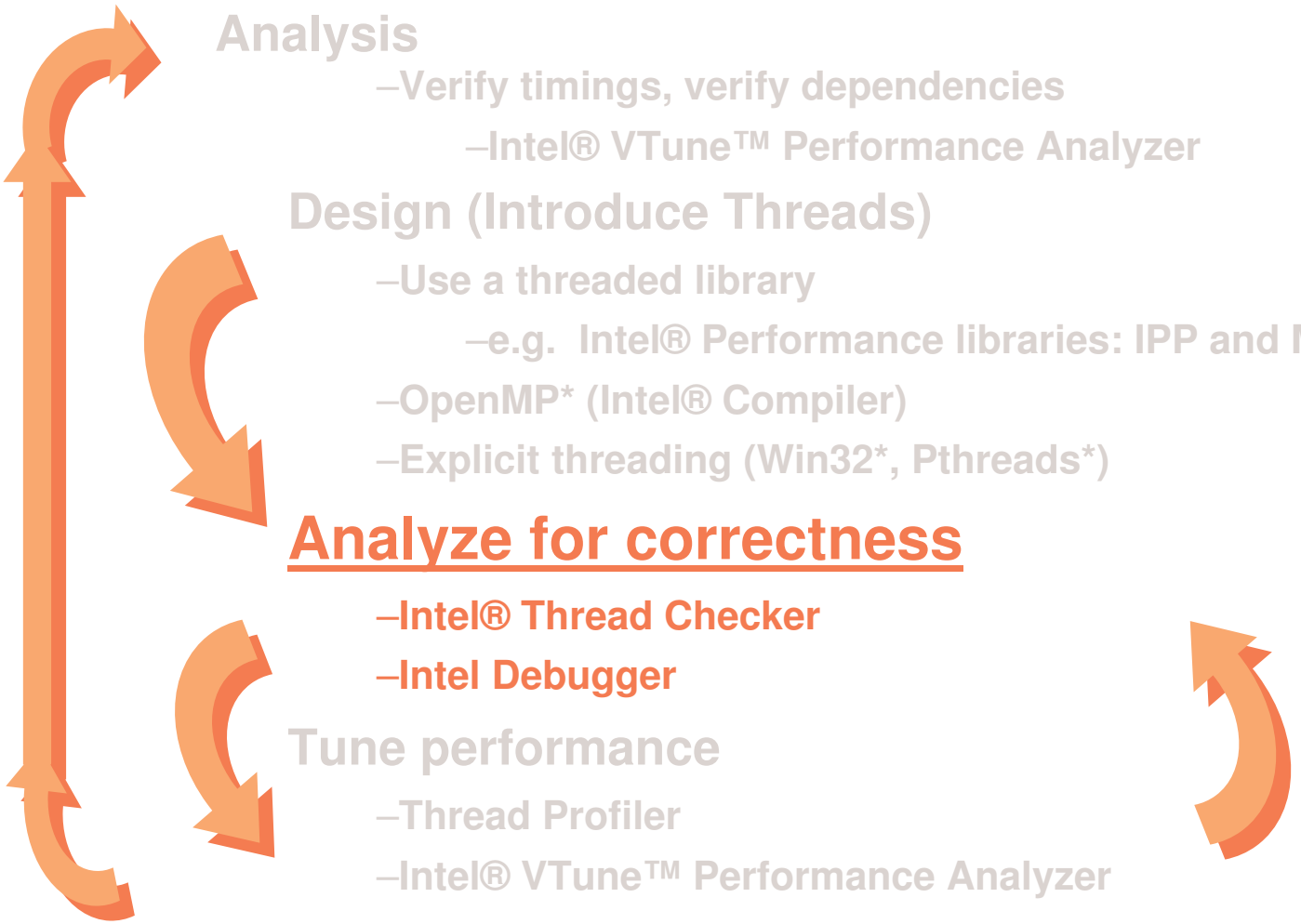


Demo #2

1. Parallelization, compilation for OpenMP*
2. Compilation and execution of sequential code
   - icl /MD /Zi /Qopenmp -o prime_omp1 prime_omp1.c

```
#pragma omp parallel for private (j, limit, prime)
for(i = start; i <= end; i += 2){
  limit = (int) sqrt((float)i) + 1;
  prime = 1; /* assume number is prime */
  for ( j=3;  (prime && (j <= limit); j+=2 )
    if (i%j == 0) prime = 0;
  if (prime)
    number_of_primes++;
}
```

Development Cycle



Intel® Thread Checker

Identification of data safety issue:

- Dead-locks, race conditions etc

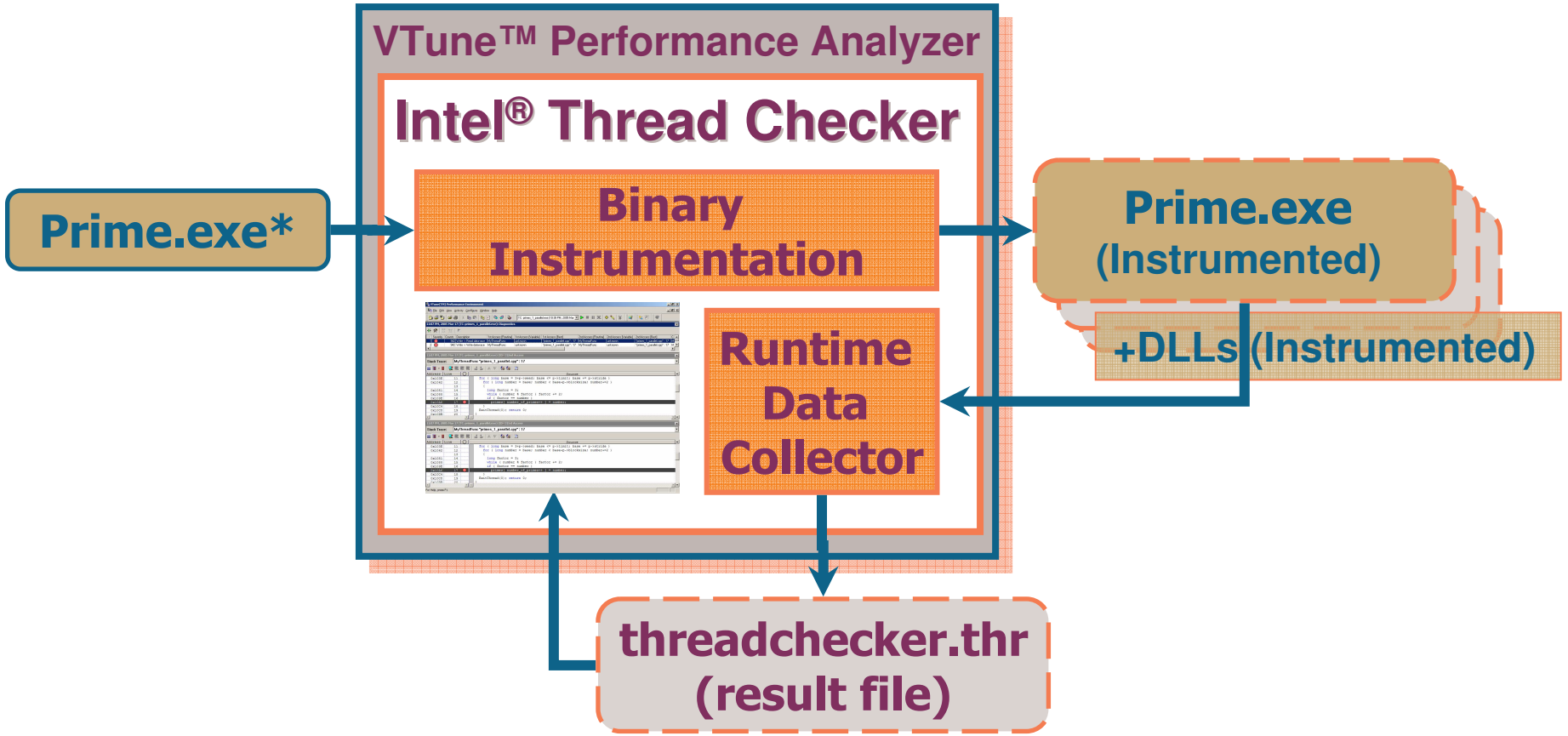
Supported environments:

- OpenMP*
- Native thread-API on Microsoft Windows* systems (Win32 Threads) and Linux* (PThreads)

Analysis :

- Instrumented application
- Dynamic monitoring while application runs

Intel® Thread Checker



* Compiled: /Zi, Linked: /debug /fixed:no

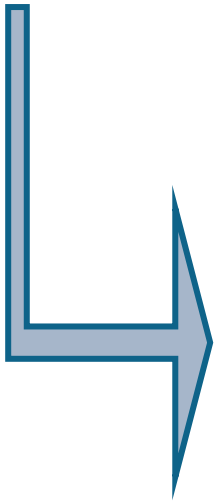
Demo #3

- Intel® Thread Checker analysis to find potential thread-specific (OpenMP*-specifc) coding issues

```
#pragma omp parallel for private (j, limit, prime)
for(i = start; i <= end; i += 2){
  limit = (int) sqrt((float)i) + 1;
  prime = 1; /* assume number is prime */
  for ( j=3;  (prime && (j <= limit); j+=2 )
    if (i%j == 0) prime = 0;
  if (prime)
    number_of_primes++;
}
```

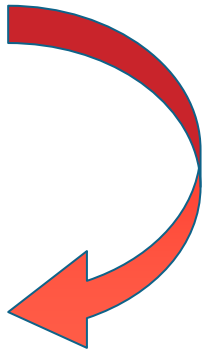
Pre-emptive context switching and shared memory conflicts (Data Races)

```
number_of_primes++
```



```
<thread 1>
mov ecx, dword ptr [number_of_primes]
add ecx, 1
mov dword ptr [number_of_primes], ecx

<thread 2>
mov ecx, dword ptr [number_of_primes]
add ecx, 1
mov dword ptr [number_of_primes], ecx
```

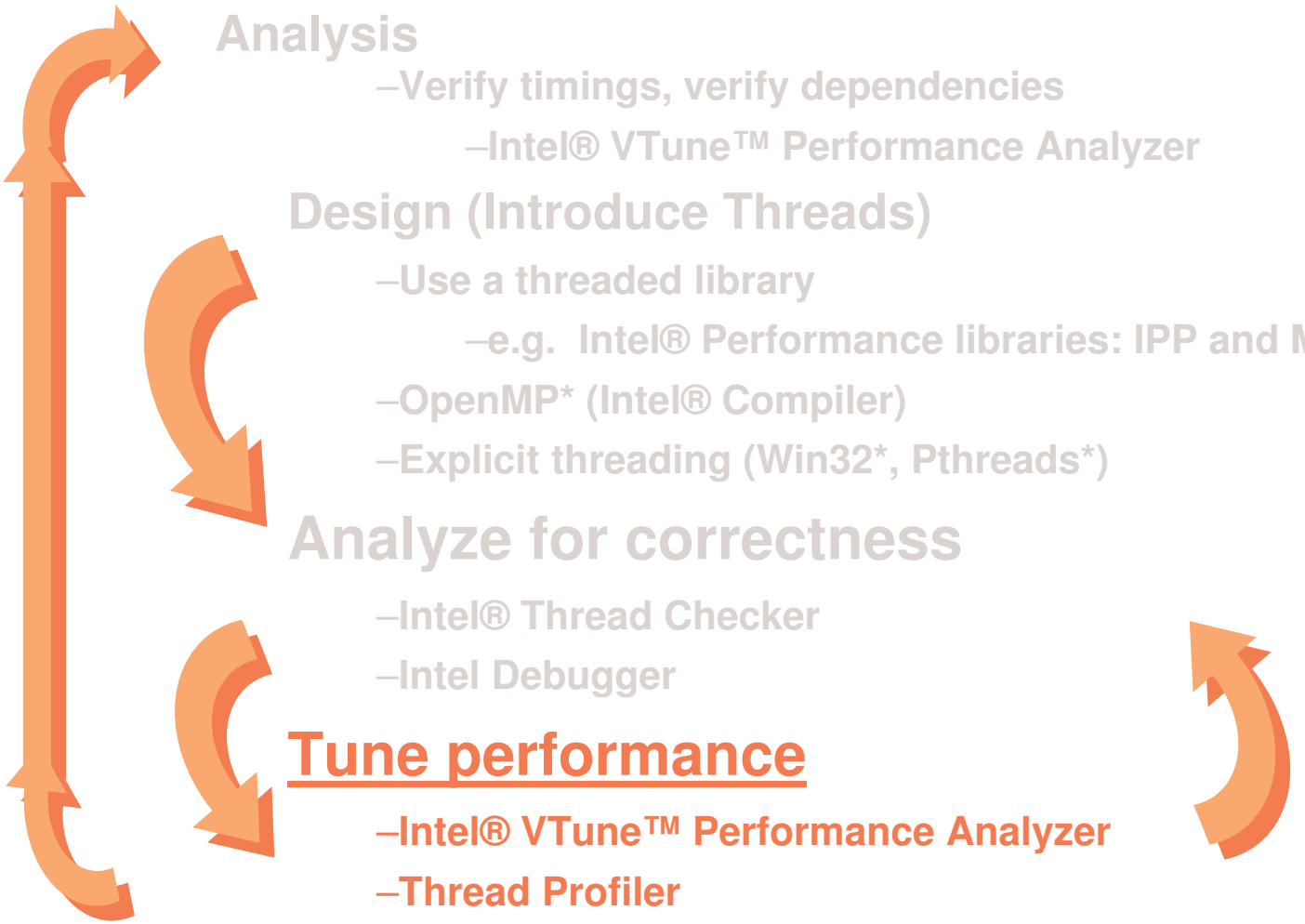


Demo #3 continued

- Intel® Thread Checker analysis after guarding critical instruction
  - icl /MD /Zi /Qopenmp -o prime_omp2 prime_omp2.c

```
#pragma omp parallel for private (j, limit, prime)
for(i = start; i <= end; i += 2){
  limit = (int) sqrt((float)i) + 1;
  prime = 1; /* assume number is prime */
  for ( j=3;  (prime && (j <= limit); j+=2 )
    if (i%j == 0) prime = 0;
  if (prime)
      #pragma omp critical
            number_of_primes++;
}
```

Development Cycle



Analysis

–Verify timings, verify dependencies

–Intel® VTune™ Performance Analyzer

Design (Introduce Threads)

–Use a threaded library

–e.g. Intel® Performance libraries: IPP and MKL

–OpenMP* (Intel® Compiler)

–Explicit threading (Win32*, Pthreads*)

Analyze for correctness

–Intel® Thread Checker

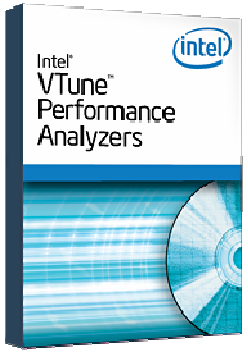
–Intel Debugger

Tune performance

–Intel® VTune™ Performance Analyzer

–Thread Profiler

Intel® VTune™ Performance Analyzer



Identifies hard to find performance bottlenecks

Via Callgraph

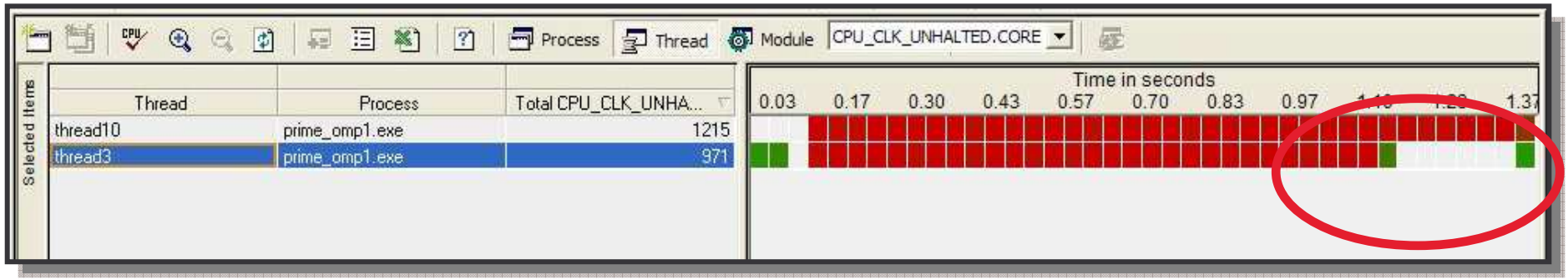
- Using instrumented application
- Some impact on execution time
- All details related to function calls collected at run-time
- Provides call relationships, frequency of calls, time spent in functions
- Quickly identifies critical path in call-tree

Via Sampling

- Using statistical sampling approach to analyze where time (CPU cycles) is spent or mirco-architectural event like cache misses are occurring
- Processor is interrupted in fixed small intervals; execution context is saved for later analysis
- Almost not intrusive at all – less then 5%
- GUI provide hierarchical display of sampling data in many ways like e.g. 'Sampling over execution time"

Demo #4

- Using *Sampling over Time* feature of Intel® VTune™ Performance Analyzer for initial load balancing checking



Threads are not balanced

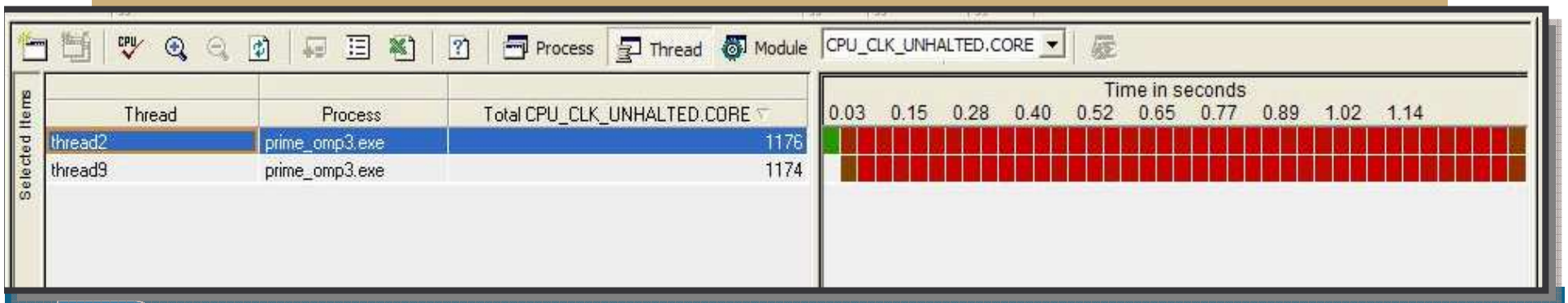
Assigning Loop Iterations in OpenMP*: Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
RUNTIME	Modify schedule at run-time via environment variable

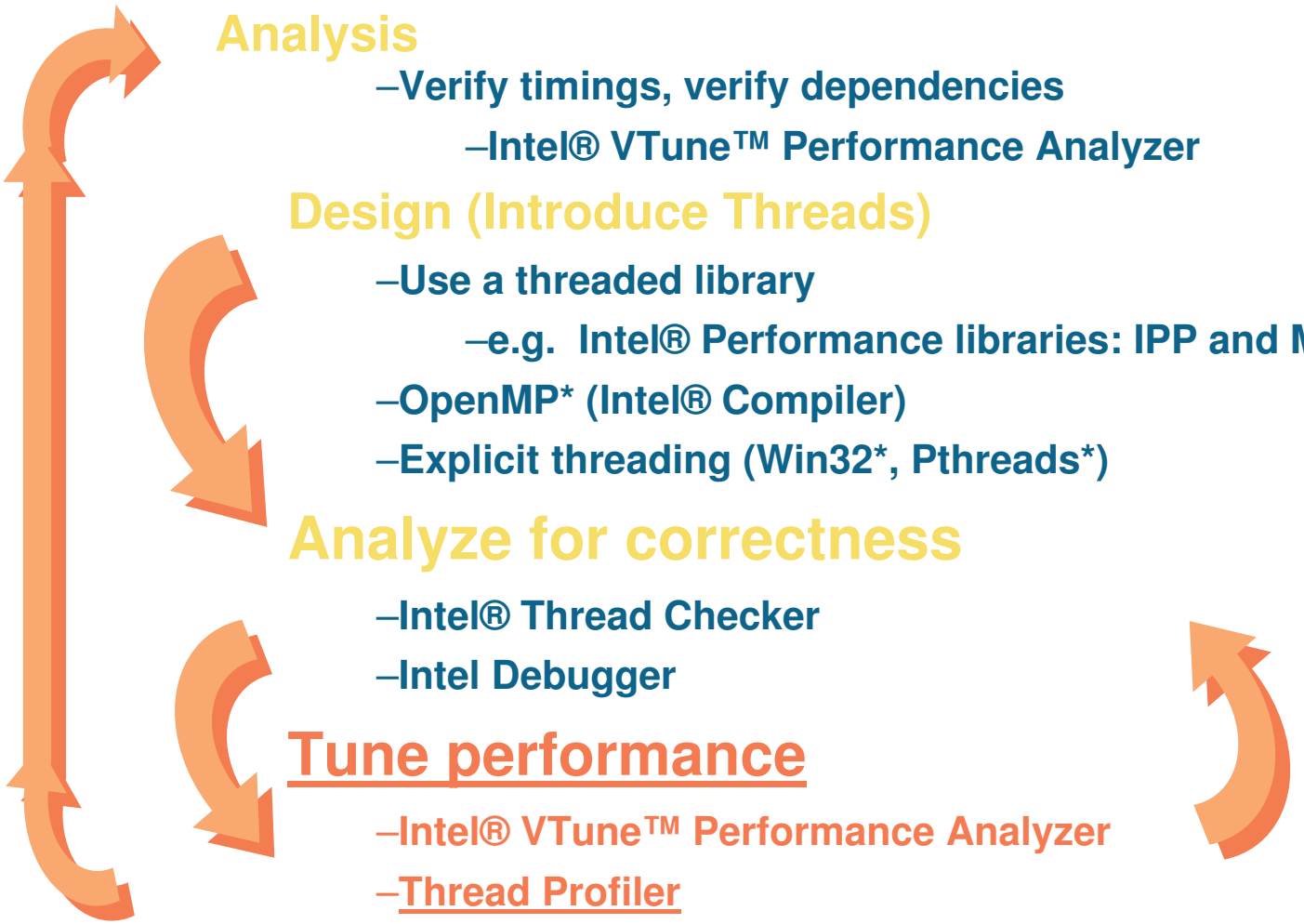
Demo #4 continued

- Using VTune *Sampling over Time* after changing scheduling strategy to *dynamic*

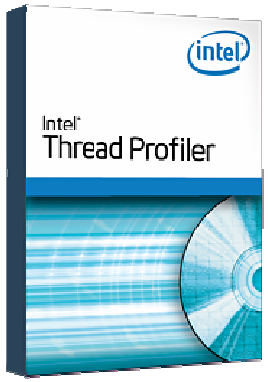
```
#pragma omp parallel for schedule (dynamic)
                         private (j, limit, prime)
for(i = start; i <= end; i += 2){
  limit = (int) sqrt((float)i) + 1;
  prime = 1; /* assume number is prime */
  for ( j=3;  (prime && (j <= limit); j+=2 )
    if (i%j == 0) prime = 0;
```



Development Cycle



Intel® Thread Profiler



Detection of thread-specific bottlenecks:

- Load imbalance
- Contention on synchronization objects
- Threading overhead

Supported environments:

- OpenMP*
- Native thread-API on Microsoft Windows* (Win32 Threads)
- Native thread-API on Linux* (PThreads)

Analysis:

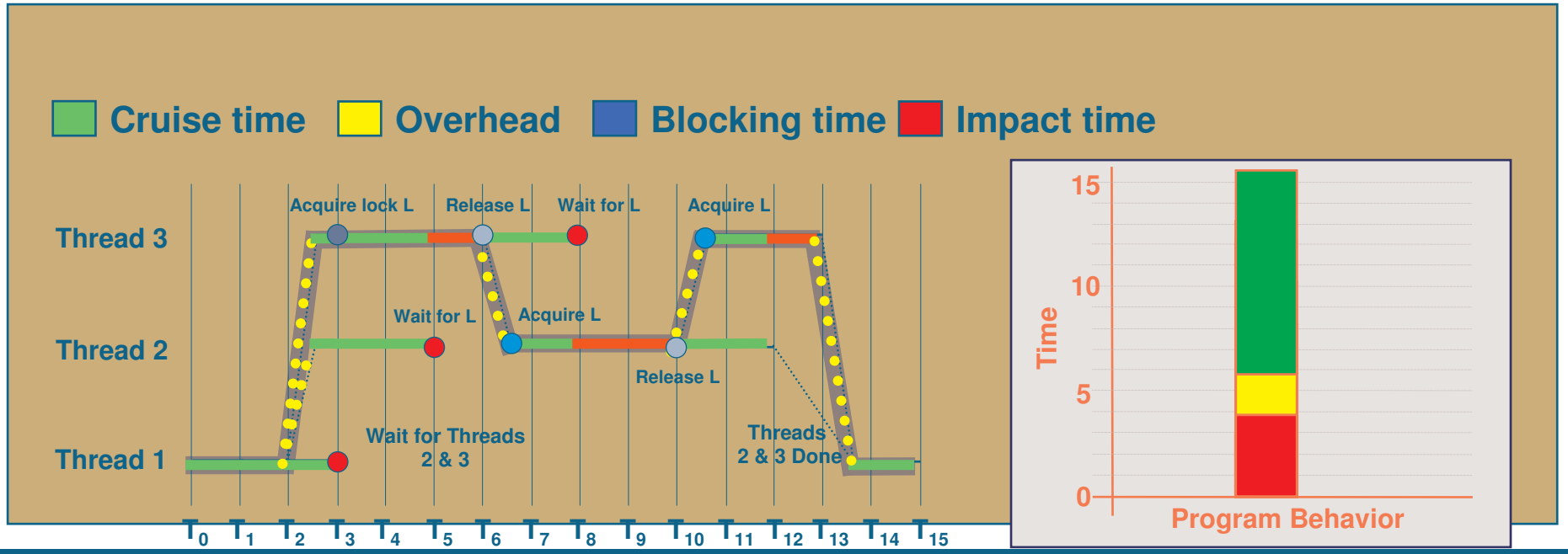
- Binary and/or source instrumentation
- Profiled run-time libraries (-Qopenmp_profile)

Execution Time Categories

Analyze program behavior along critical path execution

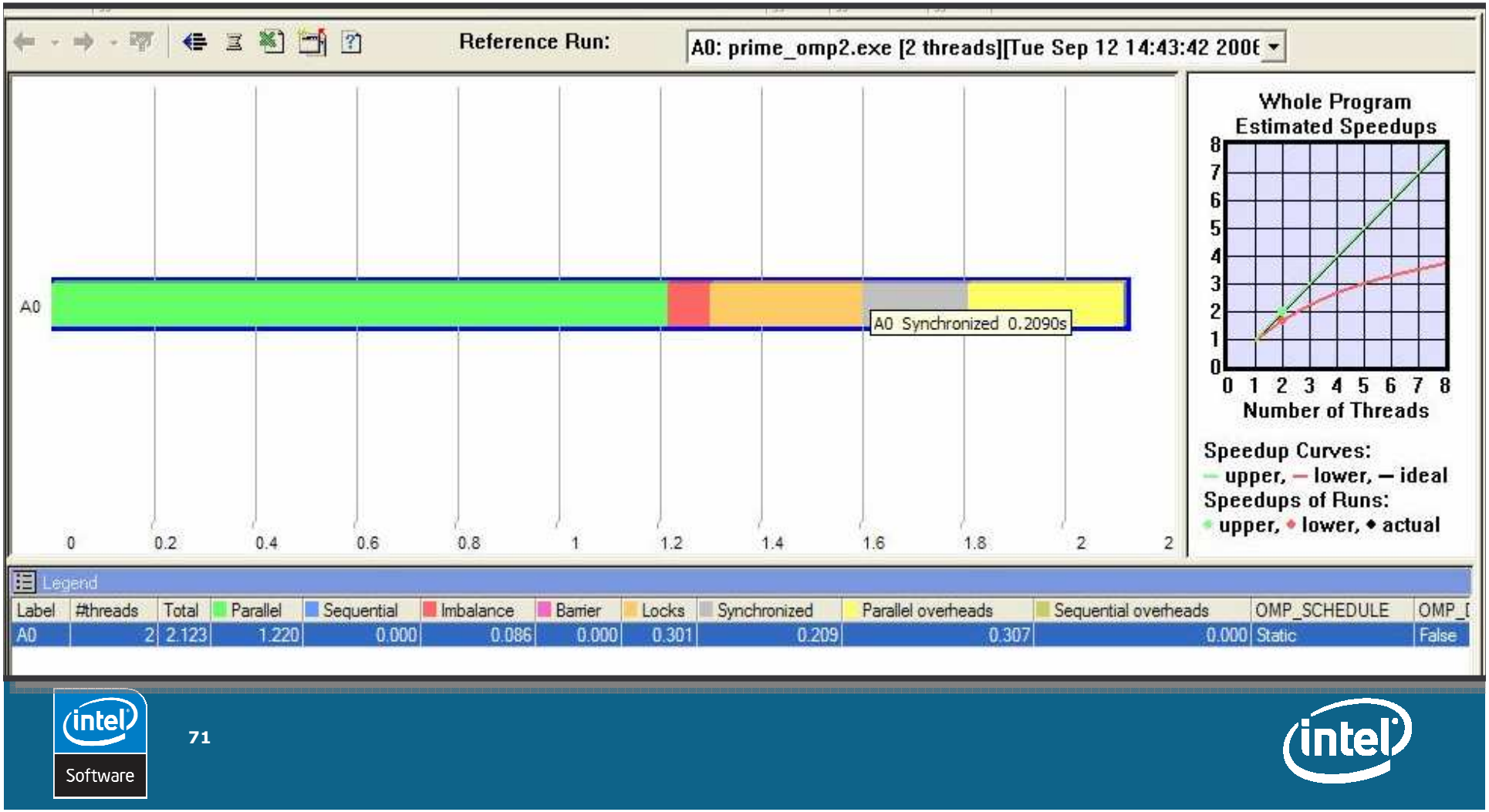
Record objects that cause transitions

Categorization shown for a system configuration with 2 processors



Demo #5

- Using Intel® Thread Profiler to fine-tune prime application

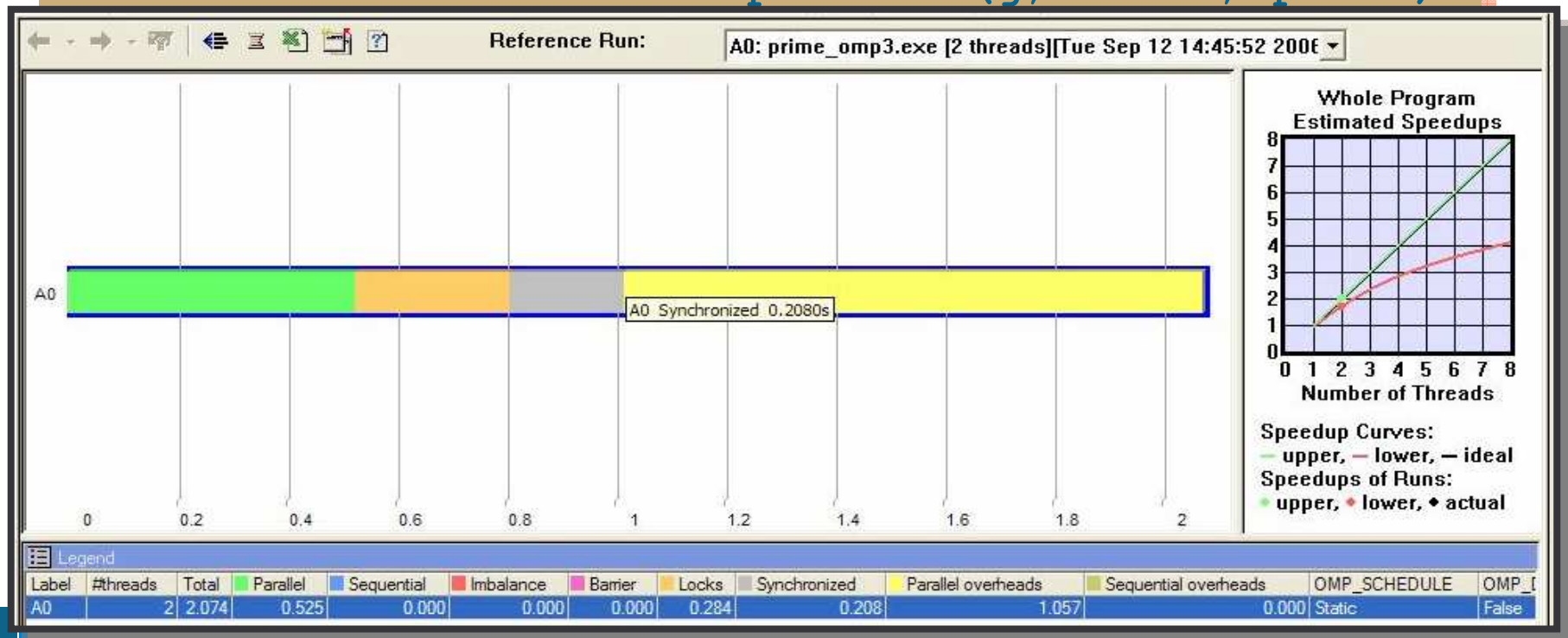


Label	#threads	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads	Sequential overheads	OMP_SCHEDULE	OMP_[
A0	2	2.123	1.220	0.000	0.086	0.000	0.301	0.209	0.307	0.000	Static	False

Demo #5 continued

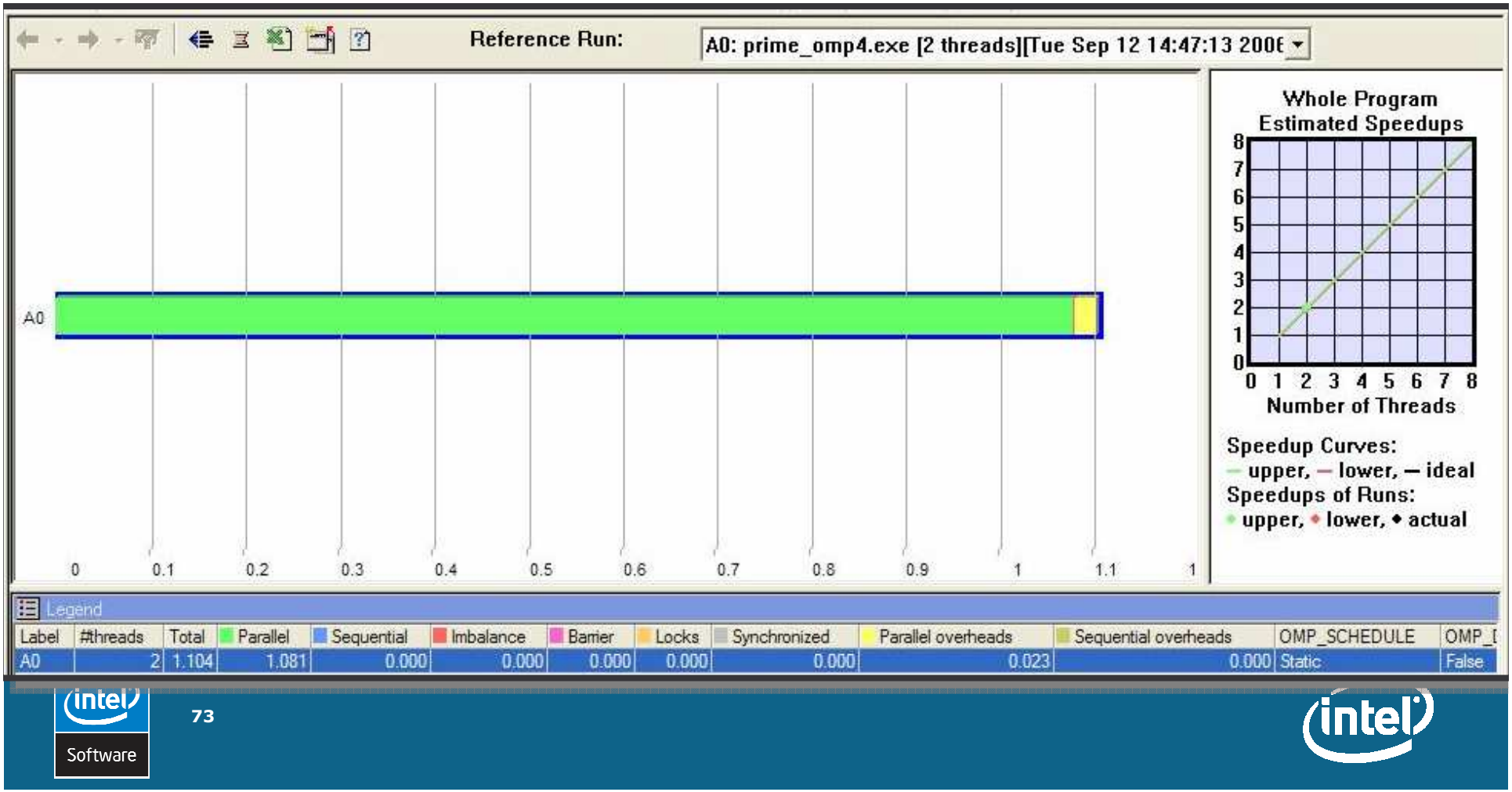
- Final Intel® Thread Profiler analysis after optimizing loop scheduling and synchronization

```
#pragma omp parallel for schedule (dynamic)
                         private (j, limit, prime)
```



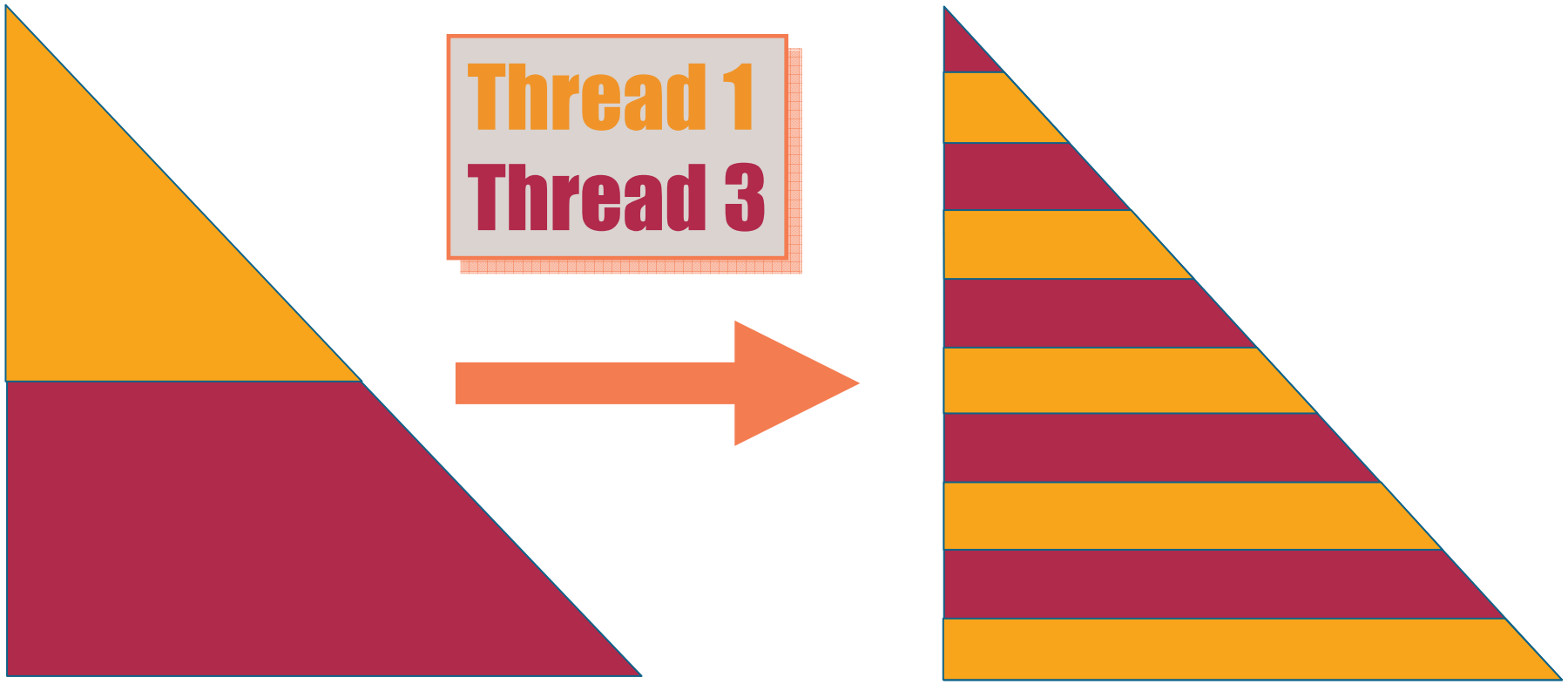
Demo #5 continued

- Final Intel® Thread Profiler analysis after optimizing loop scheduling and synchronization

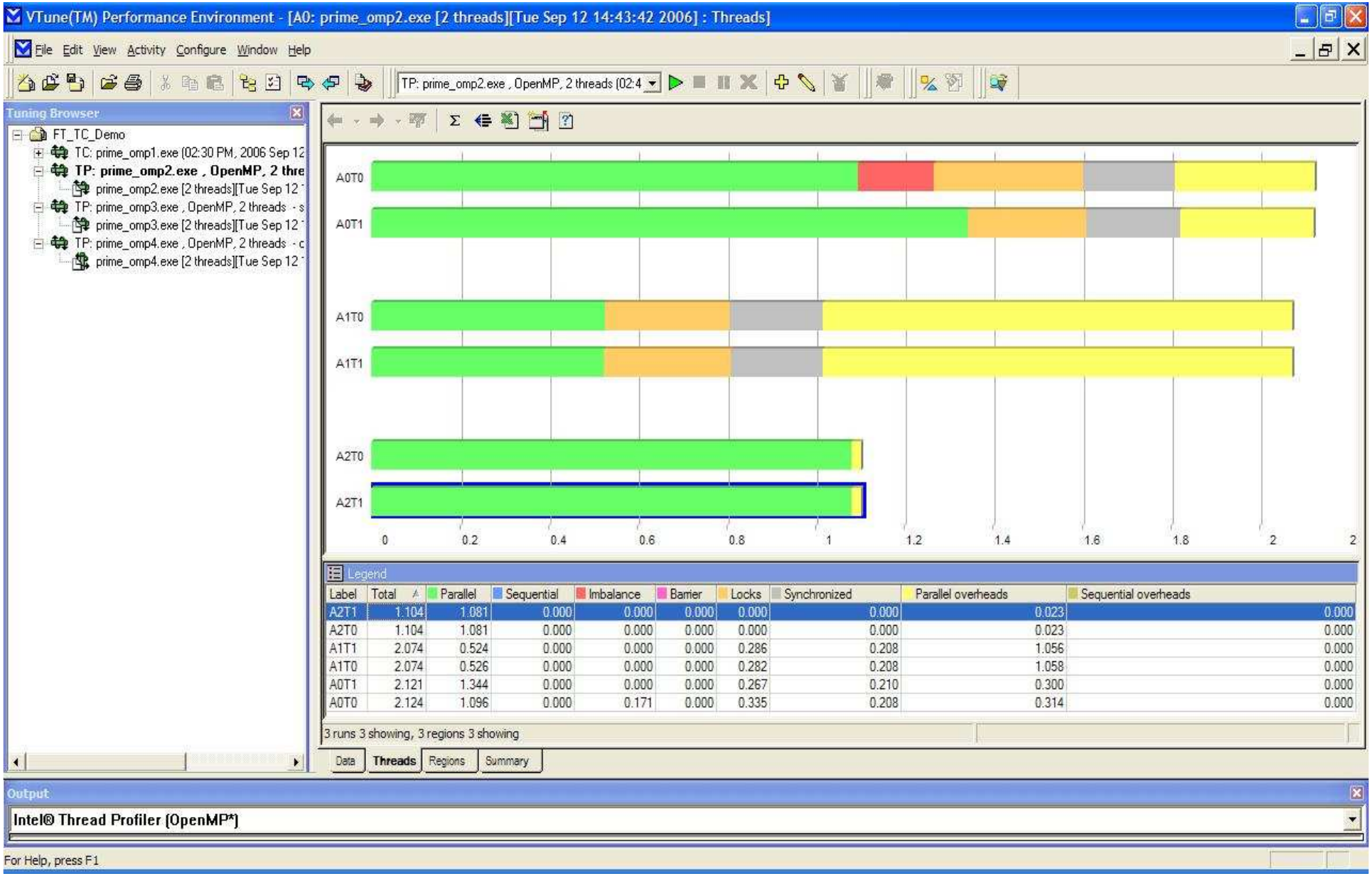


Intel® Thread Profiler Results

Adopt better work load distribution



Putting it together



Agenda

Motivation for Threading

Concepts in Parallelism

Threading in Action

Common Threading Errors

Existing Code and Intel® Tools

Summary

Summary

Concurrent compute threads are required to take full advantage of multi-processor/multi-core systems

Properly threading your code is challenging

- You don't have to do it all at once
- Focus on areas that will have the most impact to overall application performance
- Make sure to monitor scaling on systems with more processors than the current common customer configuration

A thorough understanding of actual application behavior under typical loads is necessary before you consider threading

Intel continues to invest in hardware and software to ease the transition to threaded code

But Don't Take MY Word for It

"We are optimizing RenderMan's core to be very scalable for future multi-core architectures. Intel's Threading Tools have accelerated our development cycle dramatically. Intel's Thread Checker for example, helped identify potential threading issues very quickly, in days compared to weeks if done otherwise. Thread Profiler, on the other hand, has helped us understand threading performance problems so we could fix them to improve scalability. The Intel Threading Tools are now an integral part of our development process."

Dana Batali
Director of RenderMan Development
Pixar

But Don't Take MY Word for It

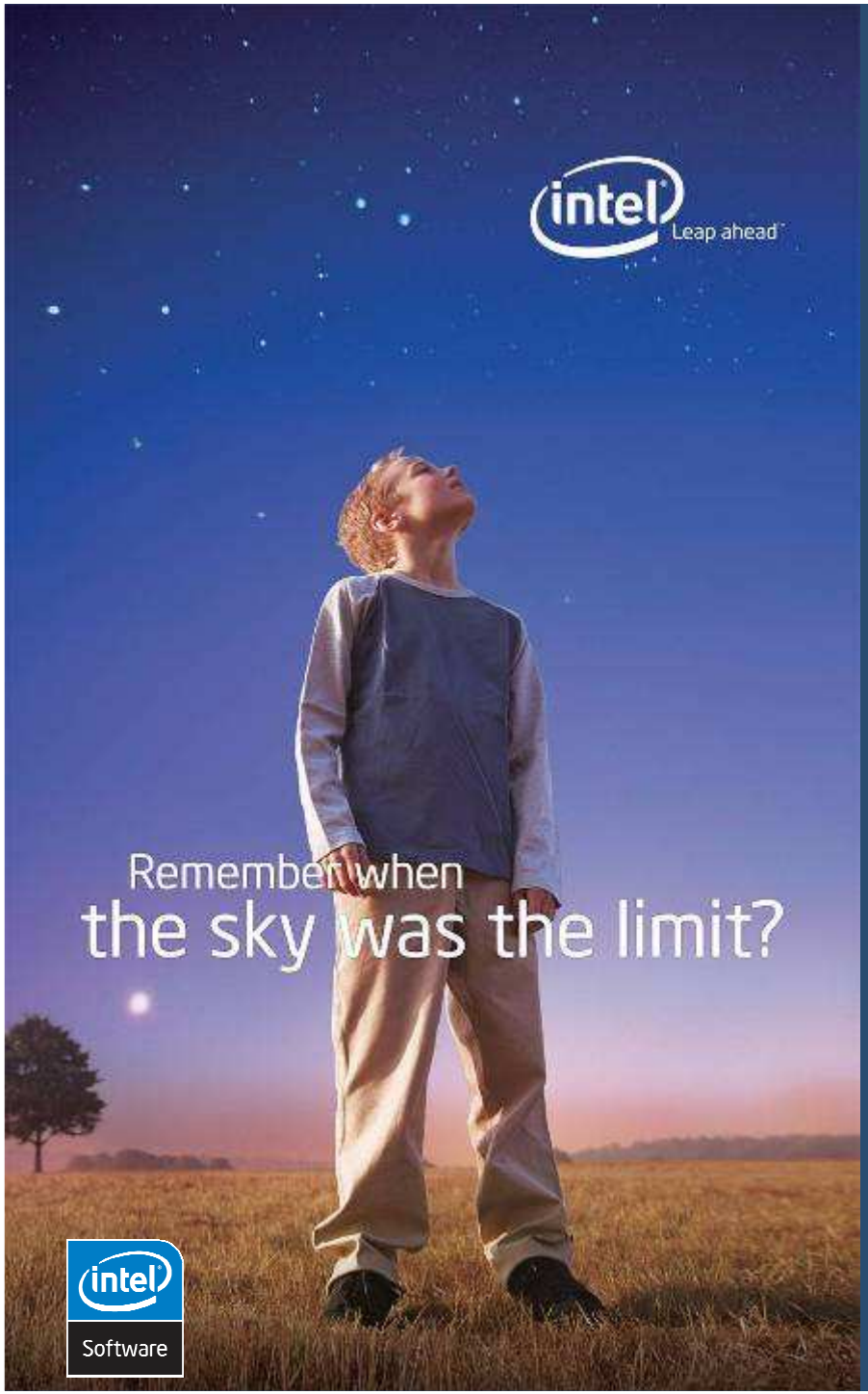
"We found Intel ThreadChecker to be an **indispensable aid** for analyzing threaded code. We were impressed at how well it handled an application as large and complex as Maya. Based on this experience I plan to use this tool on future threading projects. Intel ThreadProfiler was very useful for analyzing bottlenecks in our threaded code. ThreadProfiler . . . showed us the reasons for the slowdown, so we were able to restructure the code for better threaded performance."

Martin Watt
Software Architect
Alias

But Don't Take MY Word for It

"We used Intel Threading Tools including Intel Thread Profiler to realize improved threaded application performance of Omni Page 15 running on Intel multi-core platforms. We look forward to using Intel Thread Profiler with its critical path analysis and selective magnification of important time regions on future thread optimization projects."

Gyorgy Varszegi
Scansoft


intel
Leap ahead
Remember when
the sky was the limit?
intel
Software

Thank you