Leap ahead™

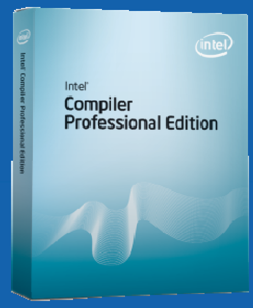
Exploiting the Power of the Intel® Compiler Suite

Dr. Mario Deilmann
Intel® Compiler and Languages Lab
Software Solutions Group

Software Products

Agenda

- Compiler Overview
  - Intel® C++ Compiler
- High level optimization
  - IPO, PGO
- Vectorization
  - Loops and Co.



Why does Intel make Compilers ?

- Performance, performance and performance
  - It all started in the early 90s by the need to provide best SPEC numbers for Intel processors-based systems
- Early Availability
  - HW support & testing, enabling, eco-system initializing
  - Utilize SIMD registers through vectorization Influence on SW industry
  - Promote features like SSE, OpenMP

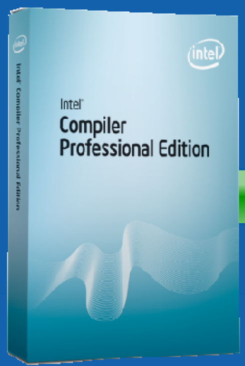
Compilers Provided in the Intel Compiler Package

- The Intel® IA32 Compiler
  - Generating binaries for IA32 systems
- The Intel® Intel 64 Compiler
  - Generating binaries for Intel 64-based or compatible systems
- The Intel® Itanium® Compilers
  - Generating binaries for Itanium-based systems

Supported Platforms

Product	Architecture	OS/platform
Intel® C++ Compiler	IA32/ EM64T	Windows*
		Linux*
		Mac OS* (starting with 9.1)
	IA-64	Windows*
		Linux*
	Xscale™	Microsoft eMbedded Visual C++*
		Platform Builder for Win CE .NET*
Intel® Fortran Compiler	IA32/ EM64T	Windows*
		Linux*
		Mac OS* (starting with 9.1)
	IA-64	Windows*
		Linux*

Two Product Lines for Two Kinds of Developers



Maximize parallel performance
C++ and Fortran on Windows, Linux* and Mac OS* X*

Sample: Intel® C++ Compiler Professional Edition

Traditional Tools, available since years

Maximize parallel productivity
C++ using Visual Studio on Windows

Sample: Intel® Parallel Composer

New, part of Intel® Parallel Studio

C++ and Fortran Compilers

- Professional Edition for Windows*, Linux* or MacOS : Bundles Intel® Compiler with Libraries
  - Intel® C++ Compiler or Intel® Fortran Compiler
  - Intel® Math Kernel Library (C++ & Fortran)
  - Intel® Integrated Performance Primitives Library (C++)
  - Intel® Threading Building Blocks (C++)

- Intel® Parallel Composer as part of Intel® Parallel Studio for Windows* only:
  - Intel® C++ Compiler
  - Intel® Integrated Performance Primitives Library (C++)
  - Intel® Threading Building Blocks (C++)

Some Generic Features

- Compatibility to standards (ANSI C, ISO C++, ANSI C99, Fortran95, Fortran2003)
- Compatibility to leading open-source tools (ICC vs. GCC, IDB vs GBD, ICL vs CL, …)
  - Windows* compiler fully integrates into MS VS 05/08
- Supports all instruction set extensions via vectorization
  - Automatic and manual code dispatching
- OpenMP* support and Automatic Parallelization
- Sophisticated optimizations
  - Profile-guided optimization
  - Multi-file inter-procedural optimization
- Detailed compilation report generation

Intel® Compilers 11.x

- Features:
  - Multi-core processor support through Auto-parallelization
  - OpenMP* 3.0
  - Advanced optimization technology: (automatic) vectorization, interprocedural Optimization, profile-guided Optimization
  - Full compatibility with Windows*, Linux* and Mac OS* X development environments
- What's New:
  - New High-performance, parallel optimizer (HPO)
  - Parallel execution of inter-procedural Optimization
  - Optimization Reports for Advanced Loop Transformations
  - Static Verifier – security, buffer overflow, OpenMP verification
  - SSE4.2 Intel® Core™ i7 processor (Nehalem) new processor support
  - ANSI/ISO C++ standard support, support parts of C++0x
  - Substantial Fortran 2003 support

Compatibility with Microsoft

Source and binary compatible

- Full compatibility with .Net* Build Environment
- Binary compatibility
- Name Mangling and Calling Convention
- Debug Format Compatibility
- Mix and match object files or DLLs

Limitations

- No support of *.pch* file; instead use *.pchi* file
- No support for attributed programming, managed C++

Compatibility with Linux (MacOSX)

Source and binary compatible

- Mixing and matching binary files created by g++, including third-party libraries
- Generating C++ code compatible with gcc/g++ 3.2 or higher (up to 4.4)
- Improved support for command-line options offered in the GNU compilers
- Support of most GNU C and C++ language extensions

Limitations

- Intel Fortran Compiler for Linux is not binary compatible with GNU g77 or GNU gfortran compiler

Compatibility with Linux (cont)

gcc/g++ language extensions

- We support most of the GNU gcc language extensions (47 out of 56)

Limitations

- No support for:
  - Nested functions
  - Constructing function calls
  - Looser Rules for Escaped Newlines
  - Prototype and Old-Style Function Definitions
  - Using Vector Instructions Through Built-in Functions
  - Built-in Functions Specific to Particular Target Machines
  - Java* Exceptions
  - Deprecated Features
  - Backward Compatibility
- We can successfully compile the Linux kernel 2.4.21 and 2.6.9 with Intel C++ Compiler on IA-32, Intel® 64 and IA-64, with a small wrapper script and patches

A few General Switches

Functionality	Windows*	Linux* & Mac OS*
Disable optimization	/Od	-O0
Optimize for speed (no code size increase), no SWP	/O1	-O1
Optimize for speed (default), includes SSE & SWP for IPF	/O2	-O2
High-level optimizer (e.g. loop unroll)	/O3	-O3
Aggressive optimizations (= -xHOST -O3 –ipo –static –prec-div-)	/fast	-fast
Create symbols for debugging	/Zi	-g
Generate assembly files	/Fa	-S
Optimization report generation	/Qopt-report	/opt-report
OpenMP 3.0 support	/Qopenmp	-openmp
Automatic parallelization for OpenMP threading	/Qparallel	-parallel

High-Level Optimizer (HLO)

- Overview
  - Loop level optimizations
    - loop unrolling, cache blocking, prefetching
  - More aggressive dependency analysis
    - Determines whether or not it is safe to reorder or parallelize statements
  - Scalar replacement
    - The goal of scalar replacement is to reduce memory references with register references.
- How to enable HLO
  - (Linux*) –O2,–O3, (Windows*) /O2,/O3
- But loops must meet certain criteria …

Multi-pass Optimization - Interprocedural Optimizations

- Interprocedural optimization works on the entire program across procedures and file boundaries
- Enabled optimizations:
    - Procedure inlining (reduc. function call overhead)
    - Procedure reordering
    - Interprocedural dead code elimination, constant propagation and procedure reordering
  - Expected Winners
    - Many small utility functions
  - IPO can be quite expensive in terms of compilation time and disk space!

Profile Guided Optimization !

- Up to know we have only done a static analysis of the program code , PGO is a dynamic analysis. The execution-time characteristics are recorded, and this information is fed into the other optimization phases.
- Static analysis leaves many questions open like:
  - How often is x > y
  - What is the size of count
  - Which code is touched how often

```
if(x > y)
    do_this();
else
    do_that();
```

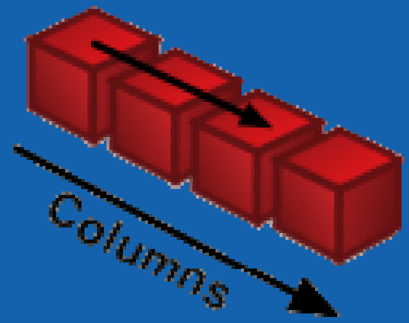
```
for(i = 0;i<count;++i)
        do_work();
```

Multi-pass Optimization - Profile Guided Optimizations (PGO)

- Uses run-time profiling to guide optimization
- Benefits code on IA-32, Intel 64 and Itanium™ architectures
  - More accurate branch prediction
  - Basic block movement to improve instruction cache behavior
  - Better decision of functions to inline (help IPO)
  - Can optimize function ordering
  - Switch-statement optimization
  - Better vectorization decisions

Agenda

- Compiler Overview
  - Intel® C++ Compiler
- High level optimization
  - IPO, PGO
- Vectorization
  - Loops and Co.



Vectorization

- For vectorization we add eight **128-bit** registers known as **XMM0-XMM7**. For the 64-bit extensions additional eight registers **XMM8-XMM15** are added
- Operations on this registers are an addition to the X86 instruction set
- The Intel® Compiler can automatically generate these instructions called SSEx (Streaming SIMD Extensions)

Evolution of SSE

1999	2000	2004	2006	2007	2008
Intel SSE	**Intel SSE2**	**Intel SSE3**	**Intel SSSE3**	**Intel SSSE4.1**	**Intel SSSE4.2**
70 instr Single-Precision Vectors Streaming operations	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr Complex Data	32 instr Decode	47 instr Video Graphics building blocks Advanced vector instr	8 instr String/XML processing POP-Count CRC

Will be continued by

- Intel® AES (Cryptographie, Westmere Architecture 2009)
- Intel® AVX (256 bit SSE, Sandy Bridge Architecture, 2010)

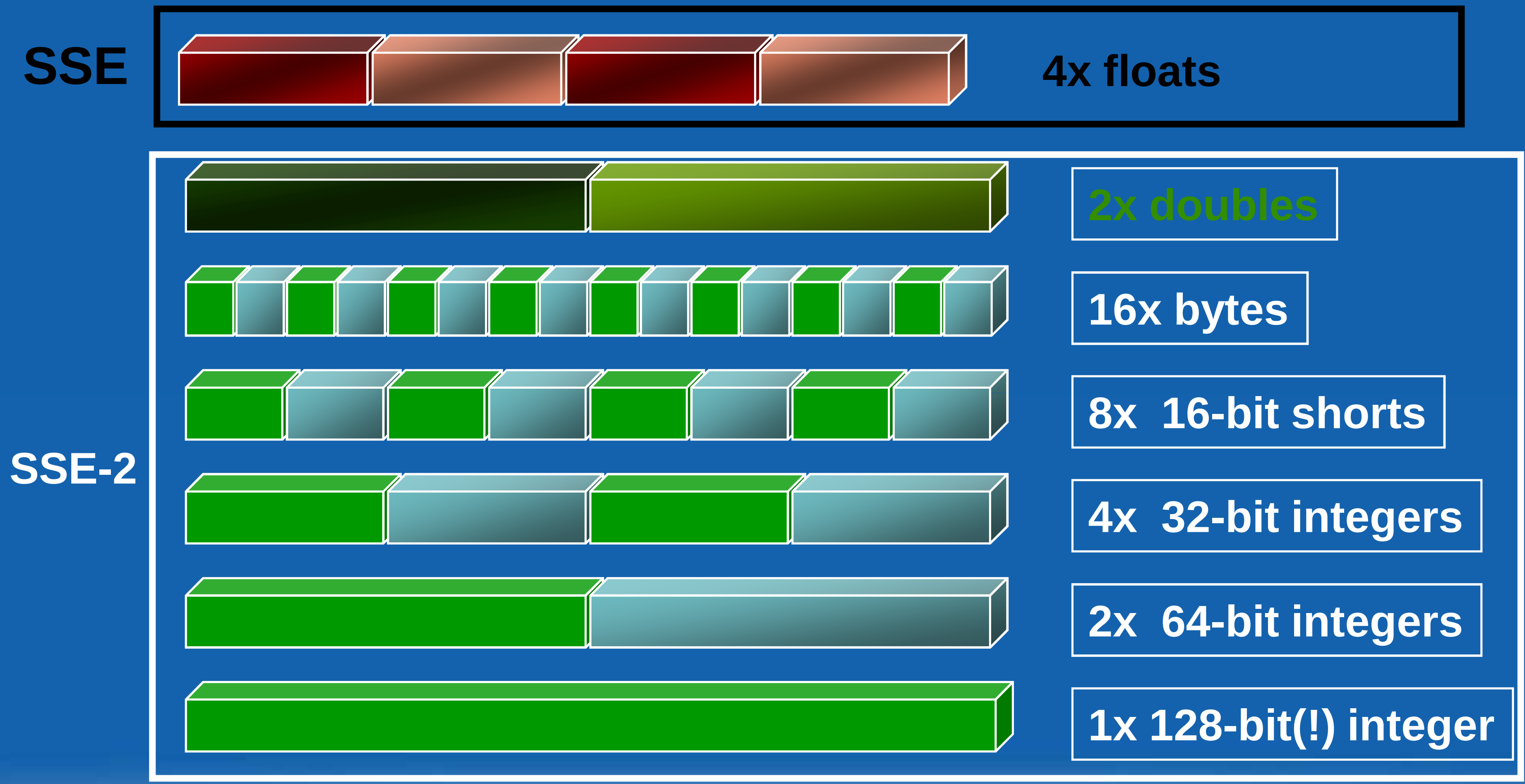
Compiler Based Vectorization

/Qx<code1>[,<code2>,...] –x<code1> …

– generate specialized code to run exclusively on processors indicated by <code>

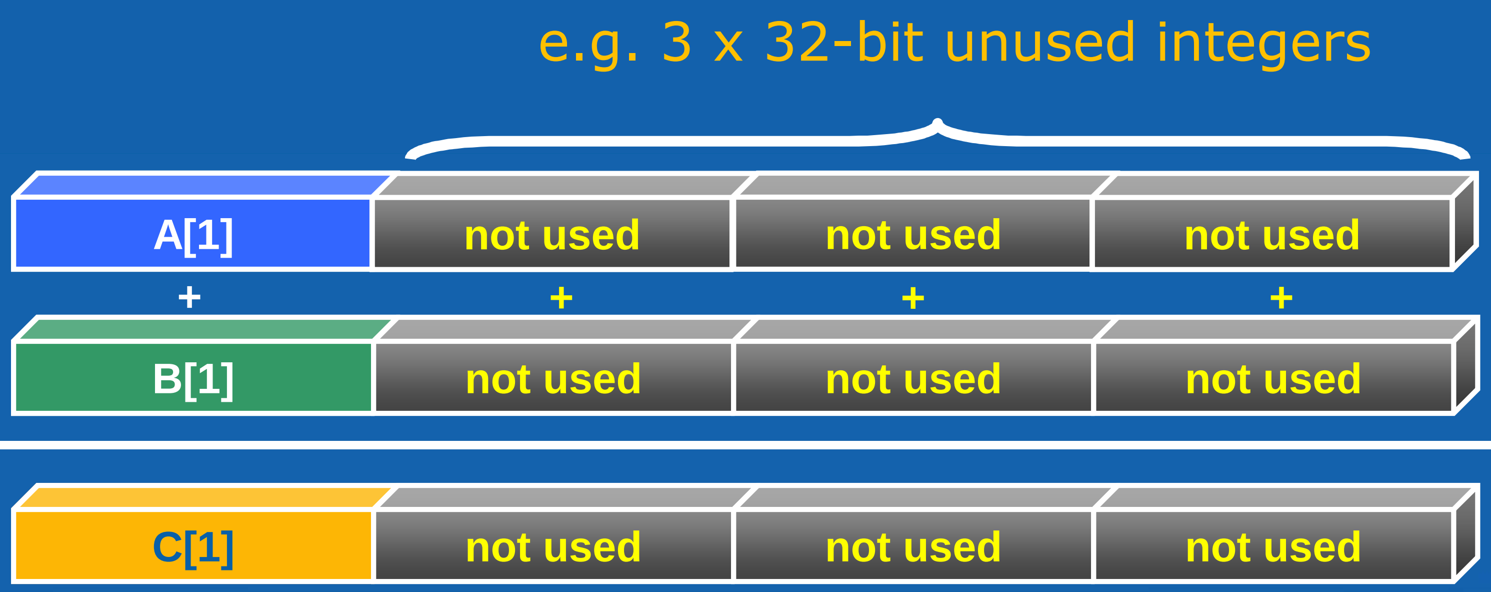
SSE2	Intel® Pentium 4 and compatible Intel processors
SSE3	Intel® Core™ processor family with Streaming SIMD Extensions 3
SSSE3	Intel® Core™2 processor family with SSSE3
SSE4.1	Intel® processors supporting SSE4 Vectorizing Compiler and Media Accelerator instructions
SSE4.2	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors

SIMD – SSE, SSE2, SSE3 Support



Using SSE3 – How to convert ?

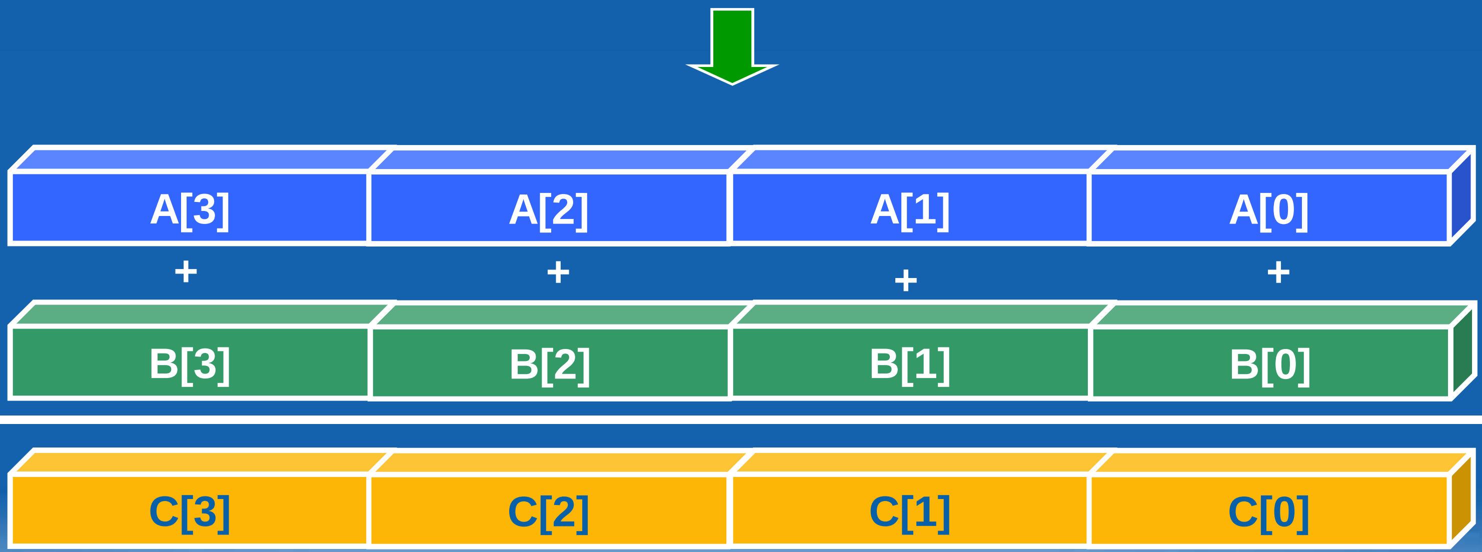
```
for (i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```



...into This...

- Processor switch is vectorizing loops for fp and scalar ops.
- Usage: Linux* -xSSE3 Windows* -QxSSE3

```
for (i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```



Vectorization

Problems and what you do …

Why Loops Don't Vectorize

- *Independence*
  - Loop Iterations generally must be independent
- *Some* relevant qualifiers:
  - Some dependent loops can be vectorized.
  - Most function calls cannot be vectorized.
  - Some conditional branches prevent vectorization.
  - Loops must be countable.
  - Outer loop of nest cannot be vectorized.
  - Mixed data types cannot be vectorized.

Why Loops Don't Vectorize

- "Existence of vector dependence"
- "Vectorization possible but seems inefficient"
- "Operator unsuited for vectorization"
- "Nonunit stride used"
- "Mixed Data Types"
- "Subscript too complex"
- "Condition too Complex"
- "Condition may protect exception"
- "Low trip count"
- "Unsupported Loop Structure"
- "Not Inner Loop"
- "Contains unvectorizable statement at line XX"

Agenda

- Compiler Overview
  - Intel® C++ Compiler
- High level optimization
  - IPO, PGO
- Vectorization
  - Loops and Co.
- Compiler Reports
  - Effective use

Optimization Report Methodology

- During "Analyzing the Data" phase of an Optimization Methodology
  - Generate the Report(s)
  - Filter The Data in the reports
    - "Hot Spot"
    - Relevant Optimization Report
  - Analyze the decisions made by the compiler
- Add assertions using Application knowledge to help the compiler
  - Modify Compiler Switch Settings (assertion flags)
  - Modify Source (pragma's or modify source)
  - File Optimization Feature Request to Intel Compiler Team

Filter The Data in the Reports

- Choose only specific phases relevant to what you are looking for

  -opt-report-phase [phase]

  Enables the report for only the selected phases

  hlo, ipo_inl, ecg_swp

- Which loops vectorized/parallelized ?

  -vec-report[0..3..5]

  -par-report[0..3]

- Views the data for a particular function

  -opt-report-routine functionname

Why Didn't *My* Loop Vectorize?

- Use Report switch.
- Syntax: **`-Qvec-reportn`**
- Sets diagnostic level dumped to stdout
- n=*0*: No diagnostic information
- n=1: **(Default)** Loops successfully vectorized
- n=*2*: Loops not vectorized – and the reason why not
- n=*3*: Adds dependency Information

Example: Vectorization Report

```
> C:\home\src\classes\compiler\MatVector>icl /QxW /Qvec-report3 Driver.c Multiply.c
      /FeMatVector.exe
Driver.c
Driver.c(64) : (col. 2) remark: loop was not vectorized: contains unvectorizable
      statement at line 65.
Driver.c(74) : (col. 2) remark: LOOP WAS VECTORIZED.
Driver.c(39) : (col. 2) remark: LOOP WAS VECTORIZED.
Driver.c(28) : (col. 10) remark: loop was not vectorized: statement cannot be
      vectorized.
Driver.c(30) : (col. 3) remark: LOOP WAS VECTORIZED.
Driver.c(12) : (col. 2) remark: loop was not vectorized: not inner loop.
Driver.c(14) : (col. 14) remark: loop was not vectorized: statement cannot be
      vectorized.
Driver.c(18) : (col. 3) remark: loop was not vectorized: not inner loop.
Driver.c(19) : (col. 4) remark: LOOP WAS VECTORIZED.
Multiply.c
Multiply.c(7) : (col. 2) remark: loop was not vectorized: not inner loop.
Multiply.c(9) : (col. 3) remark: loop was not vectorized: unsupported loop structure.
-out:MatVector.exe
Driver.obj
Multiply.obj
```

Agenda for today

10.00 – 11.30 Intel® Compiler

11.00 – 11.30 Intel® Threading Tools

11.30 – 12.00 Coffee Break

12.00 – 12.45 Intel® Vtune & Cluster Tools

Presenter: Mario Deilmann

eMail: mario.deilmann@intel.com

Checking Correctness and Performance of Parallel Programs

Intel® Thread Checker
Intel® Thread Profiler

Threading Development Cycle

Analysis

- –Intel® VTune™ Performance Analyzer

Design (Introduce Threads)

- –Intel® Performance libraries: IPP and MKL
- –OpenMP* (Intel® Compiler)
- –Intel® Threading Building Blocks

Debug for correctness

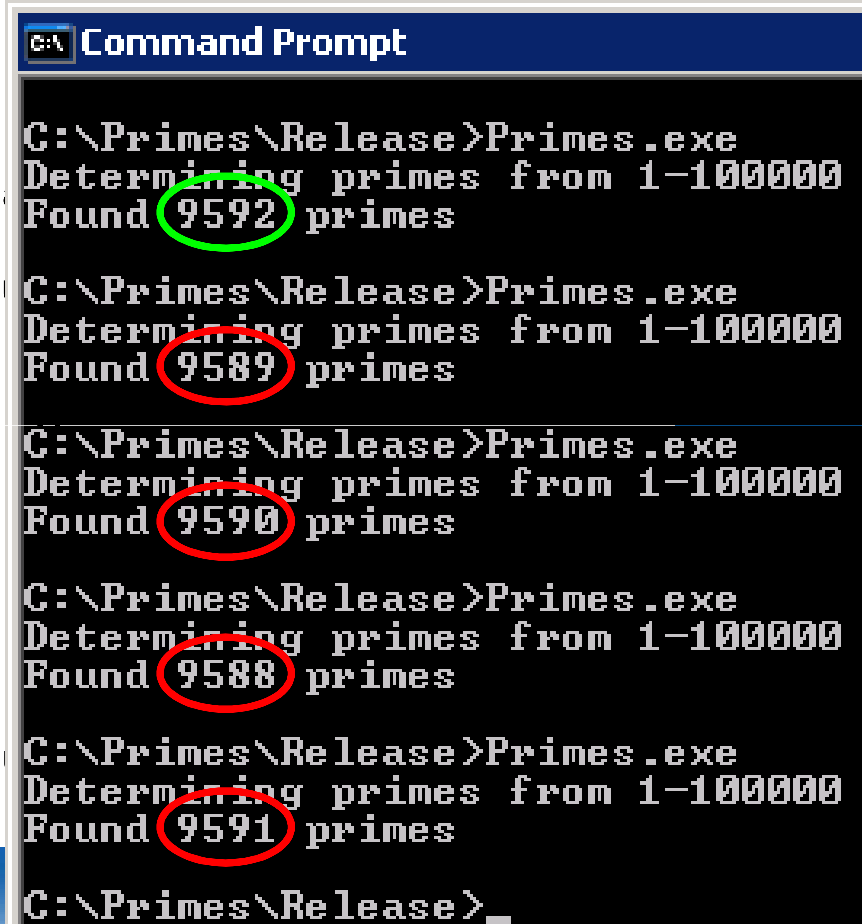
- –Intel® Thread Checker
- –Intel® Debugger

Tune for performance

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

Example: Not Quite Right

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



Intel® Thread Checker

- Debugging tool for threaded software
- Finds threading bugs (data races, dead locks) in Windows*, POSIX*, OpenMP*, and Intel® Threading Building Blocks threaded software
- API for user-defined synchronization primitives
- Locates bugs quickly that can take days to find using traditional methods and tools
  - Isolates problems, not the symptoms
  - Bug does not have to occur to find it!

Thread Checker: Analysis

- Dynamic as software runs
  - Data (workload) -driven execution
- Includes monitoring of:
  - Thread and Sync APIs used
  - Thread execution order
    - Scheduler impacts results
  - Memory accesses between threads

Code path must be executed to be analyzed

Multithreading introduces new problems

- New class of problems are introduced due to the interaction between threads which are complicated, non-deterministic and therefore hard to find !

- Correctness problems (data races)
- Performance problems (contention)
- Runtime problems

Race Conditions

- Execution order is assumed but cannot be guaranteed
  - Concurrent access of same variable by multiple threads
- Most common error in multithreaded programs
- May not be apparent at all times

Prominent problem: Race Condition

- Suppose Global Variables
  - A=1, B=2
- End Result different if:
  - T1 runs before T2
  - T2 runs before T1
- Execution order is not guaranteed unless synchronization methods are used.

Thread1
x = a + b

Thread2
b = 42

Deadlock Example

Thread1
Waiting lockB to be released

```
Func1()
{
        Lock(A)
                globalX++;
                Lock(B)
                globalY++;
                unlock(B);
        unlock(A);
}
```

Thread2
Waiting lockA to be released

```
Func2()
{
        Lock(B)
                globalY++;
                Lock(A)
                globalX++;
                unlock(A);
        unlock(B);
}
```

Deadlock - Both threads are now waiting for each other eternally

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

Intel® Thread Checker Summary

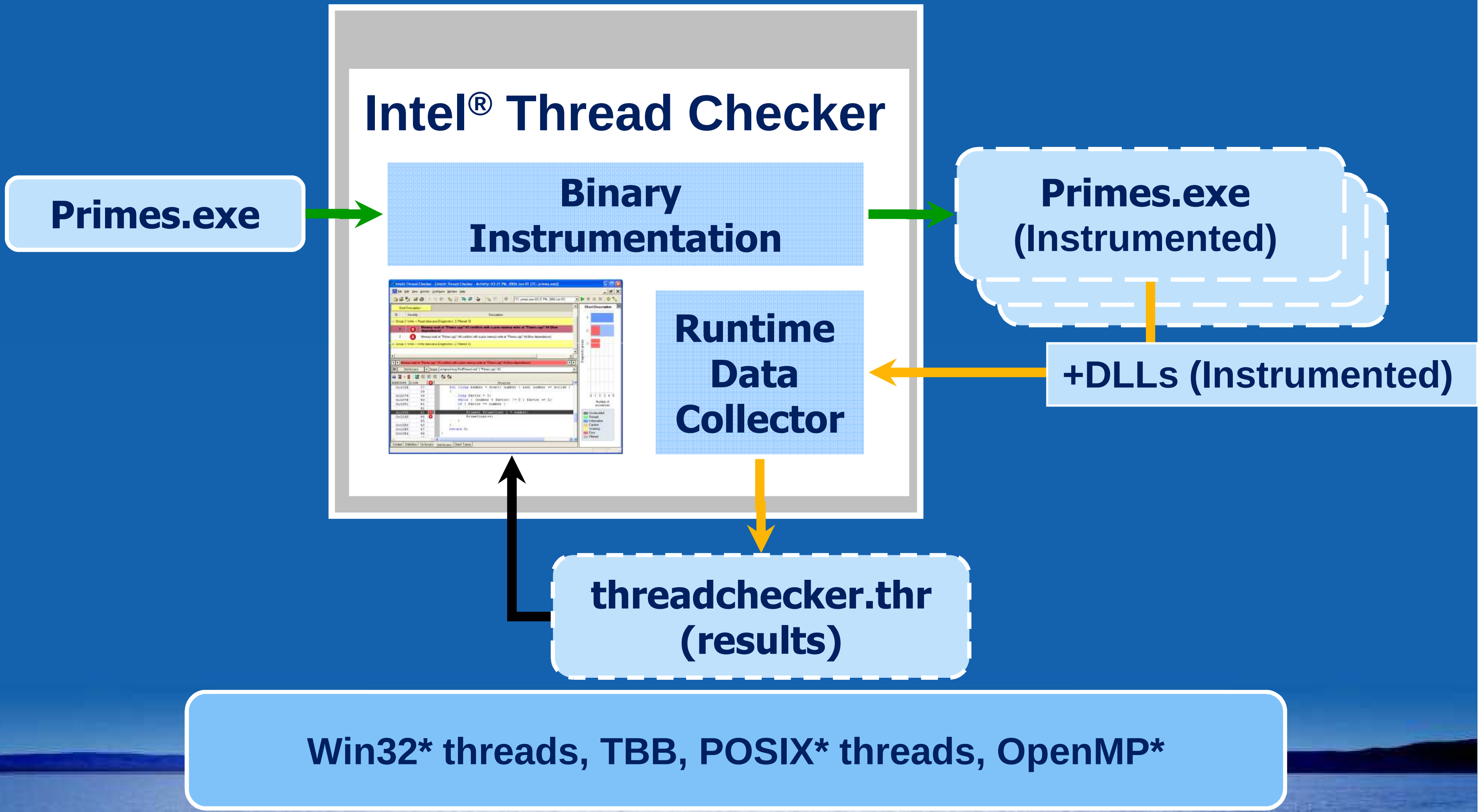
- Threading errors are easy to introduce
- Debugging these errors by traditional techniques is hard
- Intel® Thread Checker catches these errors
  - Errors do not have to occur to be detected
  - Greatly reduces debugging time
  - Improves robustness of the application

Thread Checker: Before You Start

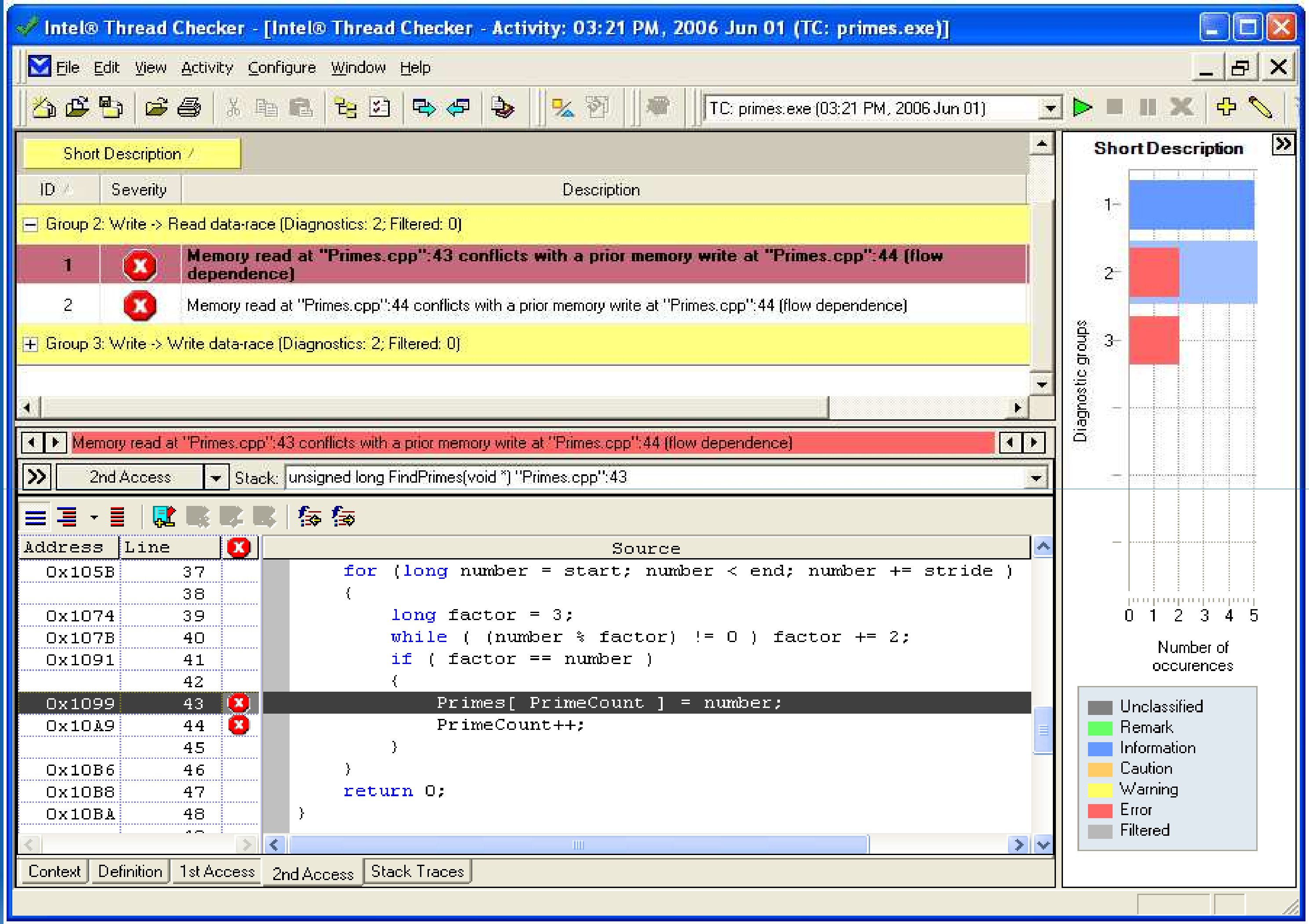
- Instrumentation: background
  - Adds calls to library to record information
    - Thread and Sync APIs
    - Memory accesses
  - Increases execution ***time*** and ***size***
- Use ***small*** data sets (workloads)
  - Execution time and space is **expanded**
  - Multiple runs over different paths yield best results

Workload selection is important!

Intel® Thread Checker
Data Flow of Binary Instrumentation

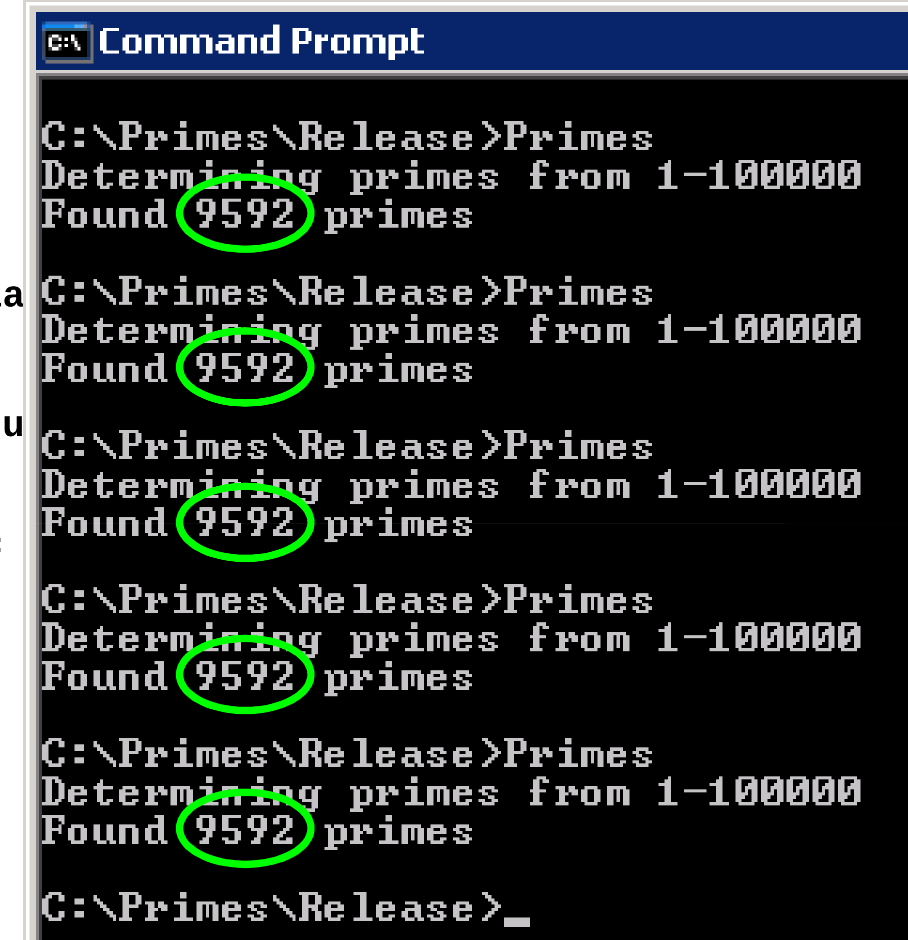


Graphical User Interface (Windows)

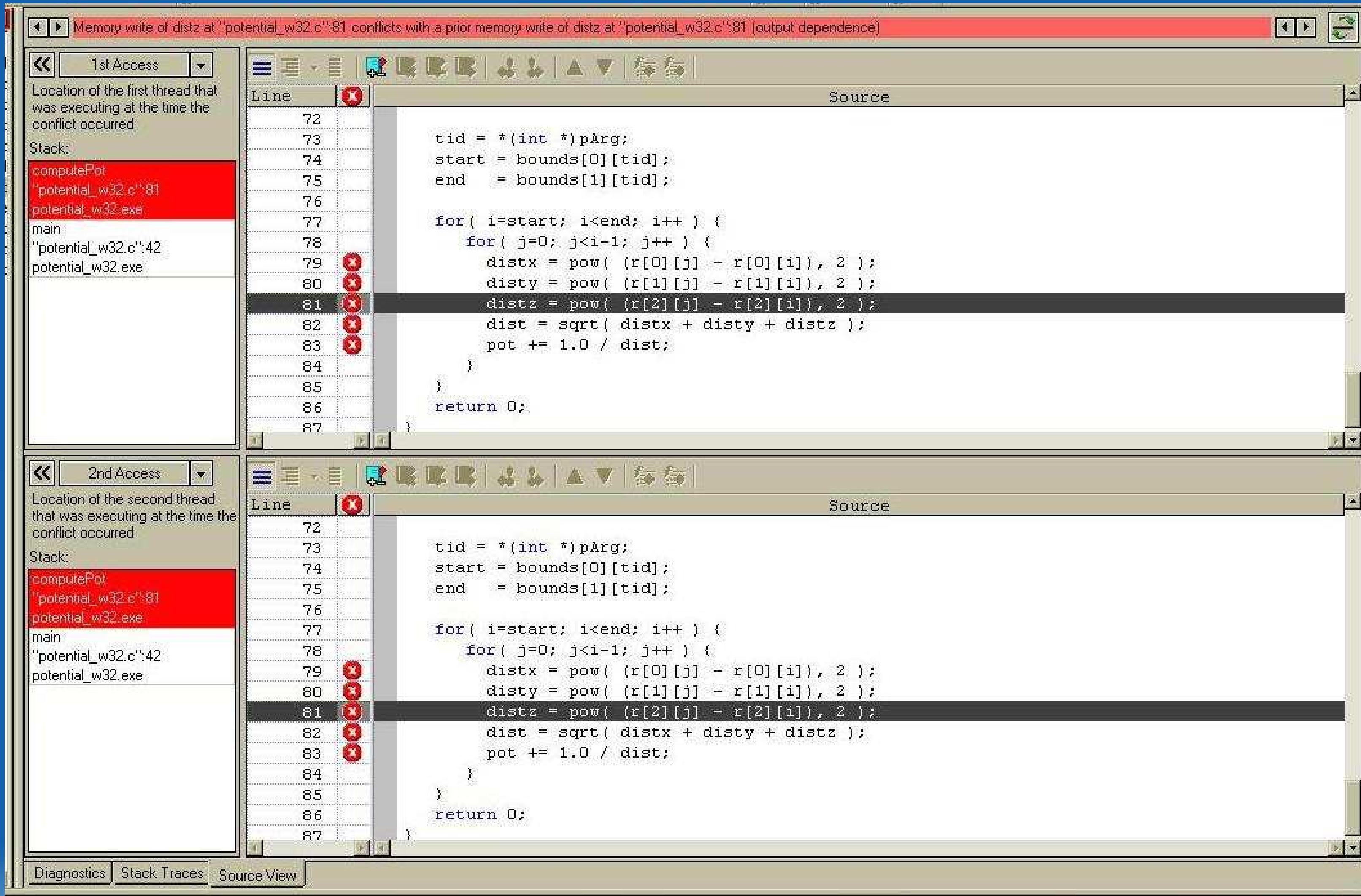


Example: Much Better Now …

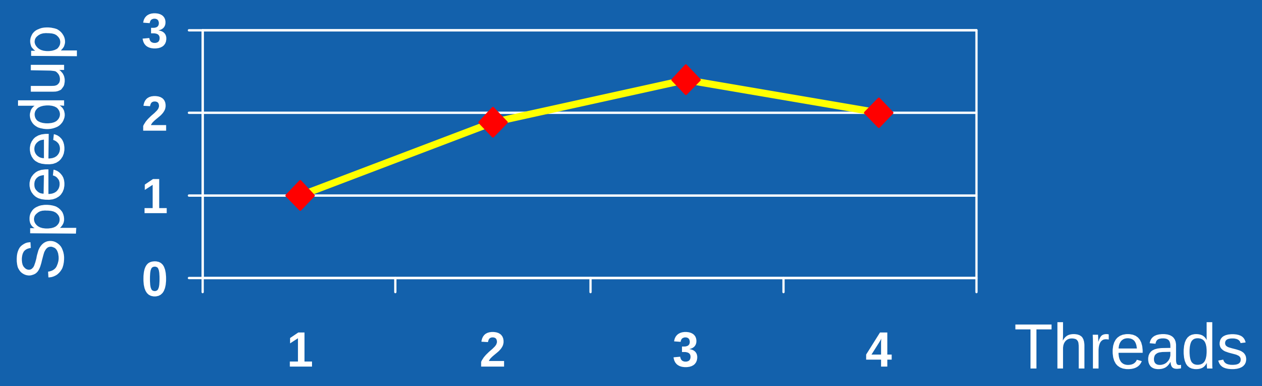
```
#include <stdio.h>
const long N = 100000;
long Primes[N], PrimesCount = 0;
main() {
  printf( "Determining primes from 1-%d
  Primes[ PrimesCount++ ] = 2; // specia
  #pragma omp parallel for
  for ( long number = 3; number <= N; nu
    long factor = 3;
    while ( number % factor ) factor +=
    if ( factor == number )
    #pragma omp critical
    {
      Primes[ PrimesCount ] = number;
      PrimesCount++;
    }
…
```



Source Code Viewer



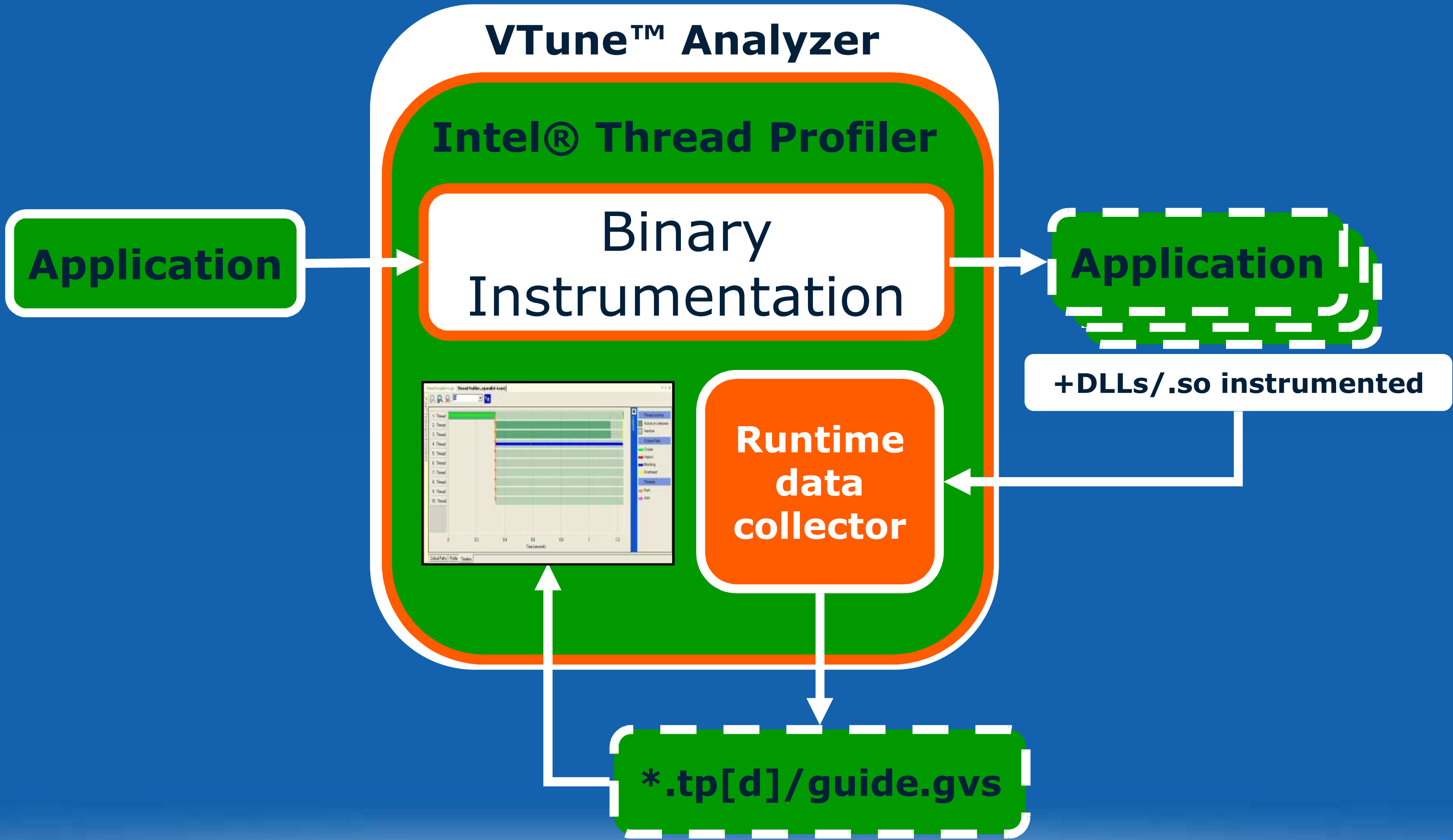
Performance Profile: Recap



Possible causes for this scalability profile:

1. Insufficient parallel work
2. Memory bandwidth limitations
3. Synchronization overhead
4. Load imbalance

Thread Profiler Phases



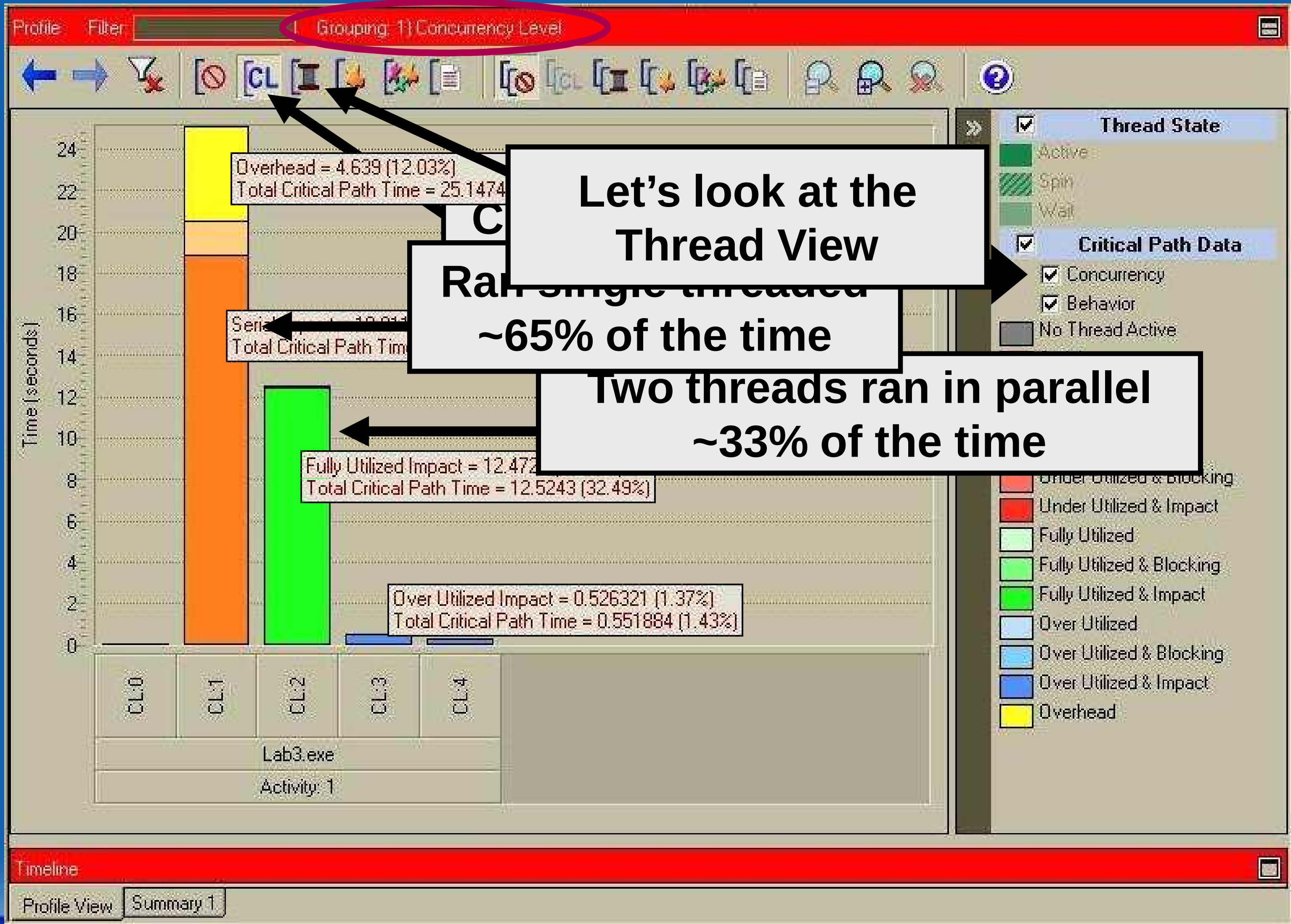
Binary Instrumentation

- Lower run-time overhead as only select events are monitored
- Usually performance within 2X of original performance for applications with reasonable synchronization
- Events recorded
  - Create Thread, Thread Entry, Wait for Synch. Object or Event, Acquire Synch. Object or Event, Release or Signal synch. Object or Event, Wait for external event, Receive external event, Thread Exit

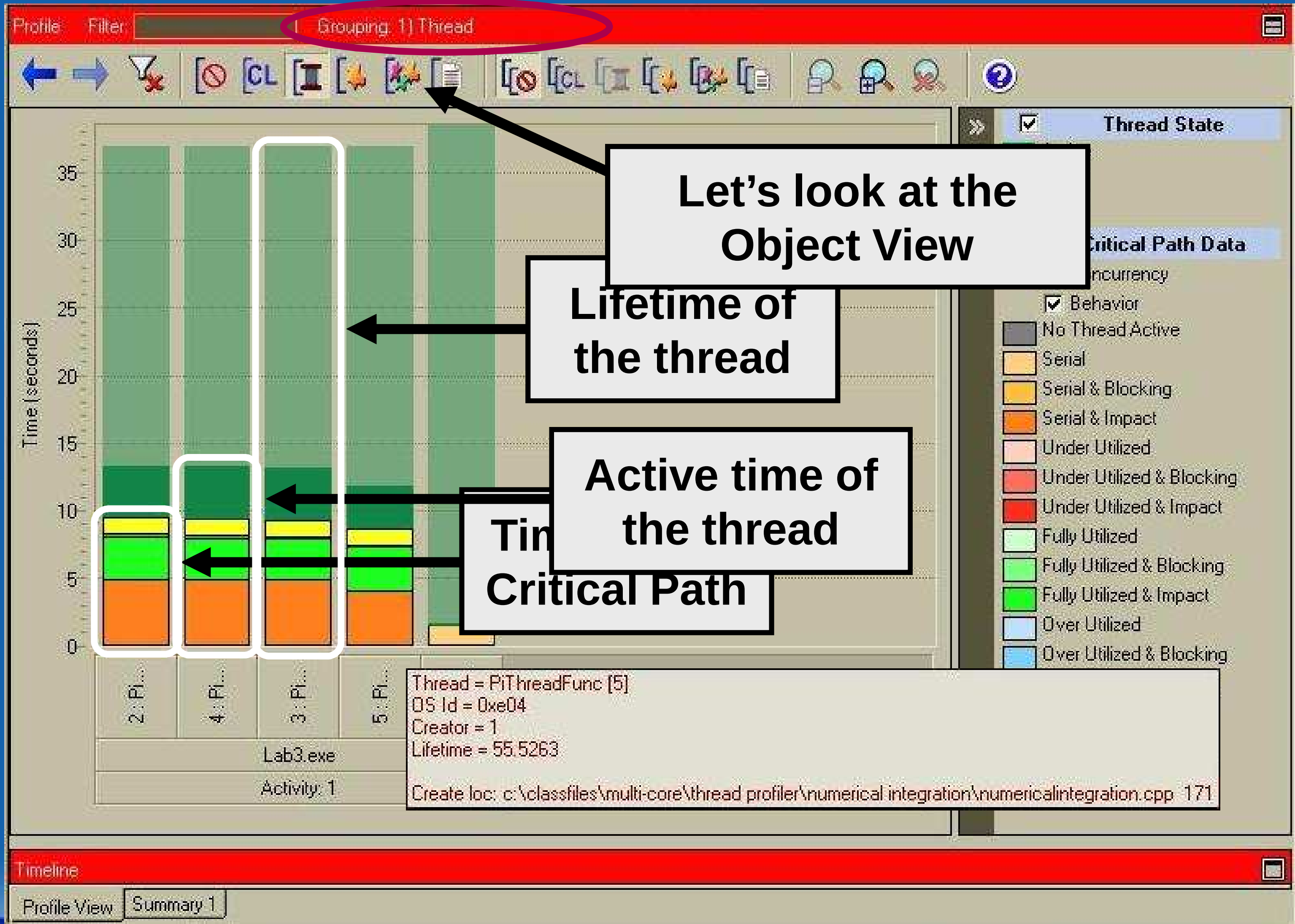
System APIs Monitored

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

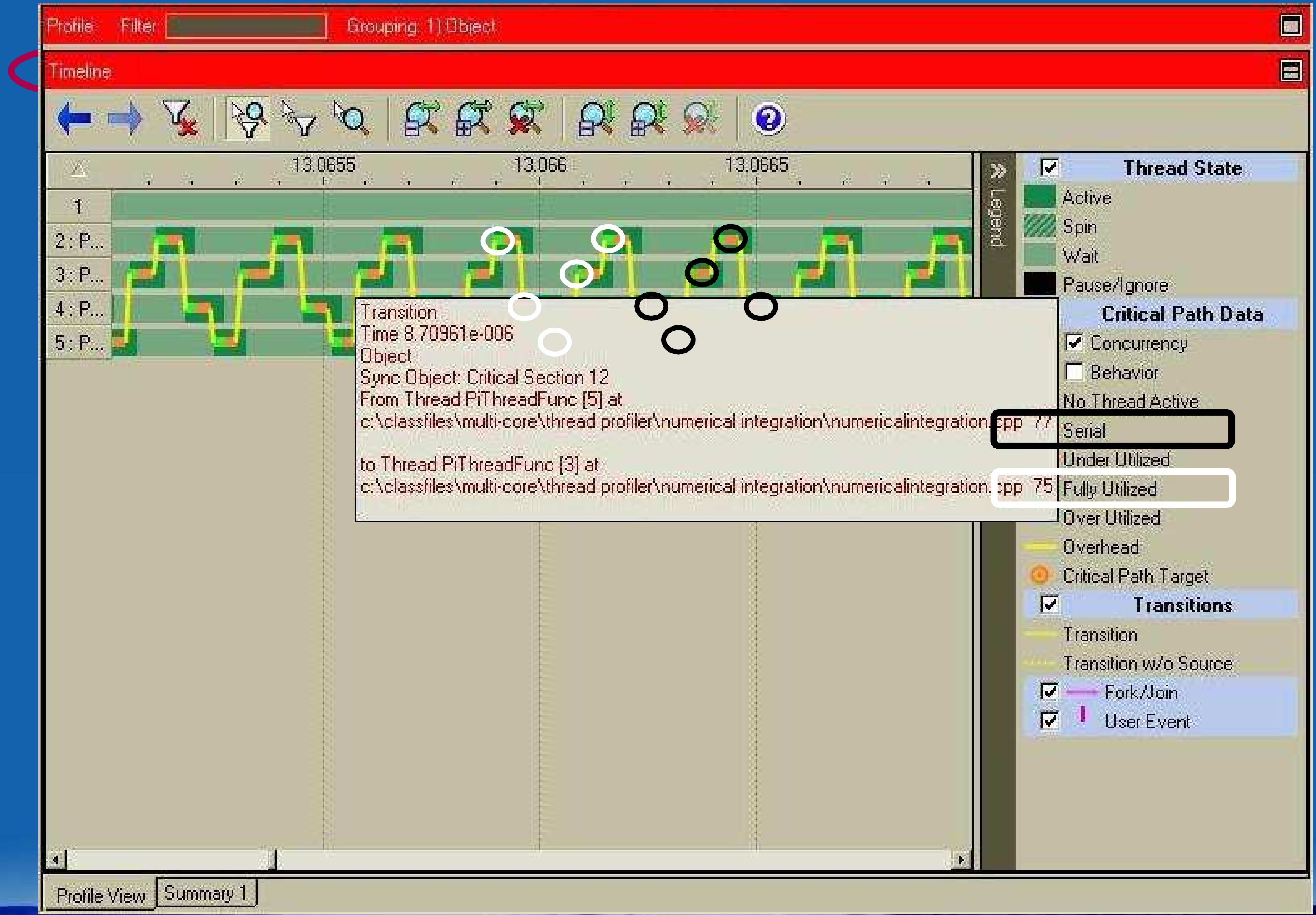
Profile Pane – Concurrency Level View



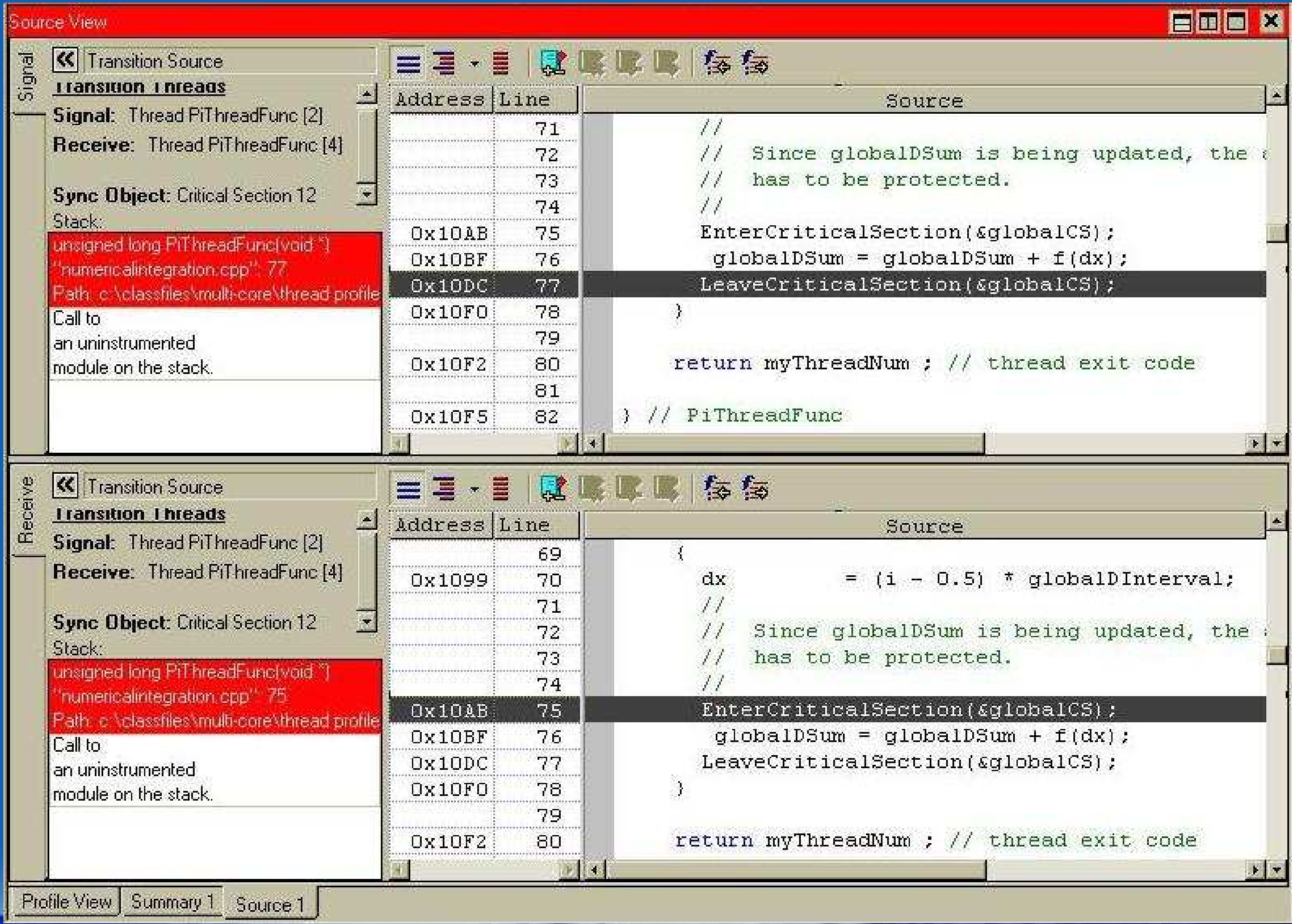
Profile Pane – Thread View



Timeline Pane



Source View



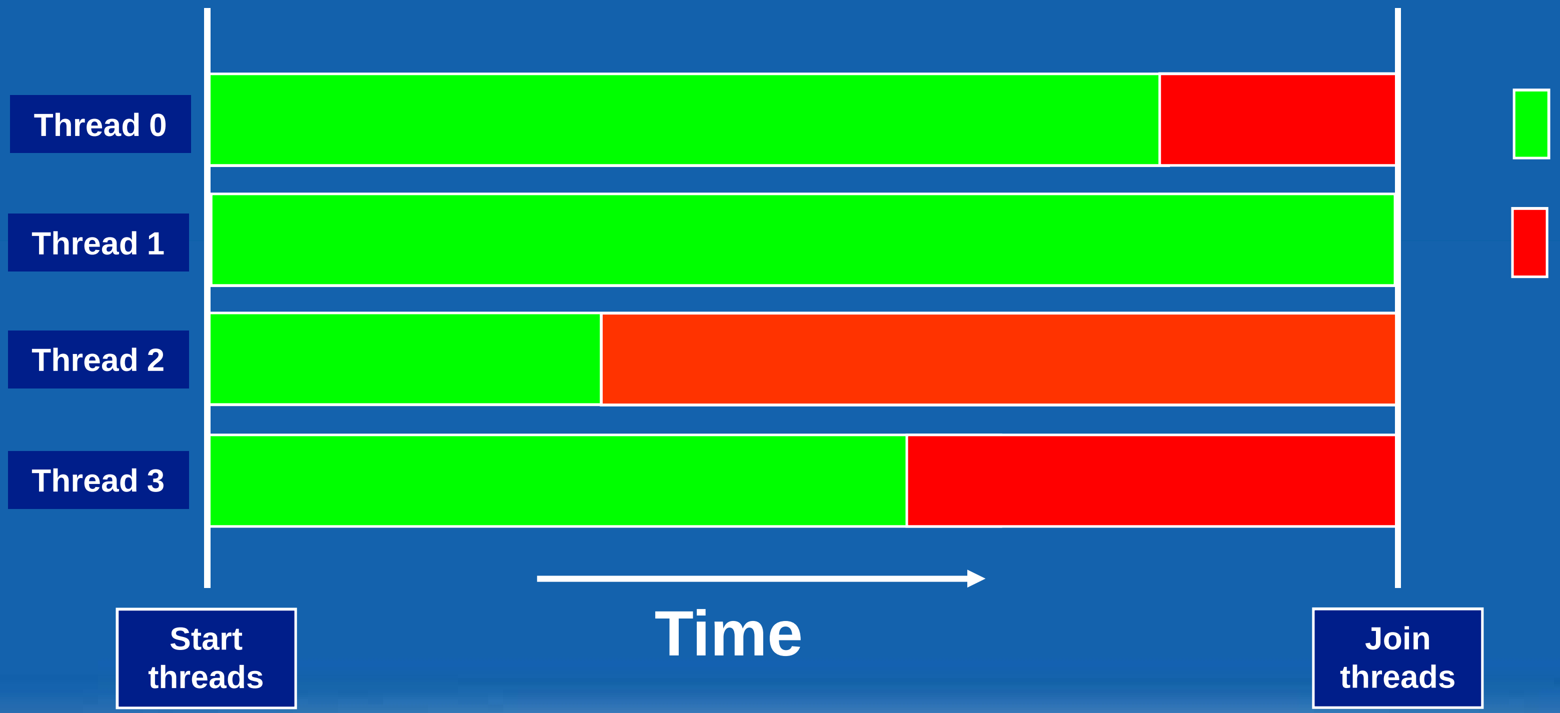
Source View
Signal
Transition Source
Transition Threads
Signal: Thread PiThreadFunc [2]
Receive: Thread PiThreadFunc [4]
Sync Object: Critical Section 12
Stack:
unsigned long PiThreadFunc(void *)
"numericalintegration.cpp": 77
Path: c:\classfiles\multi-core\thread profile
Call to
an uninstrumented
module on the stack.
Address Line Source
71 //
72 // Since globalDSum is being updated, the
73 // has to be protected.
74 //
0x10AB 75 EnterCriticalSection(&globalCS);
0x10BF 76 globalDSum = globalDSum + f(dx);
0x10DC 77 LeaveCriticalSection(&globalCS);
0x10F0 78 }
79
0x10F2 80 return myThreadNum ; // thread exit code
81
0x10F5 82 } // PiThreadFunc
Receive
Transition Source
Transition Threads
Signal: Thread PiThreadFunc [2]
Receive: Thread PiThreadFunc [4]
Sync Object: Critical Section 12
Stack:
unsigned long PiThreadFunc(void *)
"numericalintegration.cpp": 75
Path: c:\classfiles\multi-core\thread profile
Call to
an uninstrumented
module on the stack.
Address Line Source
69 {
0x1099 70 dx = (i - 0.5) * globalDInterval;
71 //
72 // Since globalDSum is being updated, the
73 // has to be protected.
74 //
0x10AB 75 EnterCriticalSection(&globalCS);
0x10BF 76 globalDSum = globalDSum + f(dx);
0x10DC 77 LeaveCriticalSection(&globalCS);
0x10F0 78 }
79
0x10F2 80 return myThreadNum ; // thread exit code
Profile View
Summary 1
Source 1

Common Performance Issues

- Load balance
  - Improper distribution of parallel work
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- Parallel Overhead
  - Due to thread creation, scheduling..
- Granularity
  - No sufficient parallel work

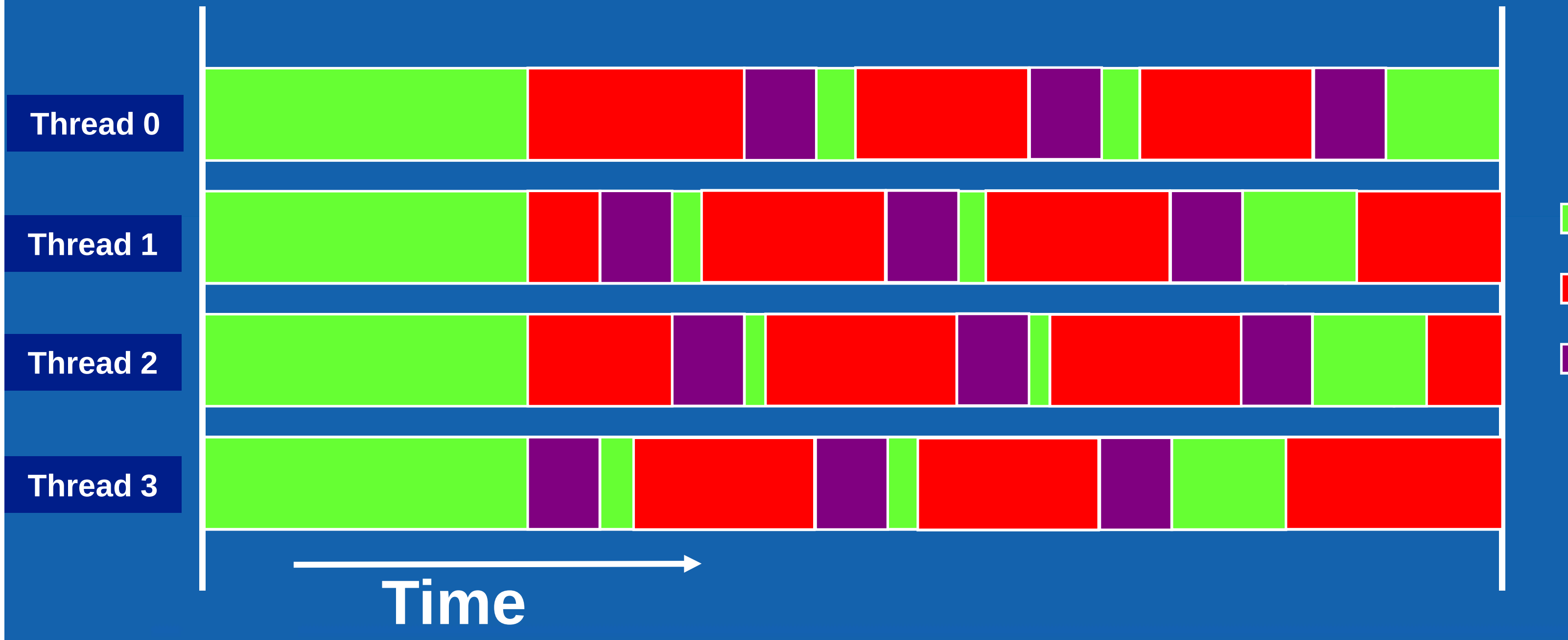
Load Imbalance

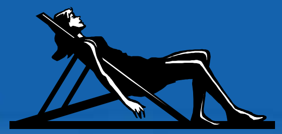
- Unequal work loads lead to idle threads and wasted time



Synchronization

- By definition, synchronization serializes execution
- Lock contention means more idle time for threads





Synchronization Fixes

- Eliminate synchronization
  - Expensive but necessary “evil”
  - Use storage local to threads
    - Use local variable for partial results, update global after local computations
    - Allocate space on thread stack (**`alloca`**)
    - Use thread-local storage API (TlsAlloc)
  - Use atomic updates whenever possible
    - Some global data updates can use atomic operations (Interlocked API family)

Synchronization Fixes

- Use best synchronization object for job
  - Critical Section
    - Local object
    - Available to threads within the same process
    - Lower overhead (~8X faster than mutex)
  - Mutex
    - Kernel object
    - Accessible to threads within different processes
    - Deadlock safety (can only be released by owner)
- Other objects are available

What's Been Covered

- Identifying performance issues can be time consuming without tools
- Tools are required to understand and to optimize parallel efficiency and hardware utilization
- Thread Profiler helps you understand your applications thread activity, system utilization, and scaling performance

www.intel.com/software/products