Intel Software Tools

Stephen Blair-Chappell
Intel Compiler Labs

This training is based on the following…

Parallel Programming with Parallel Studio XE

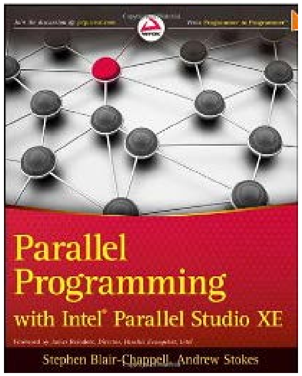
Stephen Blair-Chappell & Andrew Stokes

Wiley ISBN: *9780470891650*

Part I: Introduction
1: Parallelism Today
2: An Overview of Parallel Studio XE
3: Parallel Studio XE for the Impatient

Part II: Using Parallel Studio XE
4: Producing Optimized Code
5: Writing Secure Code
6: Where to Parallelize
7: Implementing Parallelism
8: Checking for Errors
9: Tuning Parallelism
10: Advisor-Driven Design
11: Debugging Parallel Applications
12:Event-Based Analysis with VTune Amplifier XE

Part III :Case Studies
13: The World's First Sudoku 'Thirty-Niner'
14: Nine Tips to Parallel Heaven
15: Parallel Track-Fitting in the CERN Collider
16: Parallelizing Legacy Code



Speedup using parallelism

Parallel Code

Chapter 6
Where to Parallelize

1 **Analyze**

- Amplifier XE
- Hotspot
- EBS (XE only)

Chapter 7
Implementing
Parallelism

2 **Implement**

- Composer XE
  - Compiler
  - Cilk Plus
  - OpenMP
  - Libraries
  - MKL
  - TBB
  - IPP

Chapter 8
Checking for Errors

3 **Debug**

- Inspector XE
- Threads
- Memory

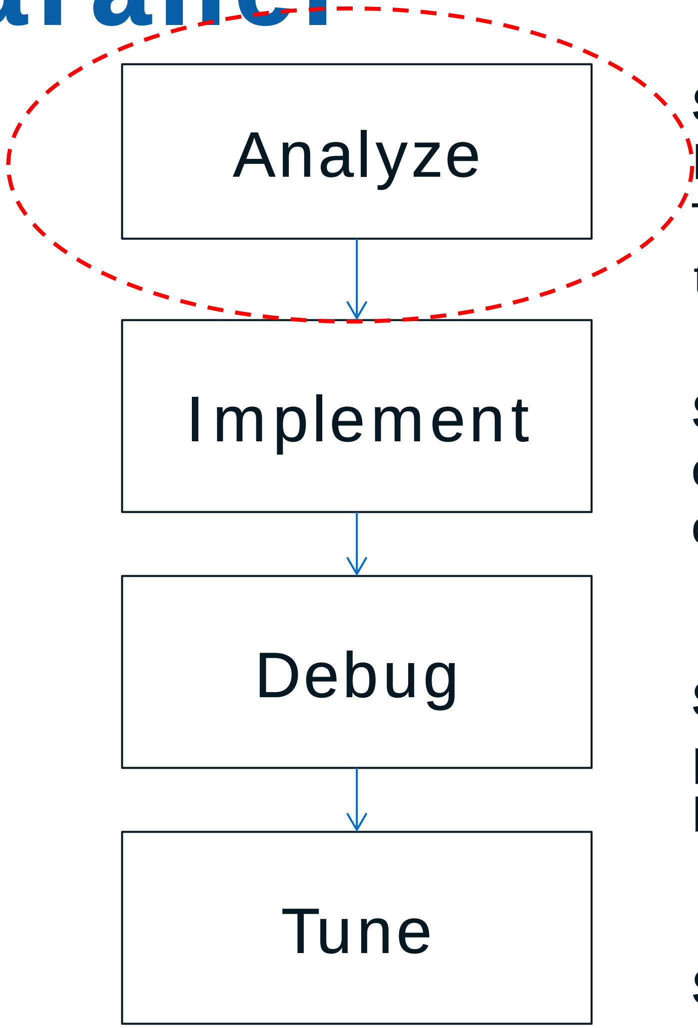
Chapter 9
Tuning Parallelism

4 **Tune**

- Amplifier XE
- concurrency
- Locks & waits

Four Step Development

Steps in moving from Serial to Parallel



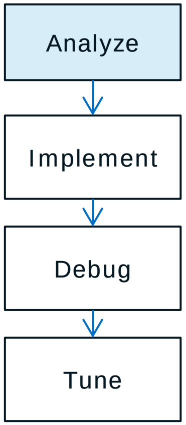
Step 1 : Look for hotspots in application. These are best candidates to make parallel

Step 2: Add parallel constructs into source code

Step 3: Check if any parallel-type errors have been introduced

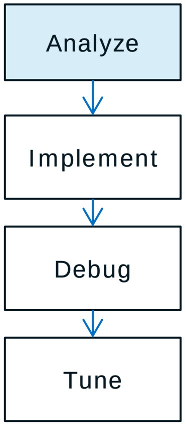
Step 4: Tune the parallel application.

Key Questions - Analyze



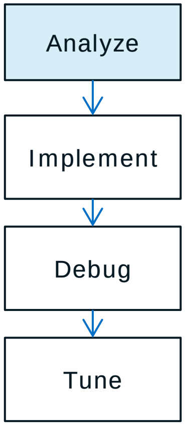
- Is my program parallel?
- Where is the best place to parallelise my program?
- How can I get my program to run faster?
- What's the expected speedup?

Four Different Ways to Find the Hotspots



1. Using Intel compiler's loop profiler & profile viewer
2. Using the compiler's Auto-parallelizer
3. Performing a Survey with Advisor
4. Using Amplifier XE

Using the loop profiler



1 Compile to add instrumentation

```
icl /Qprofile-functions
/Qprofile-loops:all prog.c
```

Run the Program 2

exe

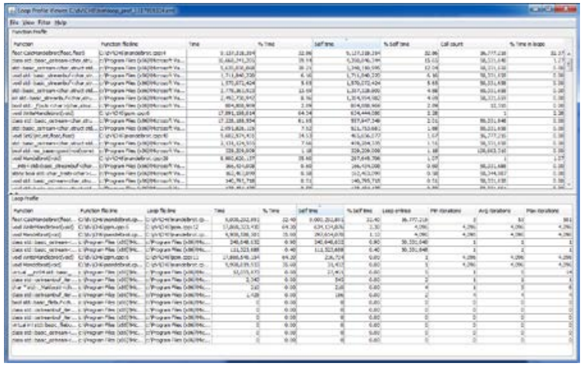
```
prog.exe
```

Function profile

Loop Profile

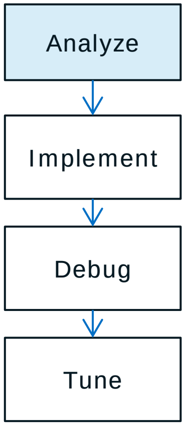
3 View the Results

```
Loopprofileviewer <filename>
```



Optimization Notice

Using the Auto-Parallelizer



1 Compile enabling auto parallelism and reports

`icl /O2 /Qipo /Qparallel /Qpar-report2 prog.c`

Look at the results 2

Results
(20) remark: LOOP WAS AUTO-PARALLELIZED
(67) remark: loop was not parallelized: existence of parallel dependence*(see note)

** Note: You can also look at the loops that could not be parallelized. It might be worth fixing the problem reported. Add the option /Qguide may give you extra information.*

3 Add your own parallel code where auto-vectorizer was successful

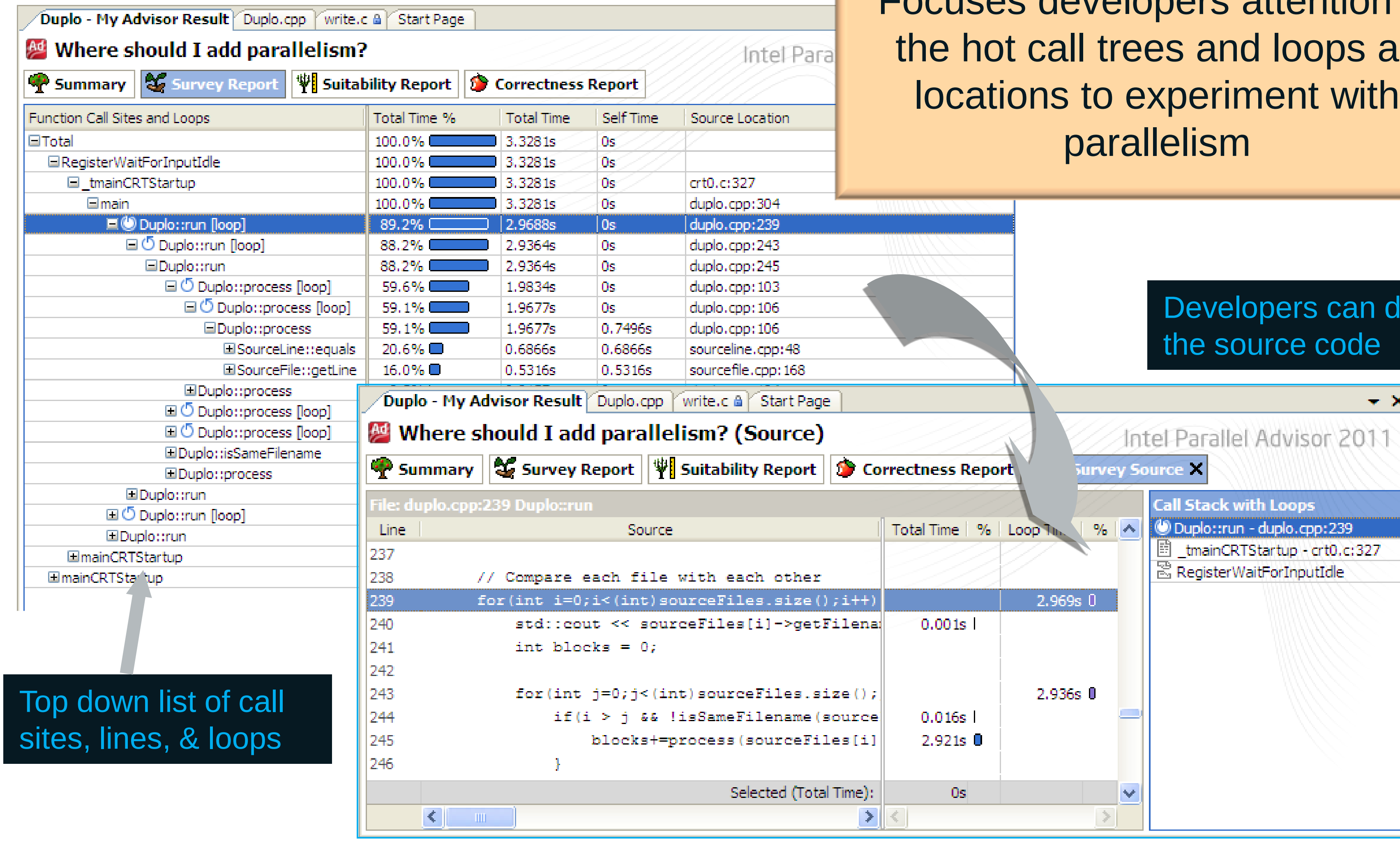
```
20: cilk_for(int=0;i<100;i++)
21:{
22:    / etc
.
45:}
```

4 Rebuild without auto parallelism

`icl /O2 /Qipo prog.c`

Intel Parallel Advisor – Survey Target

Focuses developers attention to the hot call trees and loops as locations to experiment with parallelism

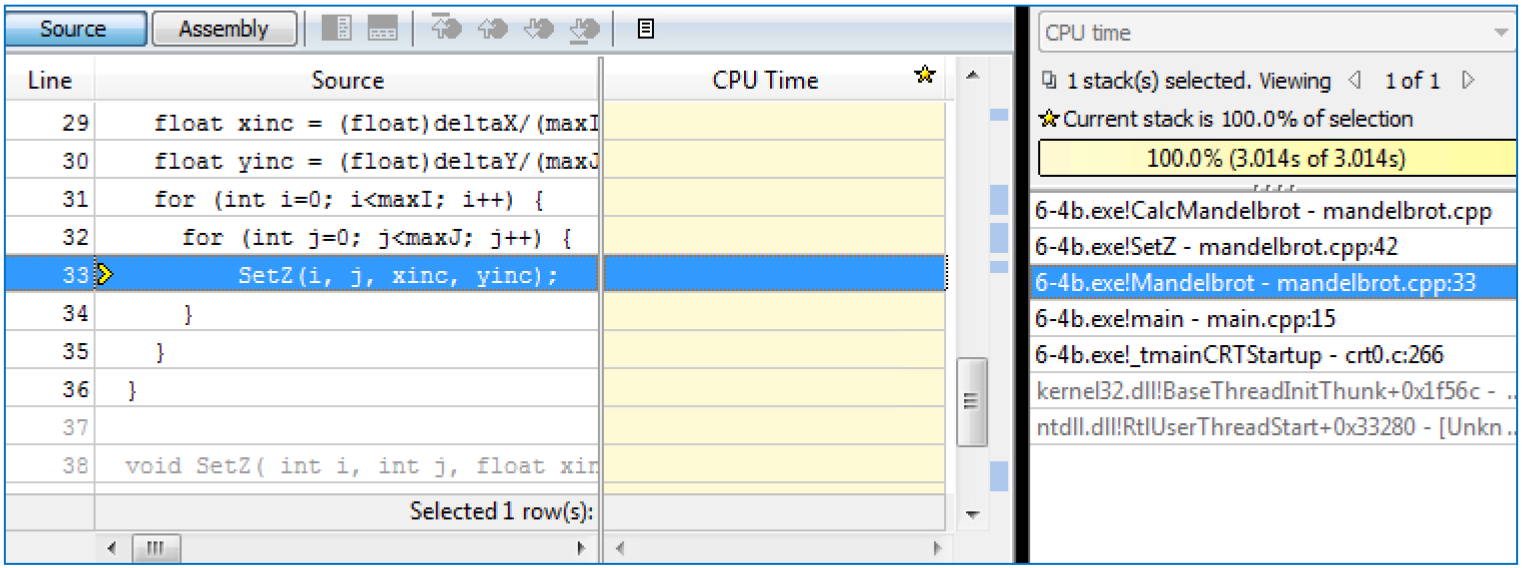


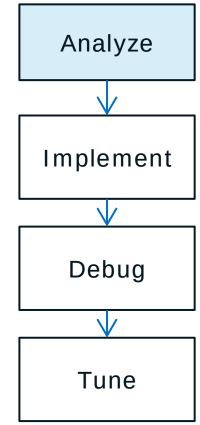
Developers can drill into the source code

Top down list of call sites, lines, & loops

Optimization Notice

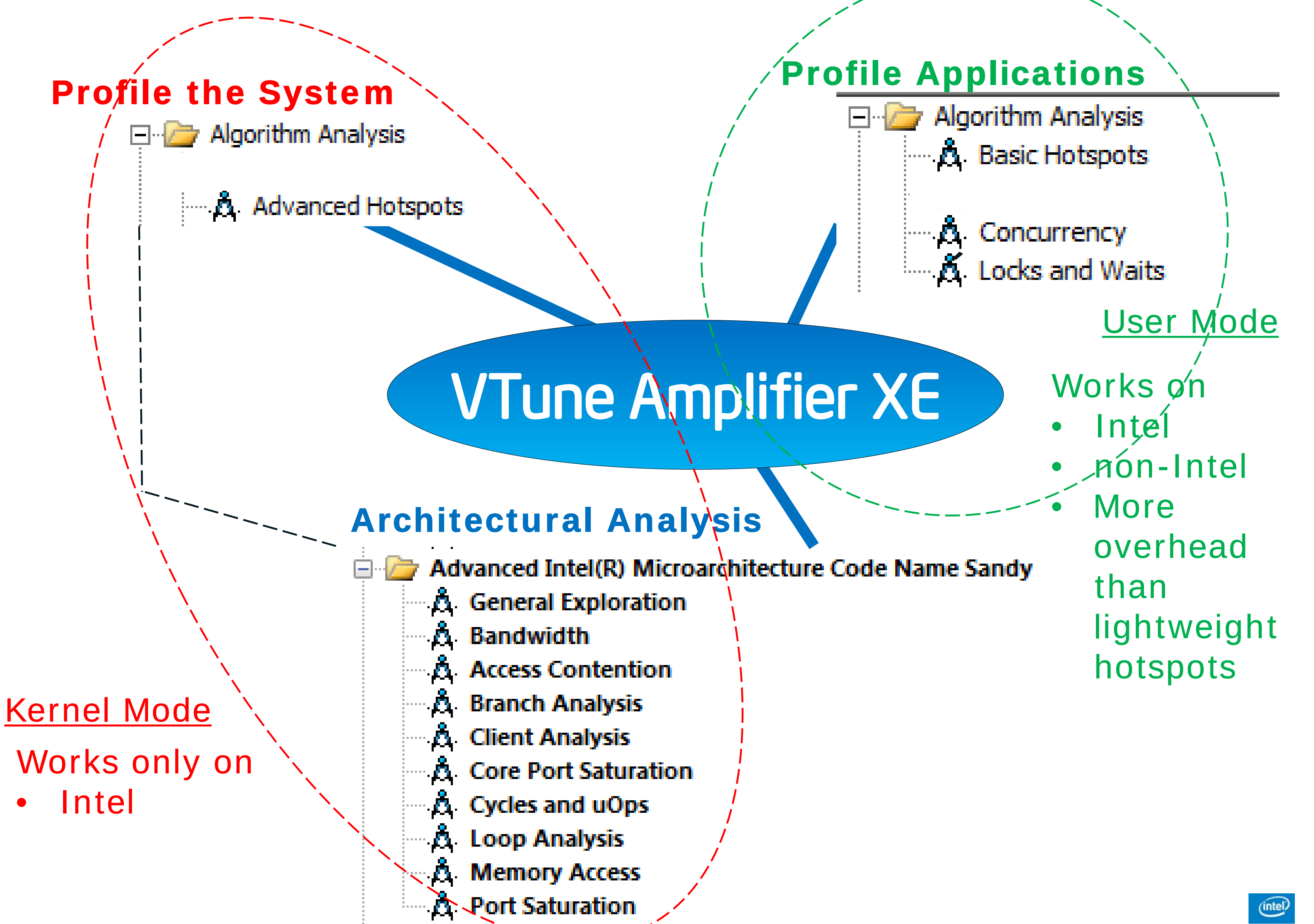
Use Amplifier XE





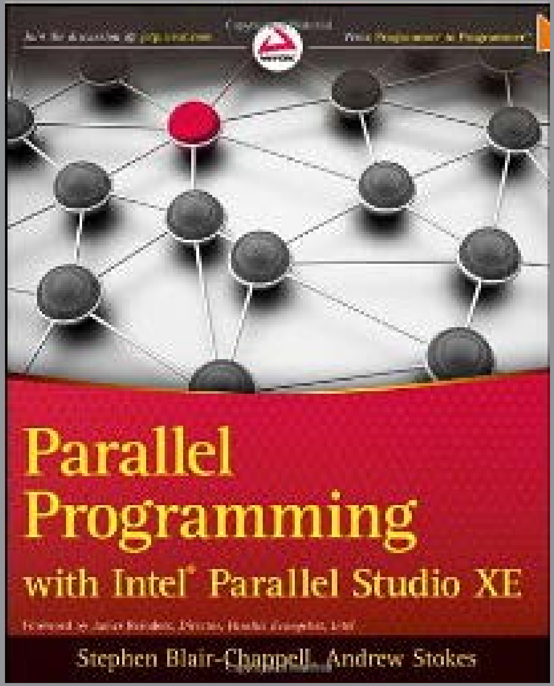
Advantages & Disadvantages of different hotspot methods

Method	Pro	Con
Loop Profiler	• Easy to use • All you need is in compiler package • Profiles loops as well as functions	• Very basic • Code needs instrumenting • No call tree • No comparison function
Auto-parallelizer	• Easy to use • Quick way of spotting right place • Comparison relatively easy	• Easily confounded • IPO obscures loops
Amplifier XE	• Small overhead • Easy to traverse call stack • No special build needed • Multiple options for collection \viewing • Results can be compared	• (~~No loop profiler~~) • No *precise* call graph


Profile the System
Algorithm Analysis
Advanced Hotspots
Profile Applications
Algorithm Analysis
Basic Hotspots
Concurrency
Locks and Waits
User Mode
Works on
• Intel
• non-Intel
• More overhead than lightweight hotspots
VTune Amplifier XE
Architectural Analysis
Advanced Intel(R) Microarchitecture Code Name Sandy
General Exploration
Bandwidth
Access Contention
Branch Analysis
Client Analysis
Core Port Saturation
Cycles and uOps
Loop Analysis
Memory Access
Port Saturation
Kernel Mode
Works only on
• Intel

Hands-on Lab

Lab 1 , Step 1
Hotspots Analysis
Activity 6-1 & 6-4



C:\CLASSFILES\Lab Sources\IPS Book\Chapter 6

Explanation of Compiler Flags

Optimization Notice

/Od (-O0) Building with Optimisation Disabled

- Code is not re-ordered
- ‘Dead code’ is not eliminated
- Improves visibility when using profiling tools.
  - You should use this option when looking for threading errors!
- The code is usually much s l o w e r ☹
- The binaries are usually much Bigger ☹
- **/Zi** (-g) produce debug information (can be used with /O1, /O2, /O3 etc).

Step 1

/O1 (-O1) Optimize for Speed and Size

- This option is very similar to /O2 except that it omits optimizations that tend to increase object code size, such as the in-lining of functions. Generally useful where memory paging due to large code size is a problem, such as server and database applications.
- Auto-vectorization is **not turned on**, even if it is invoked individually by its fine grained switch /Qvec.

Step 2

/O2 (-O2) Optimize for maximum speed.

- This option will create faster code in most cases.
- Optimizations include
  - scalar optimizations
  - inlining and some other
  - Inter-procedural optimizations between functions/subroutines in the same source file
  - vectorization
  - limited versions of a few other loop optimizations such as loop versioning and unrolling that facilitate vectorization.

Step 2

Optimization Notice

/O3 (-O3) Optimizes for further speed increases.

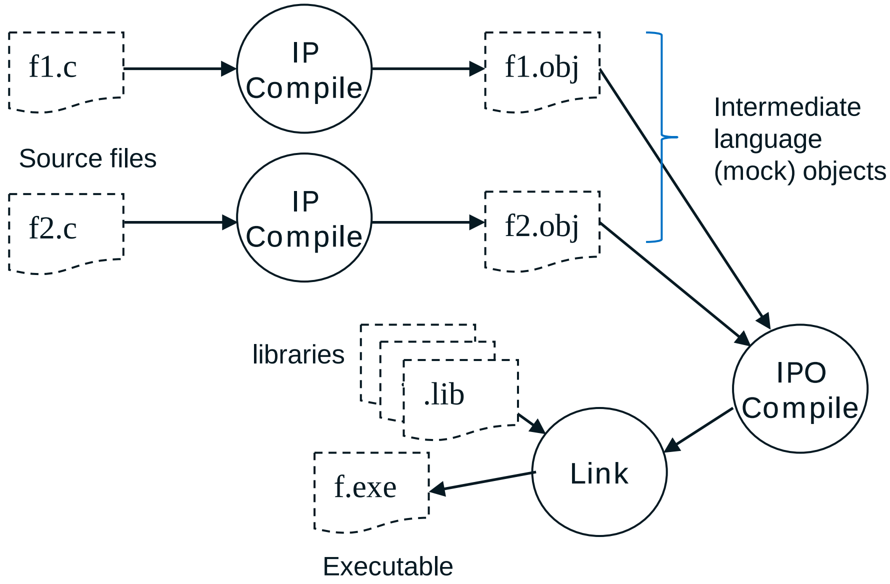
- This includes all the /O2 optimizations, together with other High Level Optimizations.

- These high level optimizations include more aggressive strategies such as:
  - scalar replacement,
  - data pre-fetching,
  - loop optimization,
  - among others.

Step 2

Optimization Notice

Interprocedural Optimisation

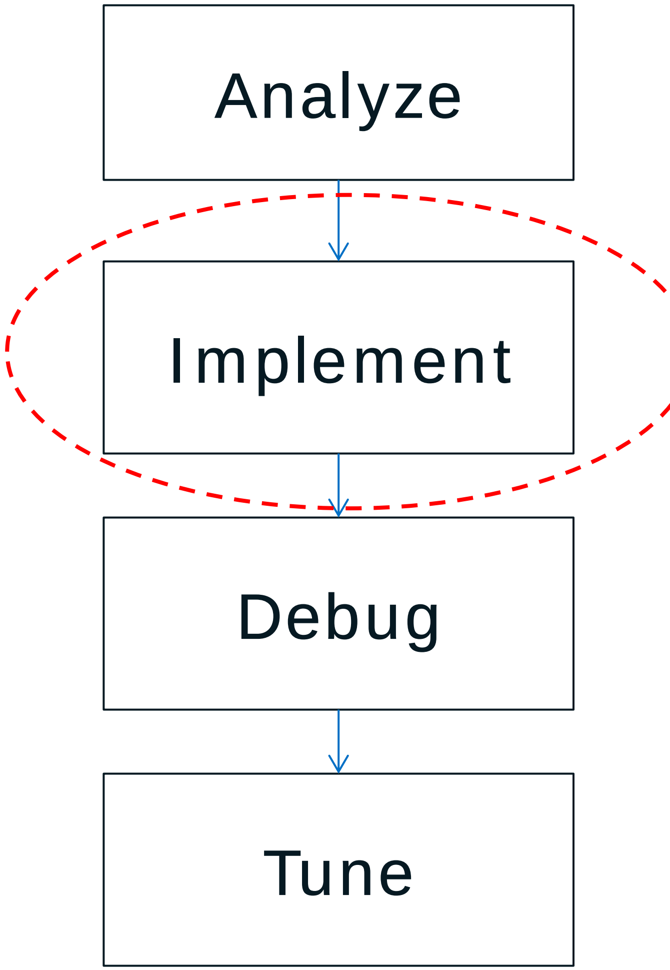


- **/Qipo** (Linux –ipo)
- *Call to functions* are replaced by the *body of the function (aka inlining)*
- Can lead to loss of symbol information (unhelpful when debugging)
- **/Ob0, /Ob1, /Ob2** turns of inlining. (Linux –inline-level=0 ,1,2

Step 4

Optimization Notice

Steps in moving from Serial to Parallel



Step 1 : Look for hotspots in application. These are best candidates to make parallel

Step 2: Add parallel constructs into source code

Step 3: Check if any parallel-type errors have been introduced

Step 4: Tune the parallel application.

Key Questions – Implement

What's the best method?

How much programming effort is required?

Is my code still working?

Optimization Notice

Language to help parallelism

Parallel Code

Intel® Cilk™ Plus

OpenMP

Intel® Threading Building Blocks

Intel® MPI

Fortran Coarrays

OpenCL

Native Threads

```
#pragma omp parallel for
for(i=1;i<=4;i++) {
    printf("Iter: %d", i);
}
```

```
cilk_for (int i = 0; i < max_row; i++)
{
    for (int j = 0; j < max_col; j++ )
    {
        p[i][j] = mandel( complex(scale(i), scale(j)));
    }
}
```

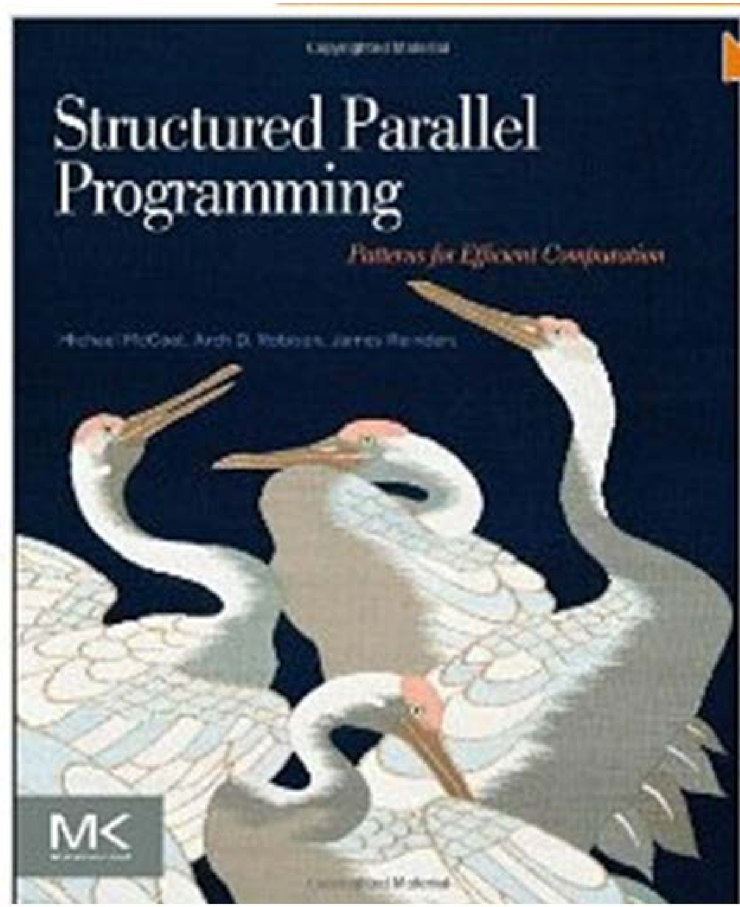

Optimization Notice

Structured Parallel Programming

Michael McCool, James Reinders, Arch Robison

Publisher: Morgan Kaufmann (31 July 2012), ISBN-10: 0124159931

Programming is now parallel programming. Much as structured programming revolutionized traditional serial programming decades ago, a new kind of structured programming, based on patterns, is relevant to parallel programming today. Parallel computing experts and industry insiders Michael McCool, Arch Robison, and James Reinders describe how to design and implement maintainable and efficient parallel algorithms using a pattern-based approach. They present both theory and practice, and give detailed concrete examples using multiple programming models. Examples are primarily given using two of the most popular and cutting edge programming models for parallel programming: Threading Building Blocks, and Cilk Plus. These architecture-independent models enable easy integration into existing applications, preserve investments in existing code, and speed the development of parallel applications. Examples from realistic contexts illustrate patterns and themes in parallel algorithm design that are widely applicable regardless of implementation technology. This title provides the patterns-based approach that offers structure and insight that developers can apply to a variety of parallel programming models. It develops a composable, structured, scalable, and machine-independent approach to parallel computing. It includes detailed examples in both Cilk Plus and the latest Threading Building Blocks, which support a wide variety of computers.



Optimization Notice

Introduction to Concurrency in Programming Languages

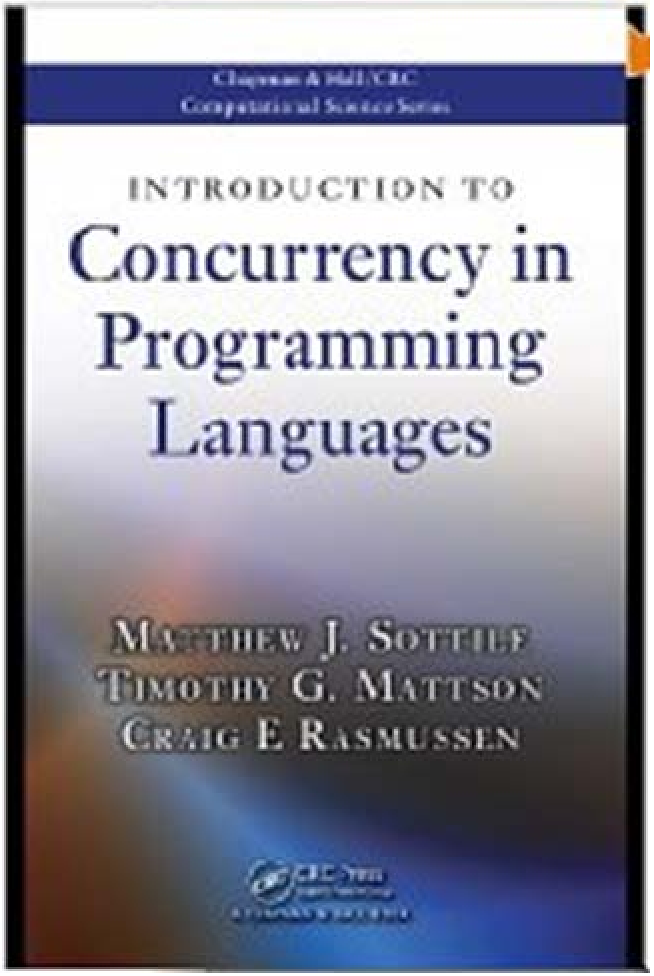
Matthew J. Sottile, Timothy G. Mattson, Craig E. Rasmussen

Publisher: Chapman and Hall/CRC (7 Oct 2009). ISBN:ISBN-10: 1420072137

Exploring how concurrent programming can be assisted by language-level techniques, **Introduction to Concurrency in Programming Languages** presents high-level language techniques for dealing with concurrency in a general context. It provides an understanding of programming languages that offer concurrency features as part of the language definition.

The book supplies a conceptual framework for different aspects of parallel algorithm design and implementation. It first addresses the limitations of traditional programming techniques and models when dealing with concurrency. The book then explores the current state of the art in concurrent programming and describes high-level language constructs for concurrency. It also discusses the historical evolution of hardware, corresponding high-level techniques that were developed, and the connection to modern systems, such as multicore and manycore processors. The remainder of the text focuses on common high-level programming techniques and their application to a range of algorithms. The authors offer case studies on genetic algorithms, fractal generation, cellular automata, game logic for solving Sudoku puzzles, pipelined algorithms, and more.

Illustrating the effect of concurrency on programs written in familiar languages, this text focuses on novel language abstractions that truly bring concurrency into the language and aid analysis and compilation tools in generating efficient, correct programs. It also explains the complexity involved in taking advantage of concurrency with regard to program correctness and performance.



Optimization Notice

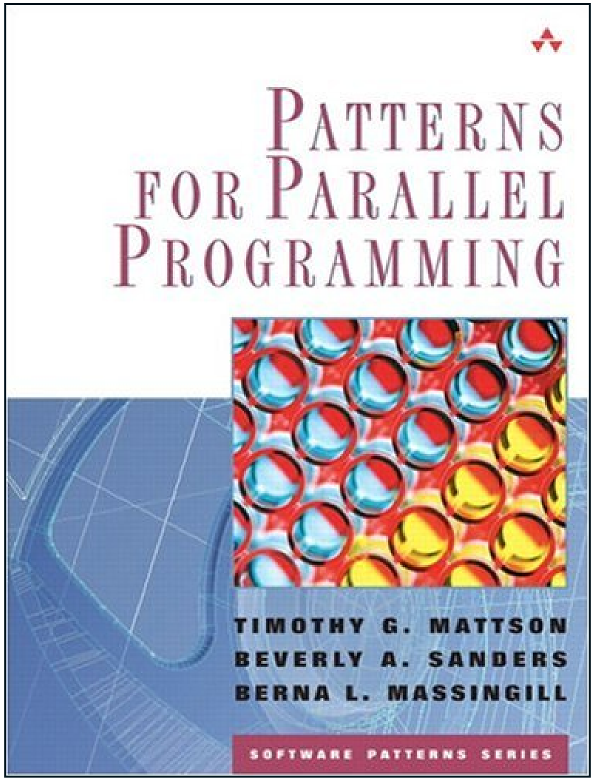
Patterns for Parallel Programming

Details a pattern language for parallel algorithm design

Examples in MPI, OpenMP and Java are given

Represents the author's hypothesis for how programmers think about parallel programming

Patterns for Parallel Programming, Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Addison-Wesley, 2005, ISBN 0321228111

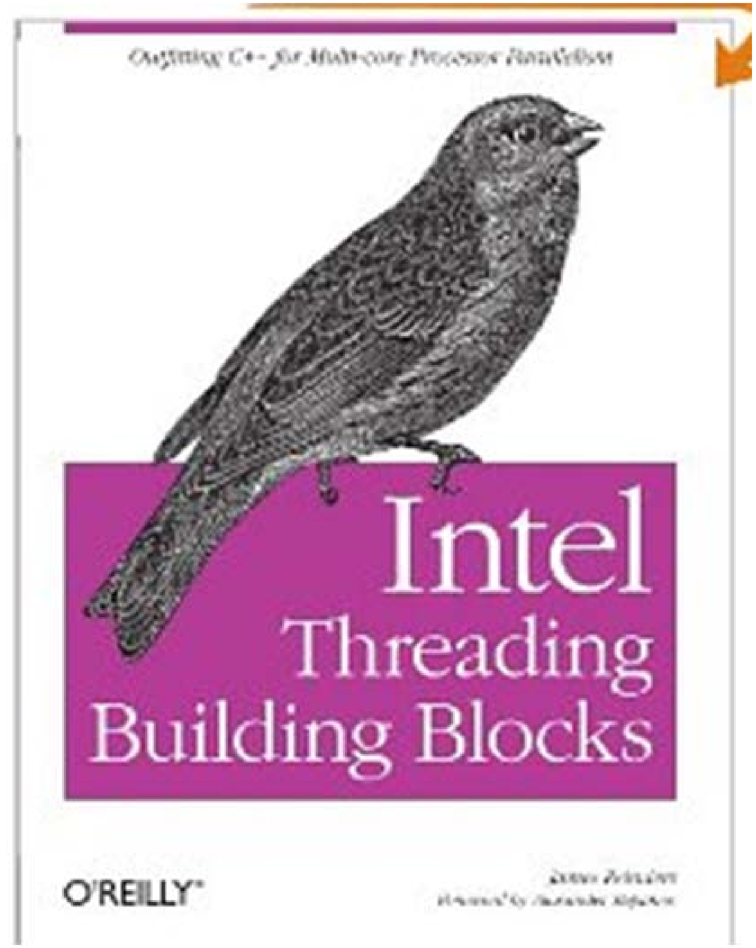


Intel Threading Building Blocks
James Reinders,

O'Reilly Media; 1 edition (19 July 2007) ISBN-10: 0596514808

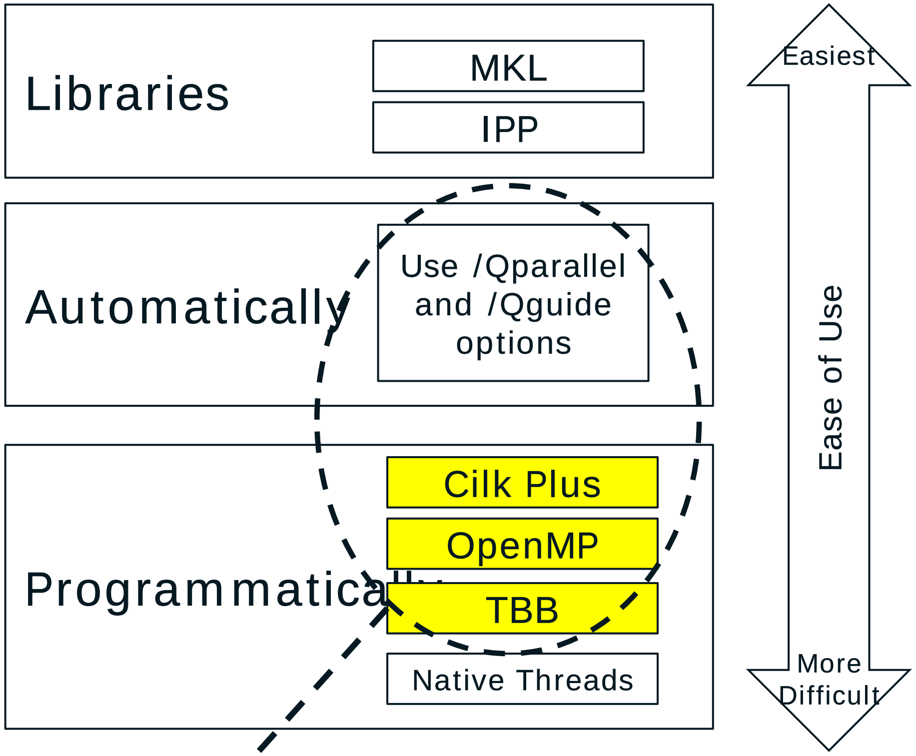
Multi-core chips from Intel and AMD offer a dramatic boost in speed and responsiveness, and plenty of opportunities for multiprocessing on ordinary desktop computers. But they also present a challenge: More than ever, multithreading is a requirement for good performance. This guide explains how to maximize the benefits of these processors through a portable C++ library that works on Windows, Linux, Macintosh, and Unix systems. With it, you'll learn how to use Intel Threading Building Blocks (TBB) effectively for parallel programming -- without having to be a threading expert.

Written by James Reinders, Chief Evangelist of Intel Software Products, and based on the experience of Intel's developers and customers, this book explains the key tasks in multithreading and how to accomplish them with TBB in a portable and robust manner. With plenty of examples and full reference material, the book lays out common patterns of uses, reveals the gotchas in TBB, and gives important guidelines for choosing among alternatives in order to get the best performance.



Optimization Notice

Scope of Chapter 6



Covered in this chapter

What Is OpenMP?

Portable, shared-memory threading API

- Fortran, C, and C++
- Multi-vendor support for both Linux and Windows

http://www.openmp.org

Current spec is OpenMP 4.0

(combined C/C++ and Fortran)

Stan ... ism

Supp

Stan ... directed threading experience

Combines serial and parallel code in single source

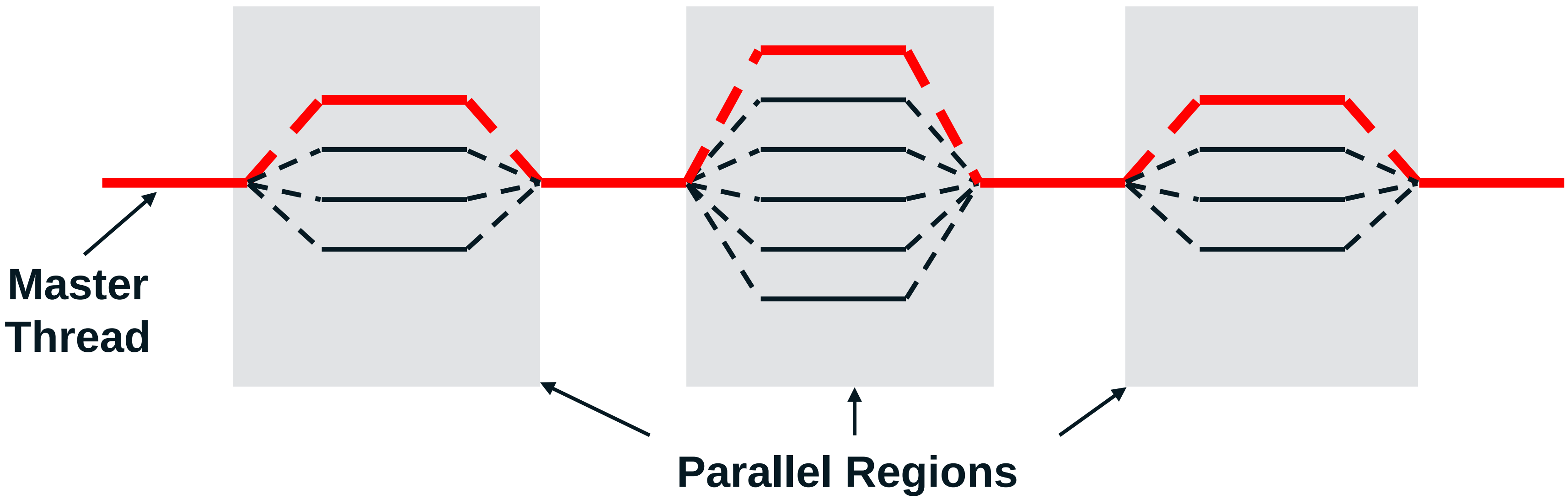
- Excellent support of “incremental” parallelization
- No need for serial and parallel source code releases

Optimization Notice

Programming Model

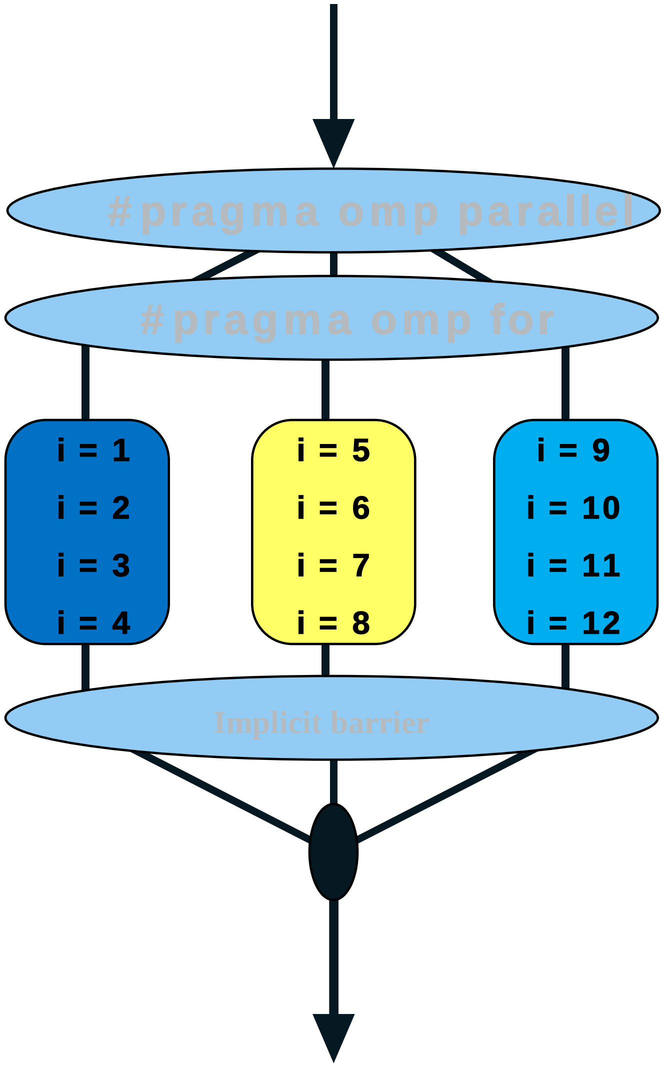
Fork-Join Parallelism:

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



omp for Construct

```
// assume N=12
#pragma omp parallel
#pragma omp for
  for(i = 1, i < N+1, i++)
    c[i] = a[i] + b[i];
```



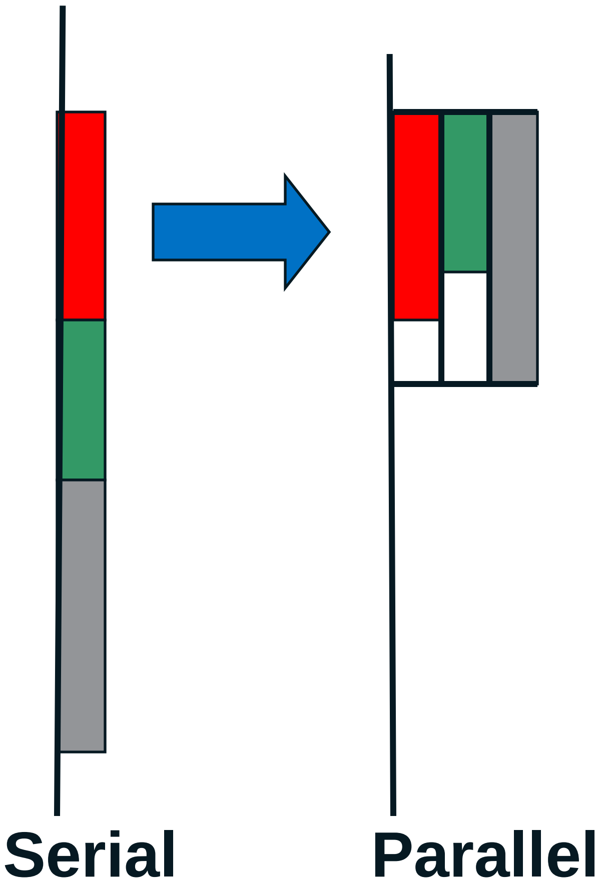
Threads are assigned an independent set of iterations

Threads must wait at the end of work-sharing construct

Advantage of Parallel Sections

Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
   #pragma omp section
   phase1();
   #pragma omp section
   phase2();
   #pragma omp section
   phase3();
}
```



Optimization Notice

The Private Clause

Reproduces the variable for each task

- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
 float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
     x = a[i]; y = b[i];
     c[i] = x + y;
     }
}
```

Schedule Clause Example

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

Iterations are divided into chunks of 8

- If start = 3, then first chunk is **i**={3,5,7,9,11,13,15,17}

Implementing Parallelism

- Parallelizing loops
- Parallelizing sections and functions
- Parallelizing recursive functions
- Parallelizing pipelined applications
- Parallelizing linked lists

Optimization Notice

The Beauty of Lambda

[*capture_mode*] (*formal_parameters*) -> *return_type* {*body*}

[&] ⇒ by-reference
[=] ⇒ by-value
[] ⇒ no capture

Can omit if there are no parameters *and* return type is implicit.

Can omit if return type is void or *code* is "return *expr*;"

An example lambda function

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
  char Message[]="The Beauty of Lambda!";

  int NumSpaces = 0;
  for_each(
    Message,              //
    Message + sizeof(Message),
    [&NumSpaces] (char c) { if (c == ' ') NumSpaces++;}
  );
  cout << "There are " << NumSpaces <<" spaces in: '"<<
  Message << "'" <<endl;
}
```

<u>STL template for_each</u>

template<class InputIterator, class Function> Function for_each(InputIterator _First, InputIterator _Last, Function _Func);

Parallelizing for loops

```
#include < cilk/ cilk.h >
cilk_for (int i = 0; i<100; i++)
{
   work( i);
}
```

```
#pragma omp parallel
{
 #pragma omp for
 for (int i = 0; i<100; i++)
 {
  work( i);
 }
} // end of parallel region
```

```
#include < tbb.h >
parallel_for (size_t( 0), 100,
   [=]( size_t i) {
   work( i);
  } // end of lambda code
); // end of parallel _for
```

Key

Cilk Plus
TBB
OpenMP

Parallelizing while loops

```
#include < tbb.h >
#include < vector >
void Work( int Val)
{ // do some work here}

Func()
{
  std:: vector < int >
  s;
  s.push_back( 0);  // etc
tbb:: parallel_do
(
  s.begin(), s.end(),
  [&]( int Val) { Work( Val);}
  );
}
```

```
#pragma omp parallel
{
 #pragma omp single nowait
 {
  int counter = 0;
  while( counter < 10)
  {
   counter++;
    #pragma omp task firstprivate(counter)
    {
     work( counter);
 } } } } // implicit barrier
```

```
int j = 0;
while (j < 100)
{
    cilk_spawn Prime( Pri);
    j + +;
}
```

Key

- Cilk Plus
- TBB
- OpenMP

Parallelizing sections and functions

```
#include < cilk/cilk.h >
int main()
{
 cilk_spawn Work1();
 cilk_spawn Work2();
 Work3();
 cilk_sync
}
```

```
#include < tbb/tbb.h >
int main()
{
 tbb:: parallel_invoke(
     []{ Work1();},
     []{ Work2();},
     []{ Work3();}
 );
}
```

```
int main()
{
  #pragma omp parallel sections
  {
    #pragma omp section
    Work1();
    #pragma omp section
    Work2();
    #pragma omp section
    Work3();
  }
}
```

Key

Cilk Plus
TBB
OpenMP

Parallelizing recursive functions

```
#include < cilk/ cilk.h >
void Work( int i)
{
  if( i > 4) return;
  printf(" S% d\ n", i);
  cilk_spawn Work( i +
1);
  printf(" E %d\ n", i);
}

int main()
{
  int i = 0;
  Work( i);
}
```

```
void Work( int i)
{
  if( i > 4) return;
  #pragma omp task firstprivate( i)
  {
     printf(" S% d\ n", i);
    Work( i + 1);
    printf(" E %d\ n", i);
  }
 }

int main()
{
  int i = 0;
  #pragma omp parallel
  #pragma omp single
  {
   Work( i);
  }
}
```

Key

- Cilk Plus
- TBB
- OpenMP

Optimization Notice

Parallelizing recursive functions

```
#include < stdio.h >
#include < tbb/tbb.h >
tbb:: task_group g;
void Work( int i)
{
  if( i > 4) return;
  g.run(
    [=]{ // spawn a task
      printf(" S% d\ n", i);
      Work( i + 1);
      printf(" E %d\ n", i);
    }
  );
}

int main()
{ int i = 0;
  Work( i);
  g.wait(); // wait for tasks to complete
}
```

Key

Cilk Plus

TBB

OpenMP

Optimization Notice

Parallelizing pipelined functions

```
int main()
{
  FILE *pF = fopen(".\\ Test.Data"," r");
  FILE *pO = fopen(" Out.Data"," w");
  for (int j=0;j<LINE_LENGTH;j++)
  fscanf( pF,"% d ",& LineIn[ 0][ j]);
  #pragma omp parallel
  {
    for (int i = 0; i < NUM_LINES; i++)
    {
      // Pipeline STAGE 1
     #pragma omp single nowait
     {
       // start reading the next line
       // Don't read beyond end
       if( i < NUM_LINES-1);
       {
         for (int j=0;j<LINE_LENGTH;j++)
          fscanf( pF,"% d",&LineIn[i+1][
j]);
        }
       }
```

```
 // Pipeline STAGE 2
 #pragma omp for schedule( dynamic)
 for (int j = 0; j<LINE_LENGTH;j++)
 LineOut[i][j]=
   sqroot(( float)LineIn[i][ j]);

 //Pipeline STAGE 3
 #pragma omp single nowait
  {
   for (int j=0; j<LINE_LENGTH;j++)
      fprintf( pO,"% f ", LineOut[ i][ j]);
      fprintf( pO,"\ n");
     }
    }
 }
 fclose( pF);
 fclose( pO);
 return 0;
}
```

Key

Cilk Plus
TBB
OpenMP

Parallelizing pipelined functions

```
int main()
{
  int i = 0;
  int ntokens = 24;
  FILE *pF = fopen(".\\ Test.Data"," r");
  FILE *pO=fopen("Data"," w");
  parallel_pipeline
  (
     ntokens,
      tbb::make_filter < void, int >
      (
       filter:: serial_in_order, [& i,& pFile]
       ( flow_control& fc)-> int
       {
         if (i < NUM_LINES)
         {
           for(int j=0;j<LINE_LENGTH;j++)
            fscanf( pF,"% d",&LineIn[i][ j]);
           return i++;
         }
         else fc.stop();
         }
        )
```

```
& tbb:: make_filter< int, int >
   ( filter:: serial_in_order, []( int i)->int
    {
        parallel_for
       (size_t( 0), (size_t) LINE_LENGTH,
        [&]( size_t j)
        {
        LineOut[i][j]=
             sqroot(( float) LineIn[ i][ j]);
         }
        );
       return i;
     }
    )
& tbb:: make_filter < int, void >
  ( filter:: serial_in_order,[& pO]( int i)
     {
       for (int j=0; j<LINE_LENGTH; j++)
         fprintf( pO,"% f ", LineOut[i][j]);
       fprintf( pOutputFile,"\ n");
      }
    )
 );
}
```

Key

Cilk Plus

TBB

OpenMP

Optimization Notice

Parallelizing Linked Lists

```
void RunThoughLinkedList() {
  tbb:: task_group g;
  node *pHead = Head;
  while( pHead != NULL)
 {
   g.run(
   [=] {
     Work( pHead);
     }
  );
   pHead = pHead->next;
  }
  g.wait();
}
```

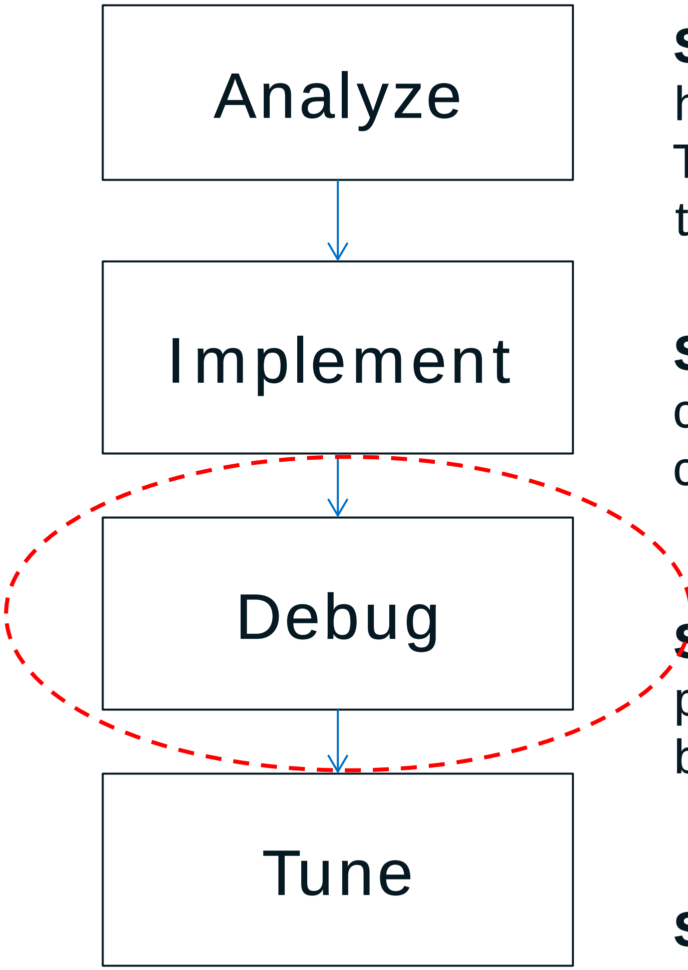
```
void RunThoughLinkedList()
{
  #pragma omp parallel {
     #pragma omp single{
       node *pHead = Head;
       while( pHead != NULL)
      {
        #pragma omp task firstprivate( pHead)
        {
          Work( pHead);
         }
         pHead = pHead->next;
} } } }
```

```
#include < cilk/ cilk.h >
void RunThoughLinkedList()
{
  node *pHead = Head;
  while( pHead != NULL)
  {
    cilk_spawn Work( pHead);
    pHead = pHead-> next;
    } }
```

Key

Cilk Plus
TBB
OpenMP

Steps in moving from Serial to Parallel



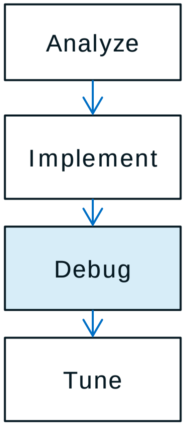
Step 1 : Look for hotspots in application. These are best candidates to make parallel

Step 2: Add parallel constructs into source code

Step 3: Check if any parallel-type errors have been introduced

Step 4: Tune the parallel application.

Key Questions - Verify



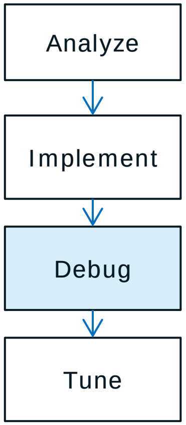
Is the parallelism correct?

Do I have deadlocks or data races?

Do I have memory errors?

Does my program still work as intended?

Ways to Find your Parallel Errors

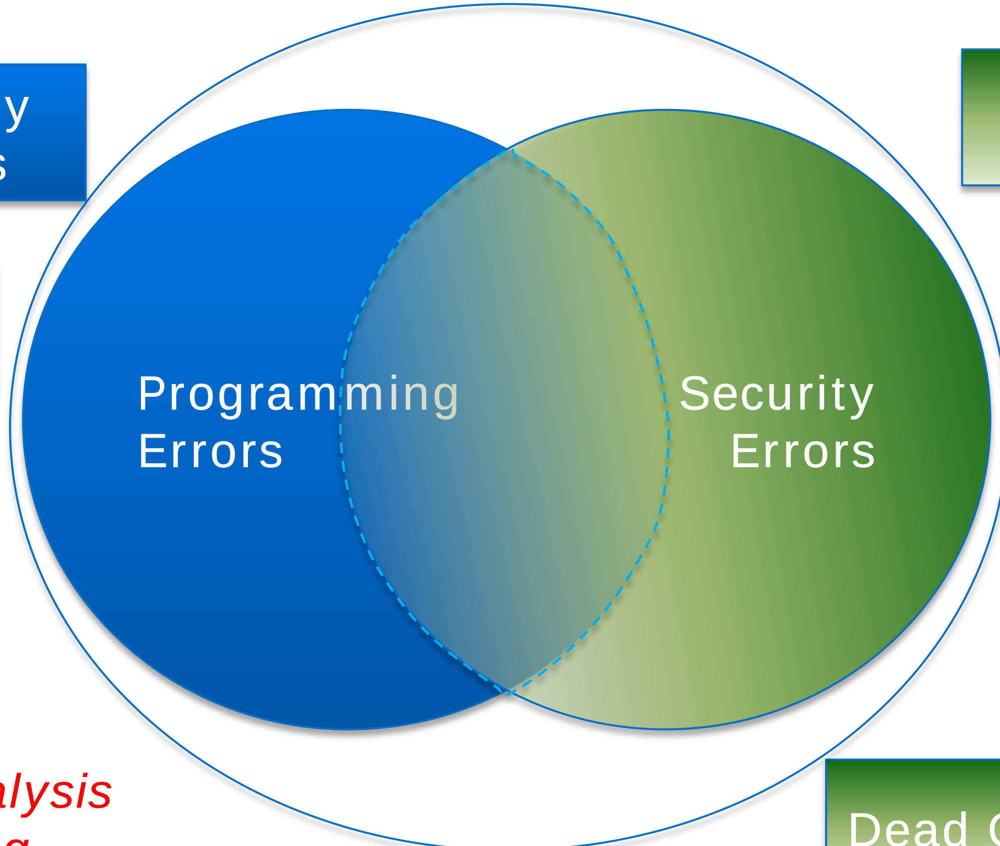


1. Use Inspector XE
2. Use Advisor
3. Use Intel-provided Data Race Detector in GDB

Dynamic vs Static Error Detection

Memory Errors

Threading Errors



Memory Errors

Threading Errors

Dangerous Usage

Module Inconsistency

Dead Code

Detect with ***Dynamic*** *Analysis and Debugging*

Intel Inspector XE
and\or
Data Race Detector in GDB

Detect with ***Static*** *Security Analysis*

Intel Compiler
and
Intel Inspector XE

Optimization Notice

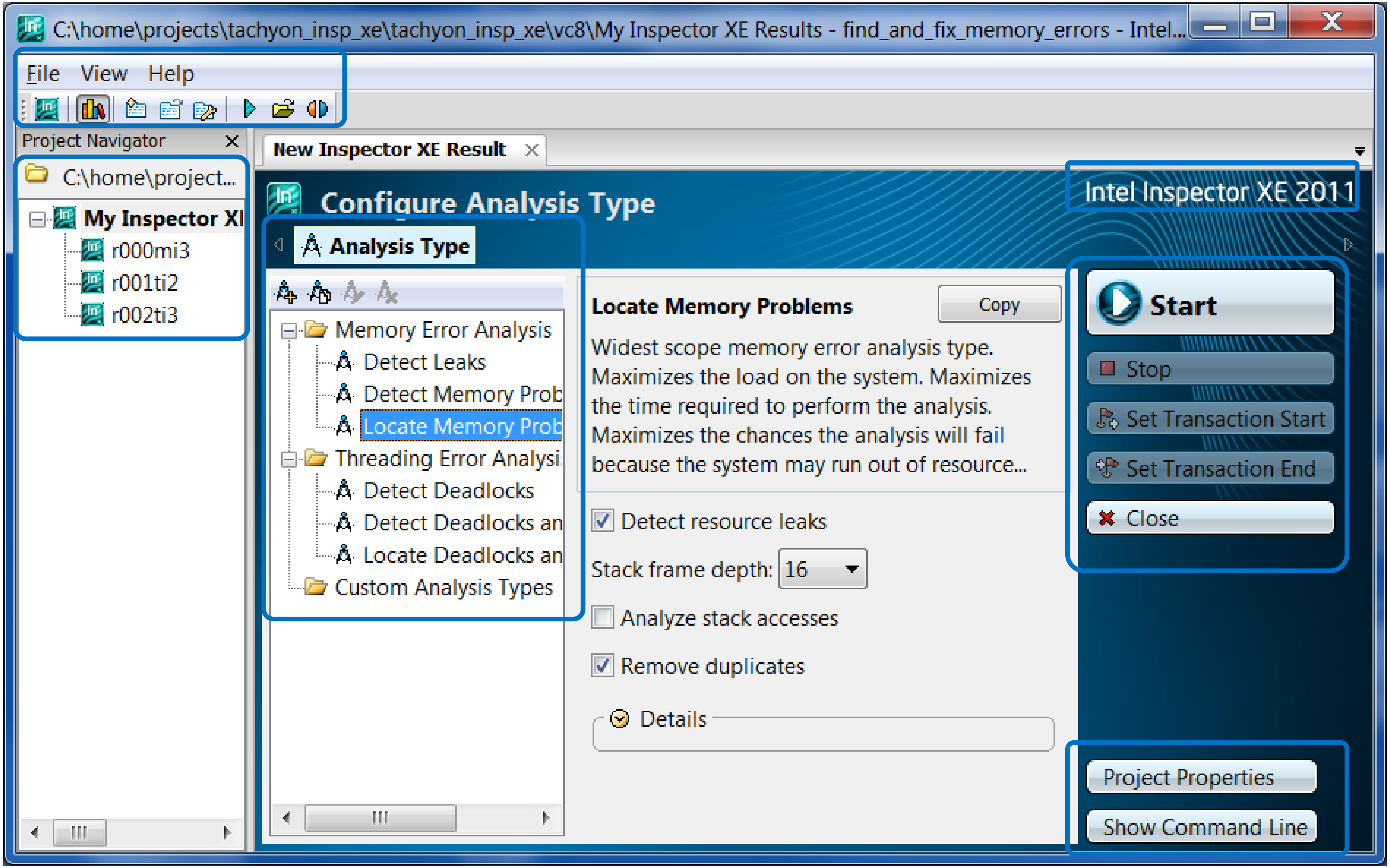
Inspector XE – Threading Errors & warnings

- Data Races
- Deadlocks
- Lock hierarchy violations
- Potential privacy infringements
- Other threading info

Inspector XE – Memory Errors & warnings

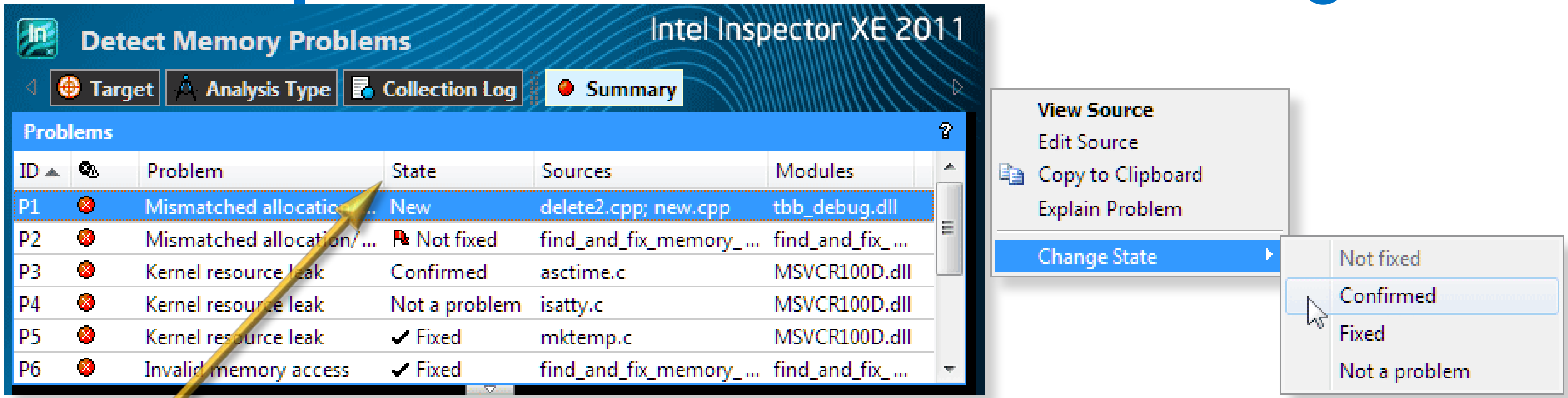
- Memory leaks
- Bad use of memory allocation \ de-allocation
- Invalid memory access

Inspector XE - Standalone GUI



Optimization Notice

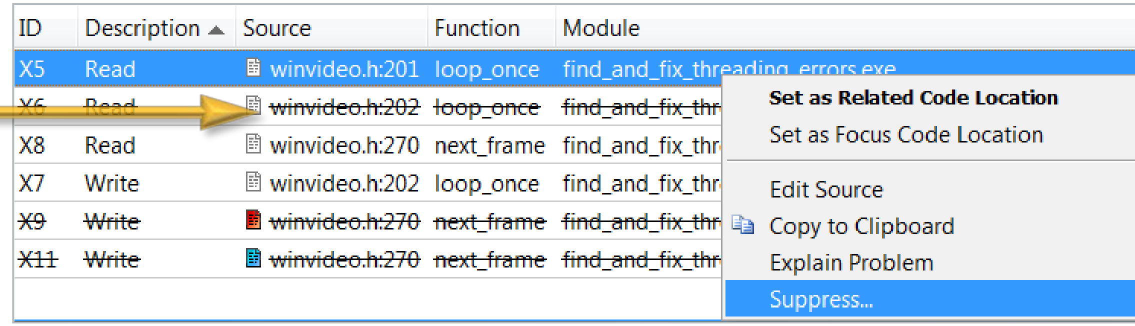
Problem State Lifecycle
Makes problems easier to manage



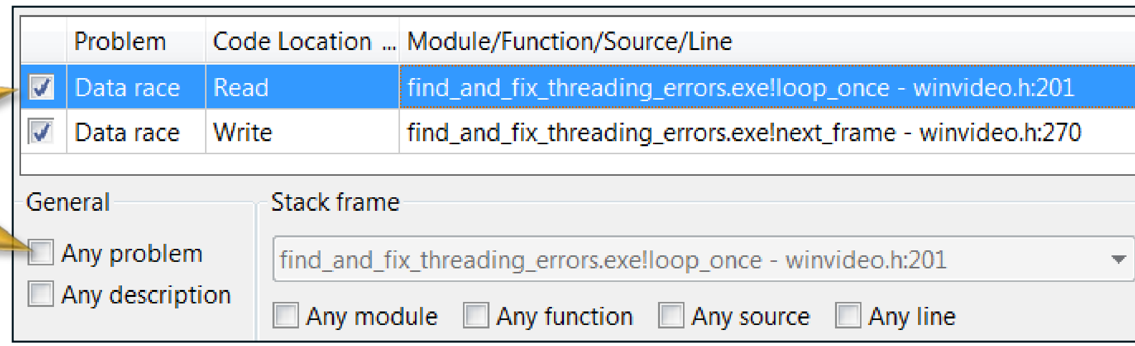
State	Description
New	Detected by this run
Not Fixed	Previously seen error detected by this run
Not a Problem	Set by user – tool will not change
Confirmed	Set by user – tool will not change
Fixed	Set by user
Regression	Error detected with previous state of “Fixed”

Suppressions Manage False Errors

Suppressions are marked (shown) or hidden entirely



Be specific or select group of similar problems



Suppressions are saved in one or more files

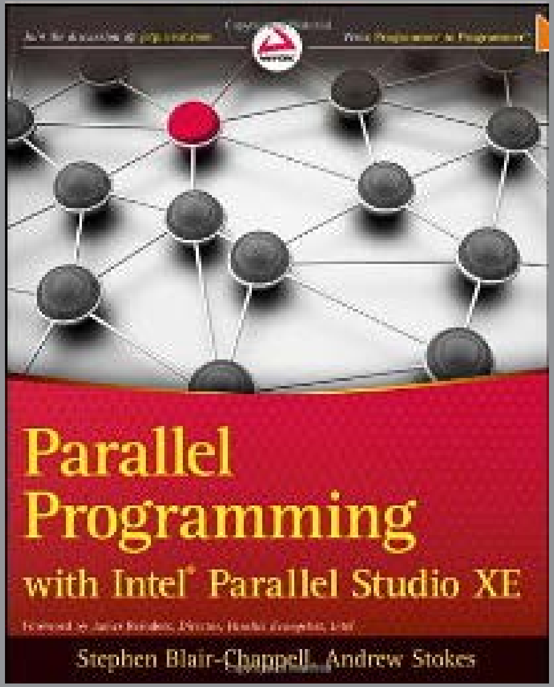
Tool suppresses all files from specified folder(s)

Private & Public suppression folders

Copy a suppression to public folder to promote

Hands-on Lab

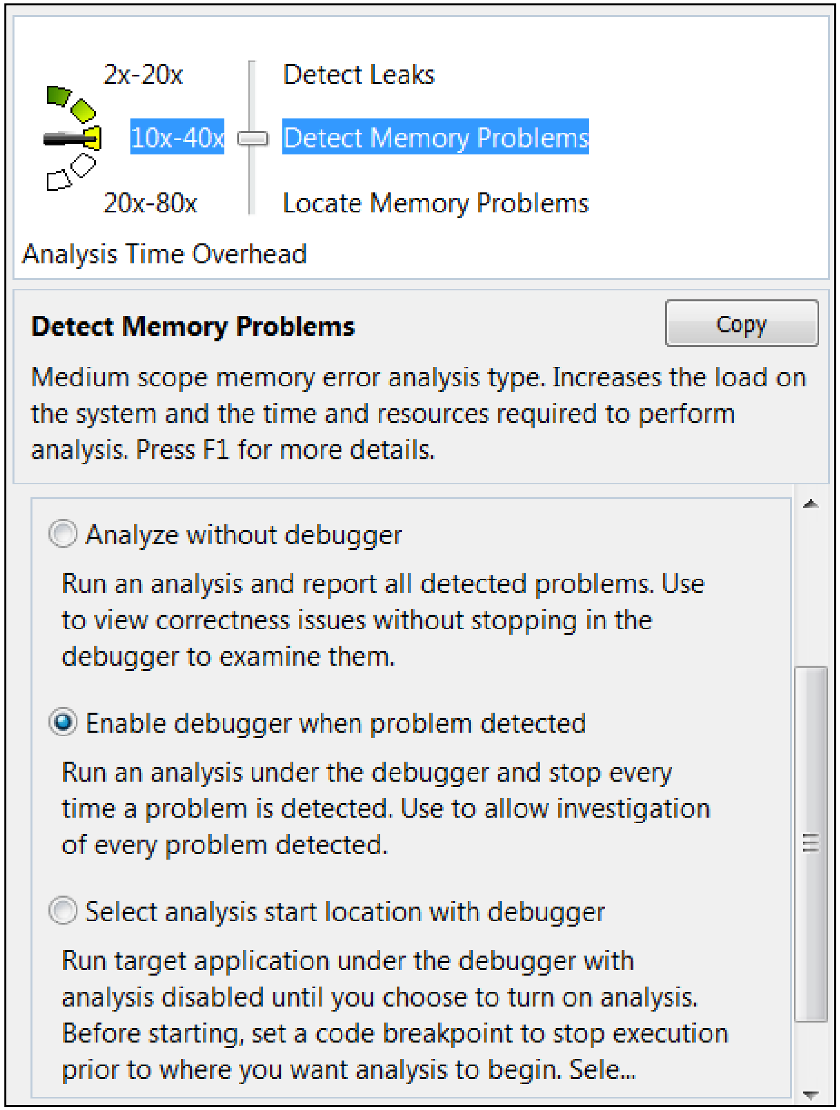
Lab 1 , Step 3
Debugging Parallel Errors
Activity 8-1 & 8-2



C:\CLASSFILES\Lab Sources\IPS Book\Chapter 8

Integrated Debugger Support

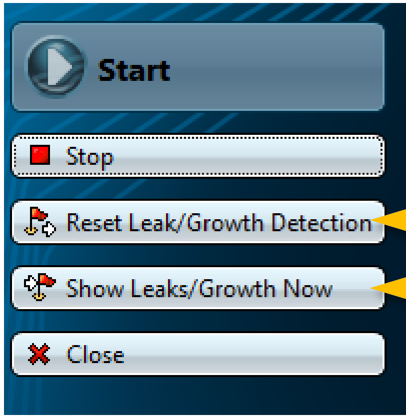
- Break into debugger
  - Analysis can stop when it detects a problem
  - User is put into a standard debugging session
- Windows*
  - Microsoft* Visual Studio Debugger
- Linux*
  - gdb
  - Intel® Debugger



Optimization Notice

Analyze Memory Growth

Transactional Applications



During Analysis:

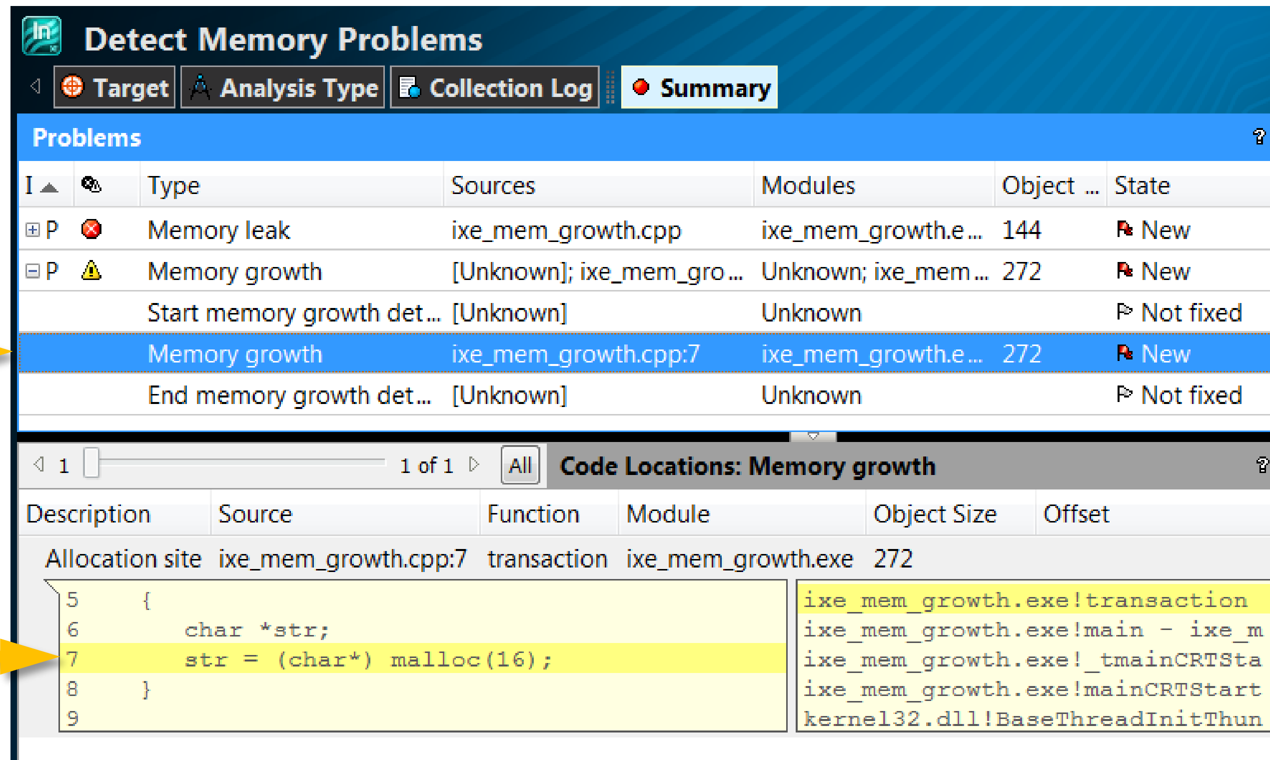
Set Start Point

Set End Point

Analysis Results:

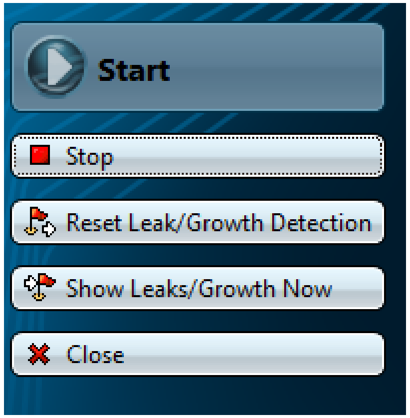
Memory Growth Problem Set

Code location for each block of memory that was allocated but not de-allocated during the time period



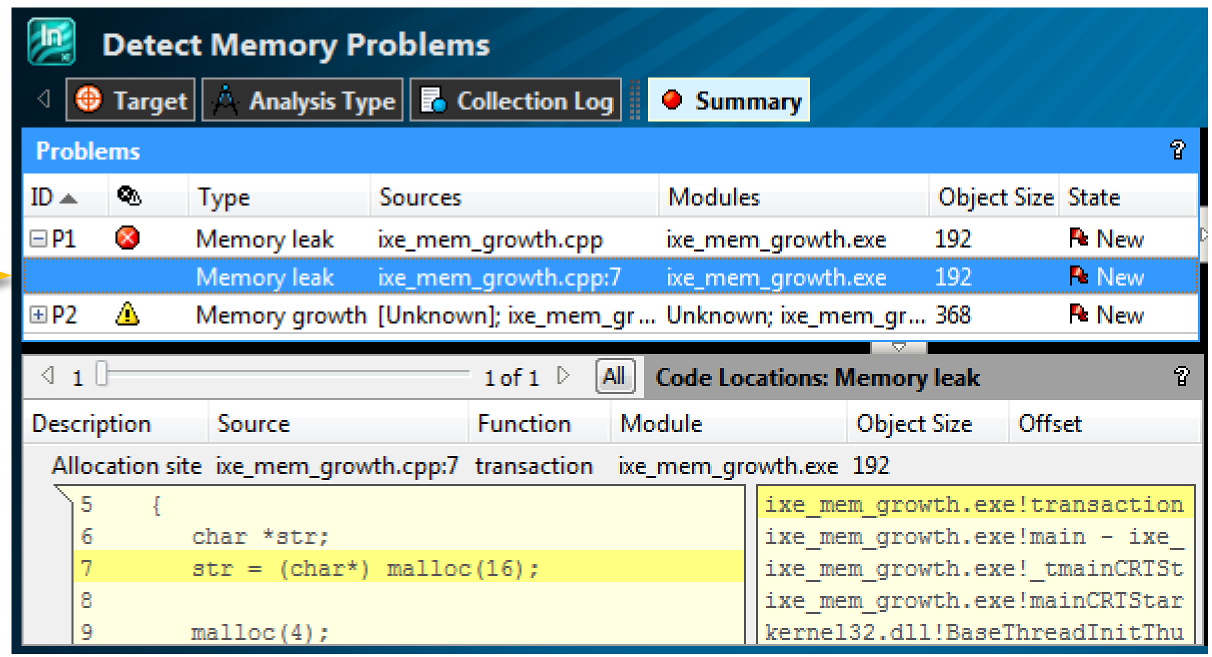
On-demand leak detection

Detect memory leaks before application exits



- Check code regions between points 'A' and 'B‘ for leaks
- Check daemon processes for leaks
- Check crashing processes for leaks

Memory Leak
shown during run time



Optimization Notice

Inspector APIs for memory growth and leak detection.

API	Purpose
__itt_heap_growth	memory growth detection
__itt_heap_leaks	on-demand leak detection
__itt_heap_reset_detection	reset the growth/leak baseline
__itt_heap_record	generate a report

Child Program Analysis

Running a top level script is the norm for some Linux apps. For such cases, a different *Child Program* can be analyzed (not necessarily the app launched by Inspector XE).

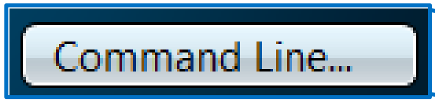
Limitations:

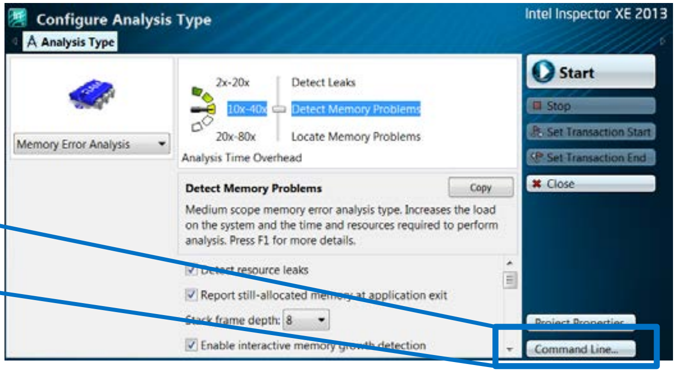
- Only the first instance of *Child Program* will be analyzed by Inspector XE analysis.
- *Child Program* name is the one shown in Windows Task Manager or the name shown in “ps –aef” on Linux.
- Multi-process analysis is not supported for .NET applications.

Child Program Analysis is very useful in multi-process scenario

Command Line Interface

- inspxe-cl is the command line:
  - Windows: `C:\Program Files\Intel\Inspector XE \bin[32|64]\inspxe-cl.exe`
  - Linux: `/opt/intel/inspector_xe/bin[32|64]/inspxe-cl`
- Help:
  `inspxe-cl –help`
- Set up command line with GUI





- Command examples:

```
inspxe-cl -collect-list
inspxe-cl –collect ti2 -- MyApp.exe
inspxe-cl –report problems
```

Great for regression analysis – send results file to developer
Command line results can also be opened in the GUI

Optimization Notice

Reporting

To generate a report:

```
inspxe-cl –R=<report-type> <results directory name>
```

Sample commands:

```
inspxe-cl –report-list
inspxe-cl -report=summary
inspxe-cl -report=problems
```

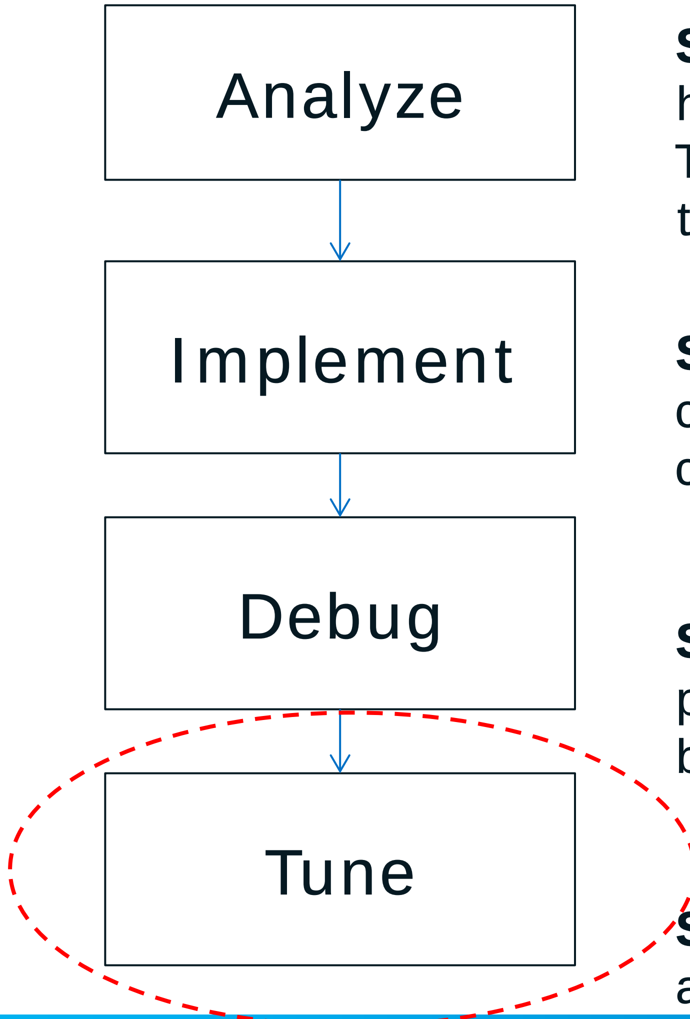
Example:

```
cd /home/user/testProgram/r000mi
cd ..
inspxe-cl -R=observations r000mi
```

If you have time,
re-run the inspection
from the command line

Report generation is very convenient to use from command line.

Steps in moving from Serial to Parallel



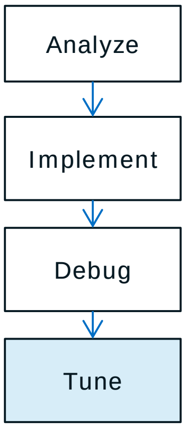
Step 1 : Look for hotspots in application. These are best candidates to make parallel

Step 2: Add parallel constructs into source code

Step 3: Check if any parallel-type errors have been introduced

Step 4: Tune the parallel application.

Key Questions -Tune



Do my tasks do equal amounts of work?

Is my application scalable?

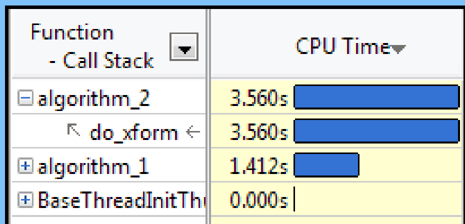
Is the threading running efficiently?

Intel® VTune™ Amplifier XE

Performance Profiler

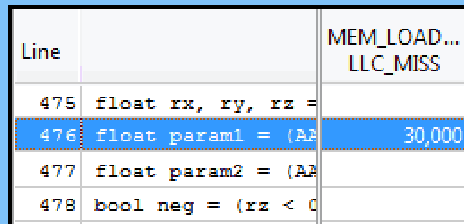
Where is my application…

Spending Time?



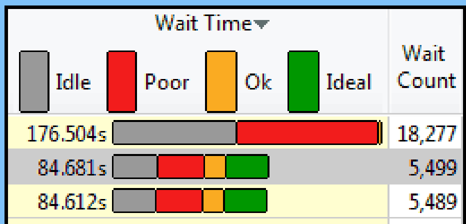
- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?



- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?



- See locks by wait time
- Red/Green for CPU utilization during wait

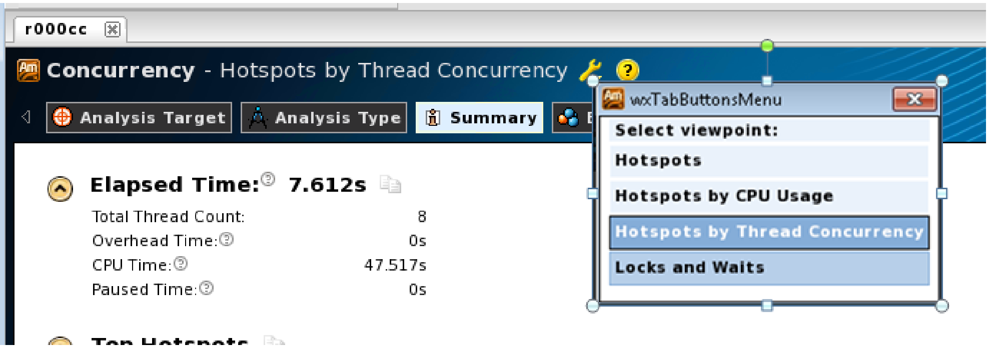
- **Windows & Linux**
- **Low overhead**
- **No special recompiles**

Advanced Profiling For Scalable Multicore Performance

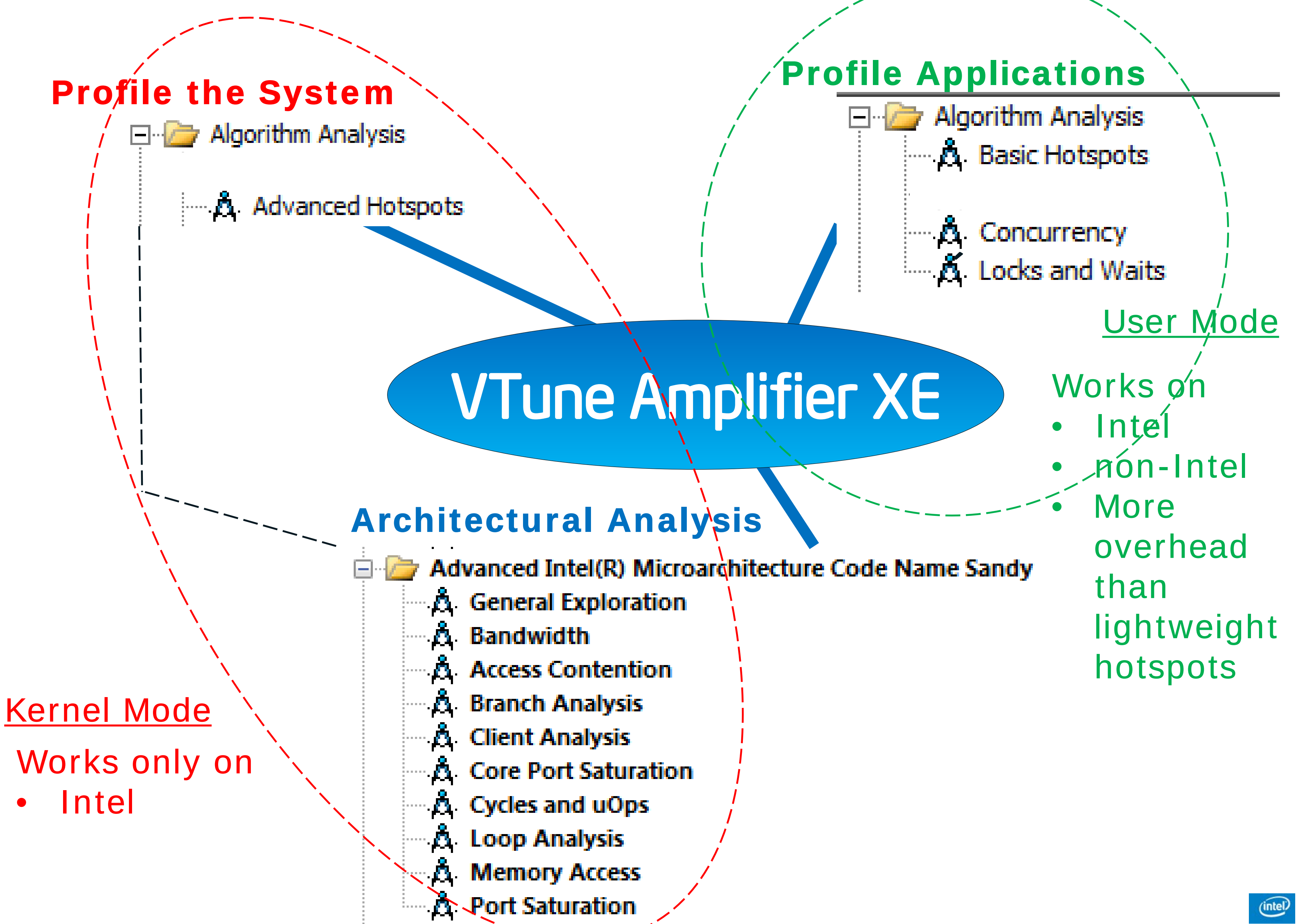
Amplifier XE – 'Parallel' Analysis Types

- Hotspots
- Concurrency
- Locks and waits

You can swap between the different analysis types – but not all data is captured in each analysis type



Optimization Notice



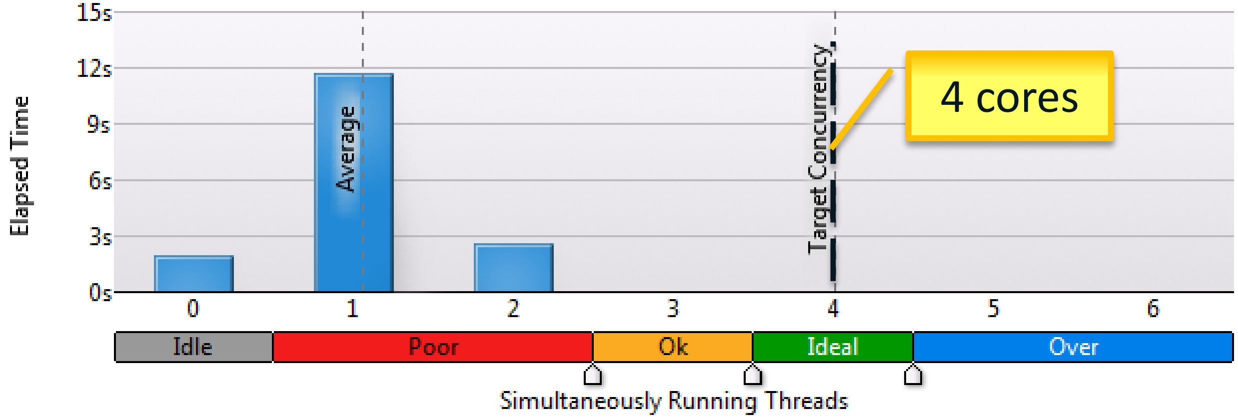
Optimization Notice

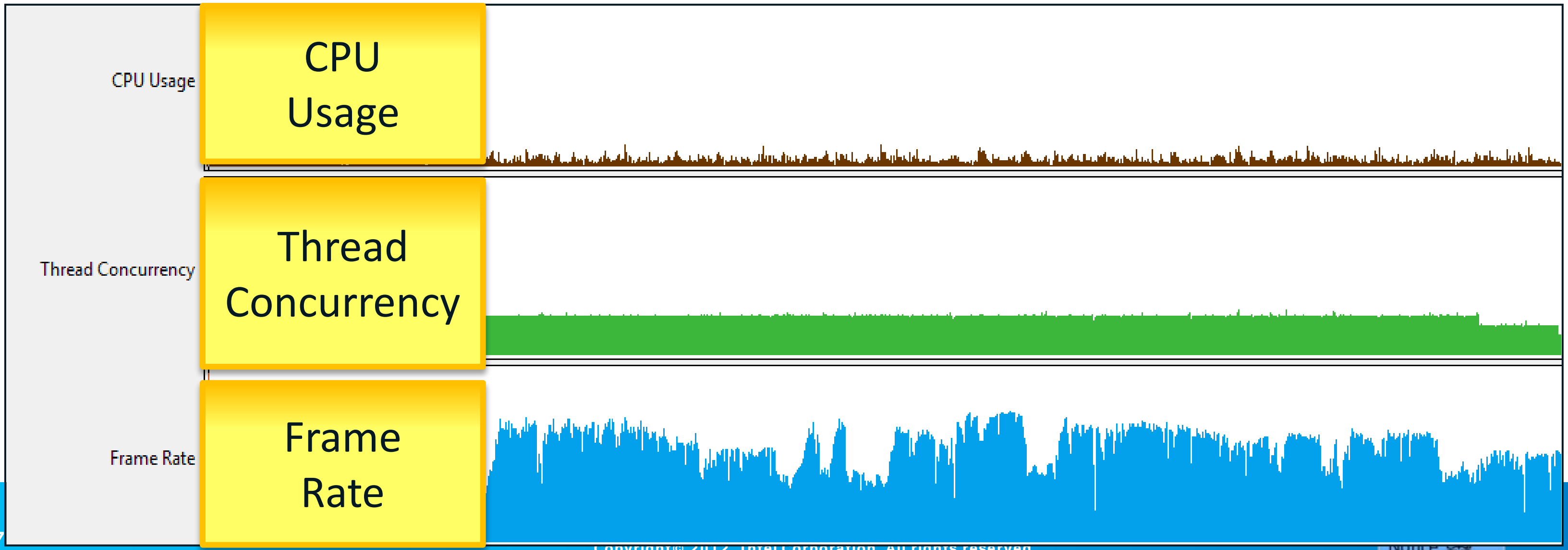
Intel® VTune™ Amplifier XE

Get a quick snapshot

Thread Concurrency Histogram

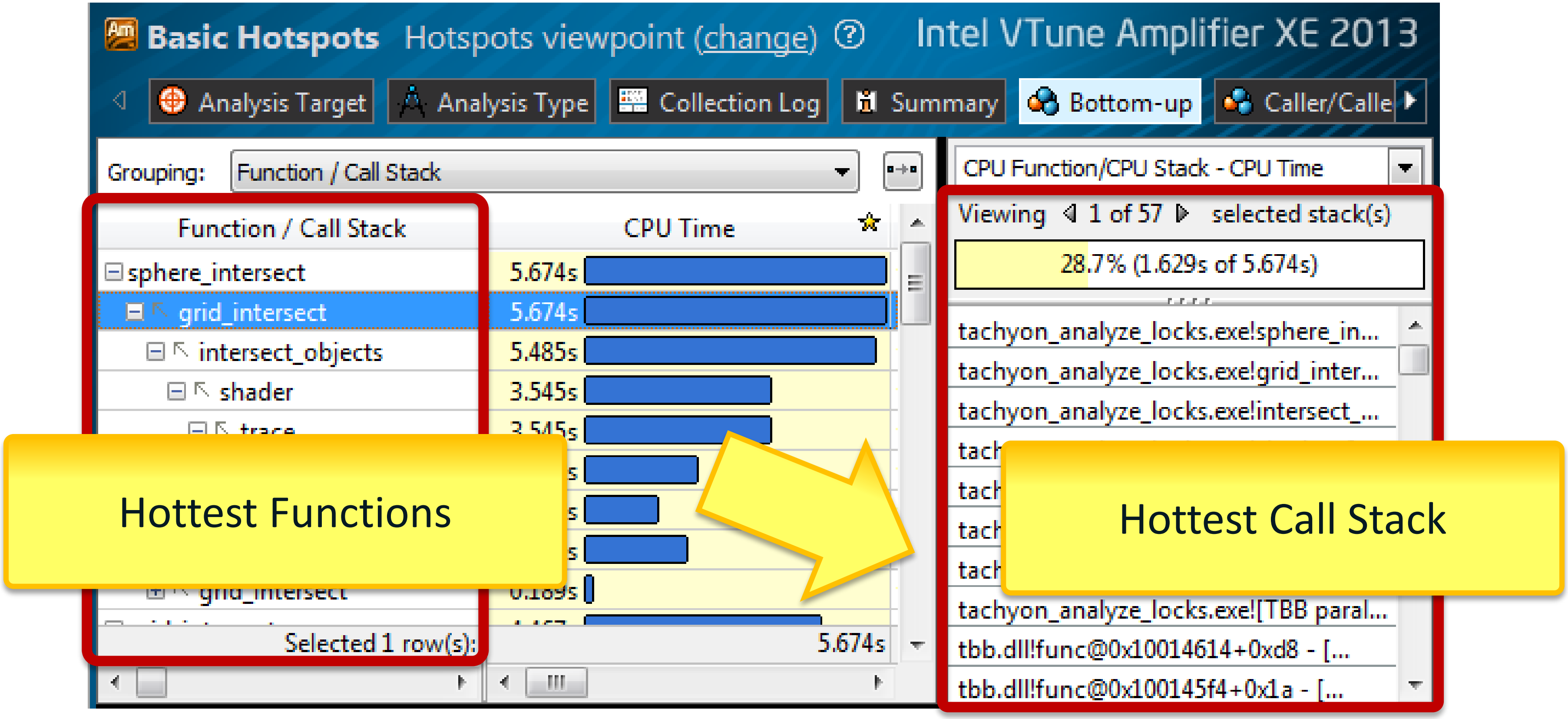
This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrer that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.





67

Intel® VTune™ Amplifier XE
Identify hotspots



Quickly identify what is important

Optimization Notice

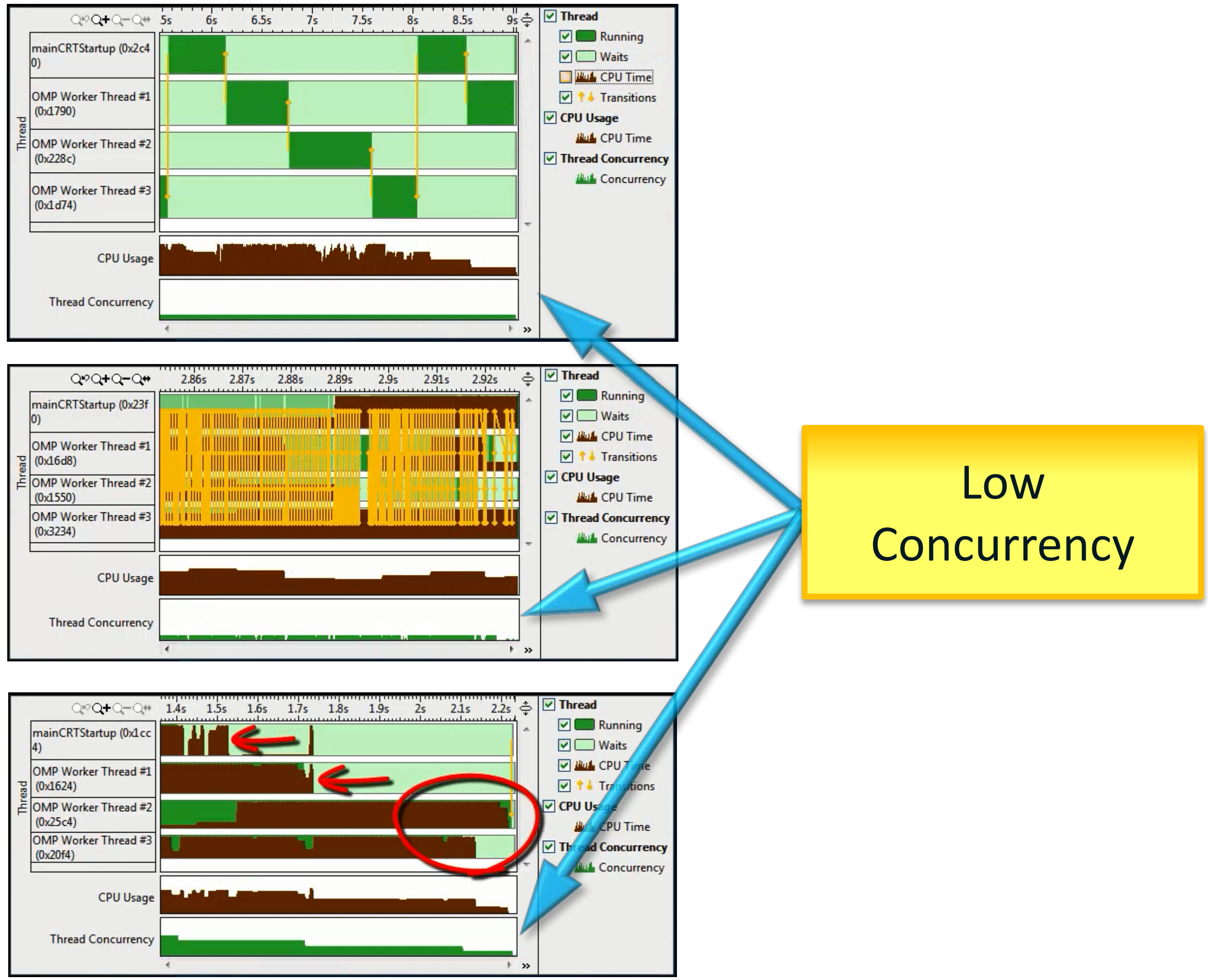
Intel® VTune™ Amplifier XE

Identify threading inefficiency

Coarse Grain Locks

High Lock Contention

Load Imbalance



Optimization Notice

Intel® VTune™ Amplifier XE

Find Answers Fast

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

… (Partial list shown)

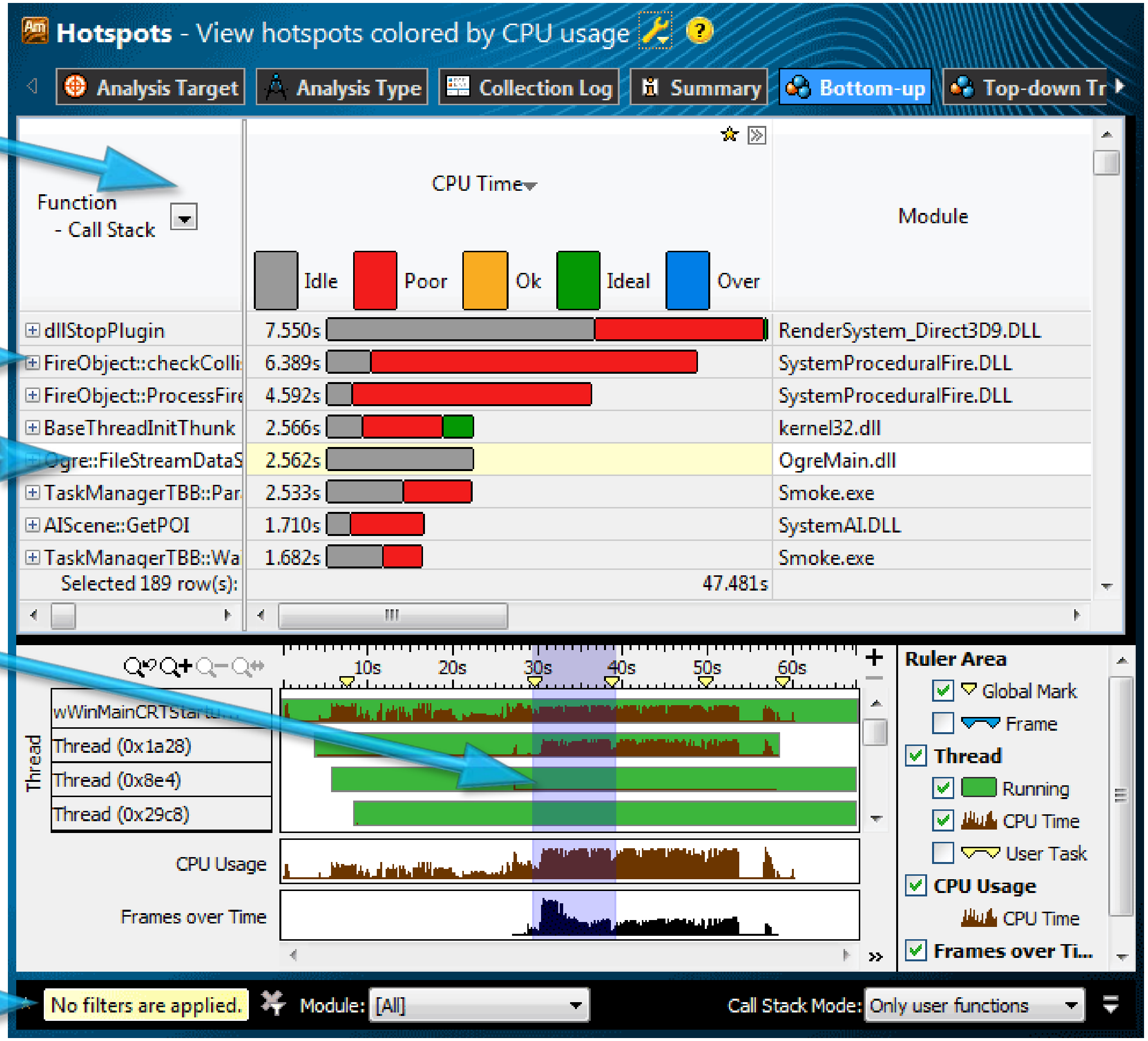
Click [+] for Call Stack

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

- Zoom In And Filter On Selection
- Filter In by Selection
- Remove All Filters

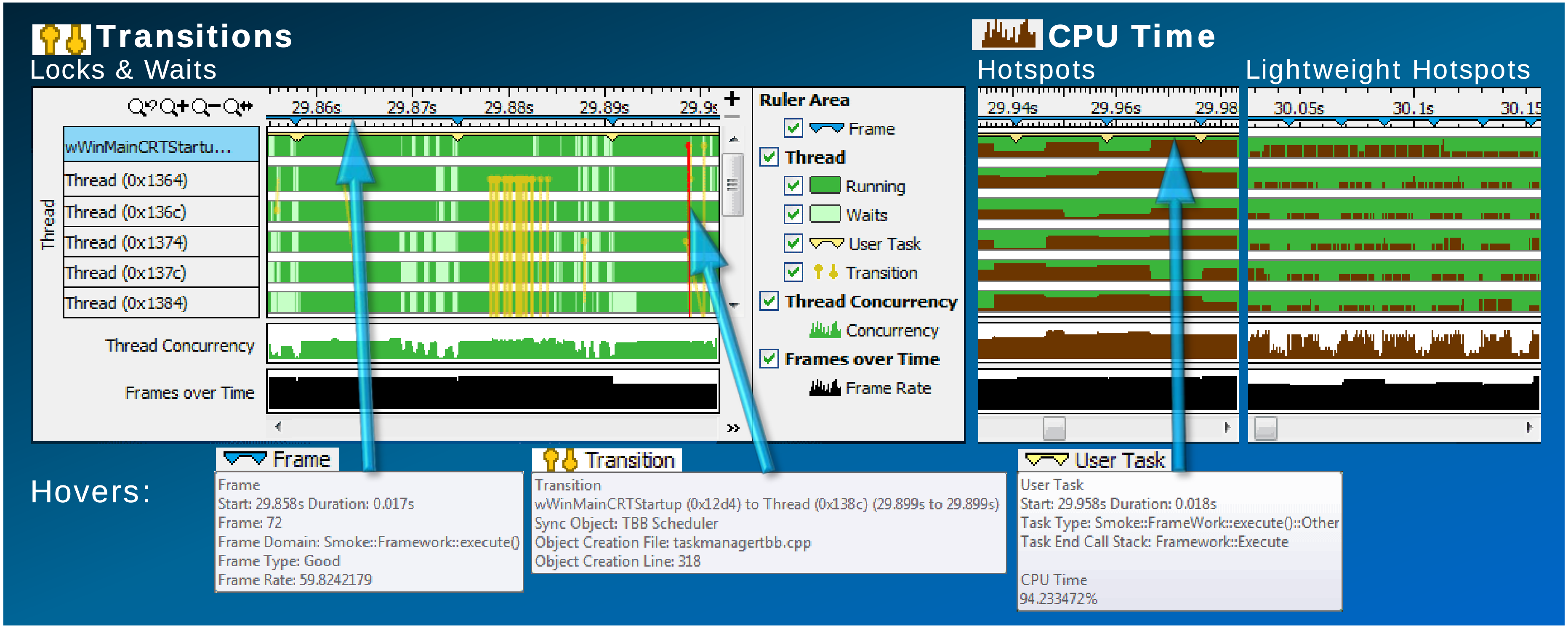
Filter by Module & Other Controls



Optimization Notice

Intel® VTune™ Amplifier XE

Timeline Visualizes Thread Behavior

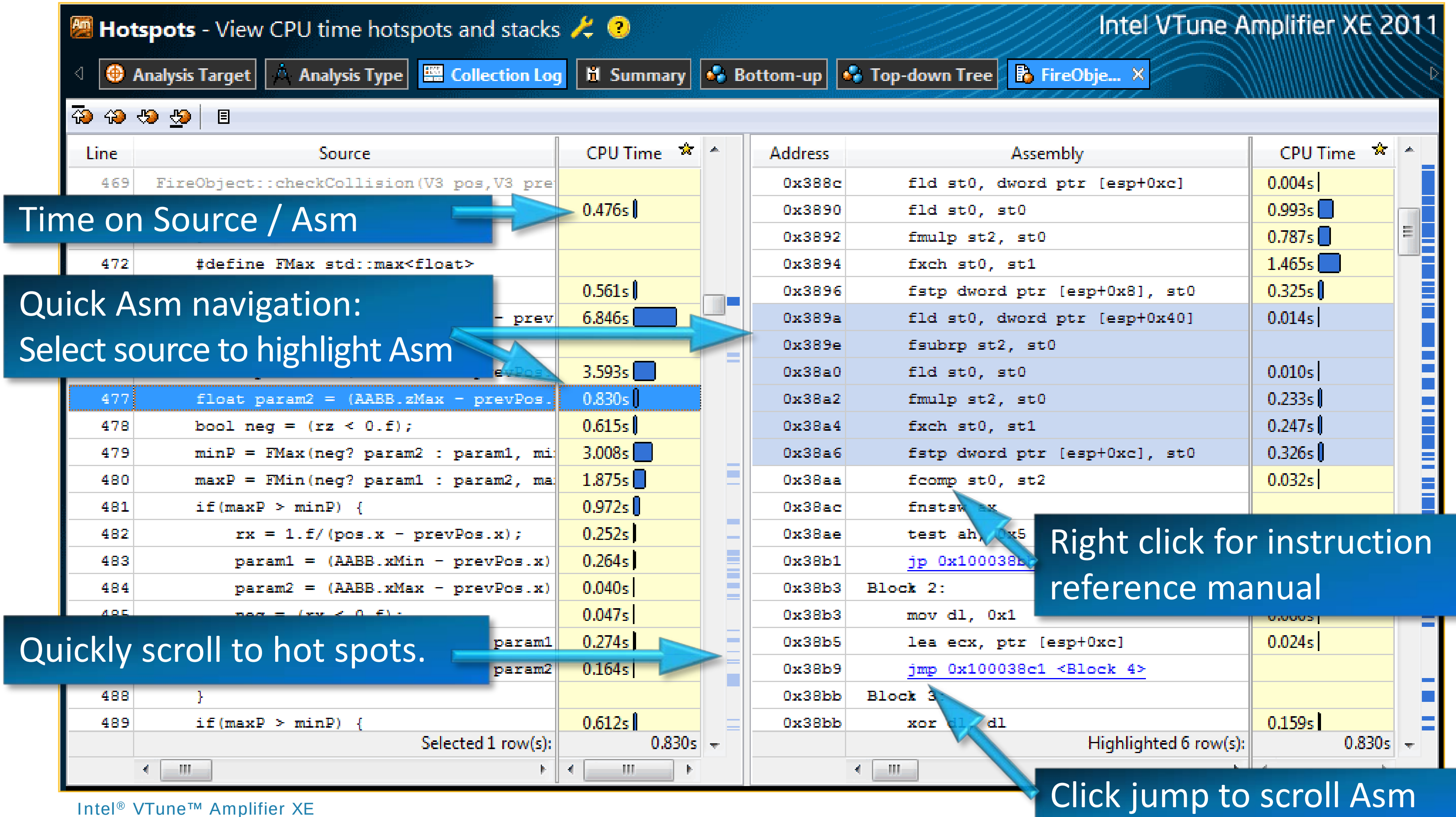


Optional: Use API to mark frames and user tasks

Optional: Add a mark during collection

Intel® VTune™ Amplifier XE

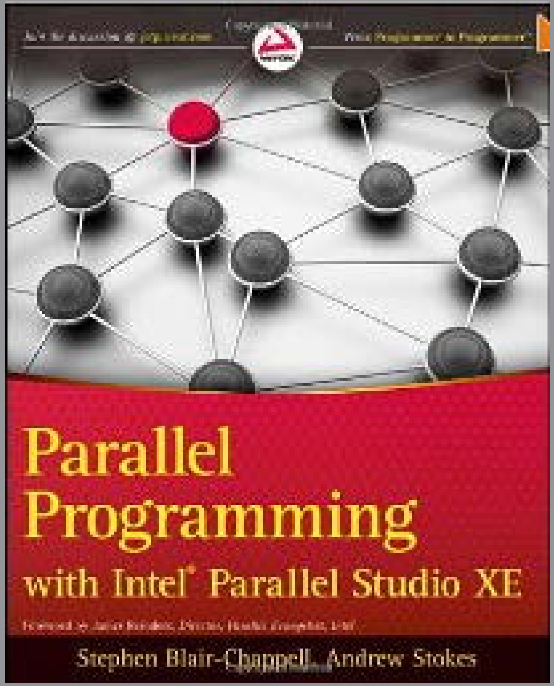
See Profile Data On Source / Asm



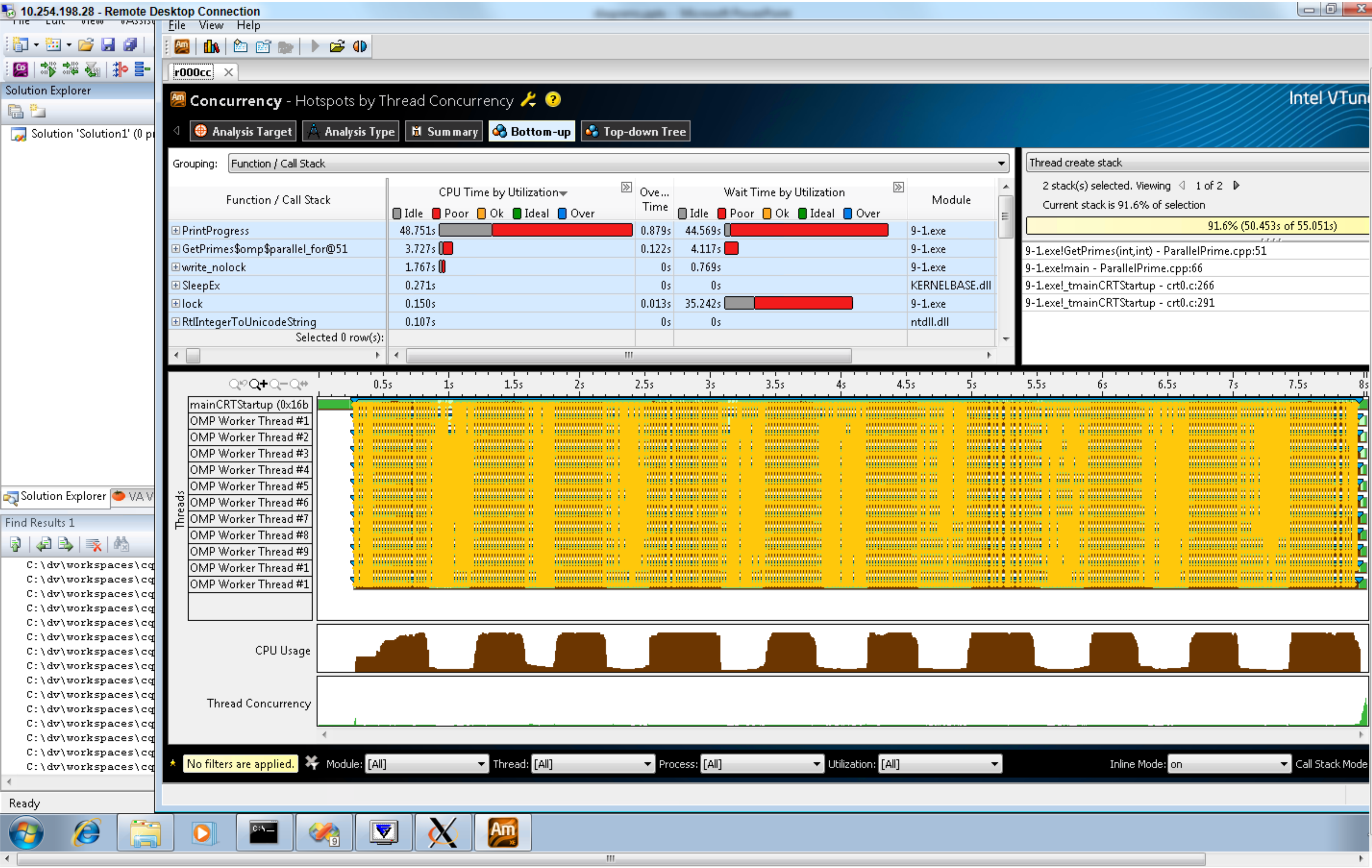
Intel® VTune™ Amplifier XE

Hands-on Lab

Lab 1 , Step 4
Tuning
Activity 9-1,9-2 & 9-3

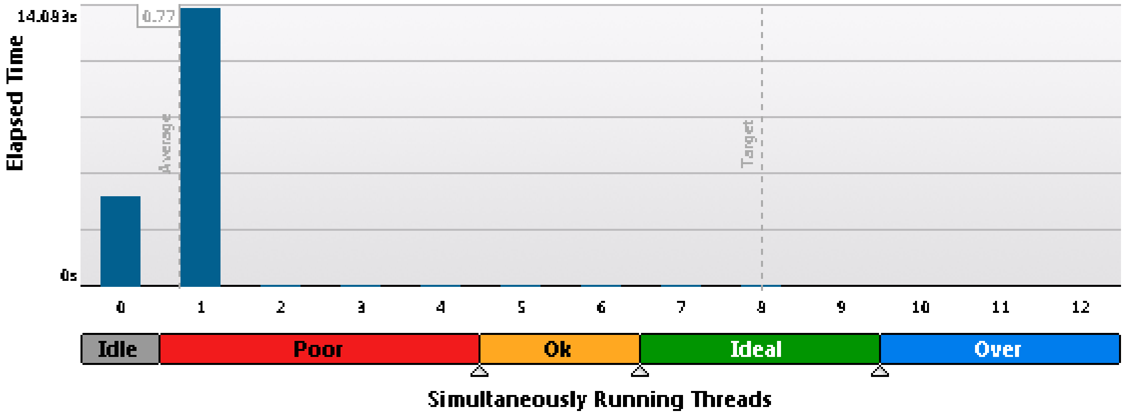


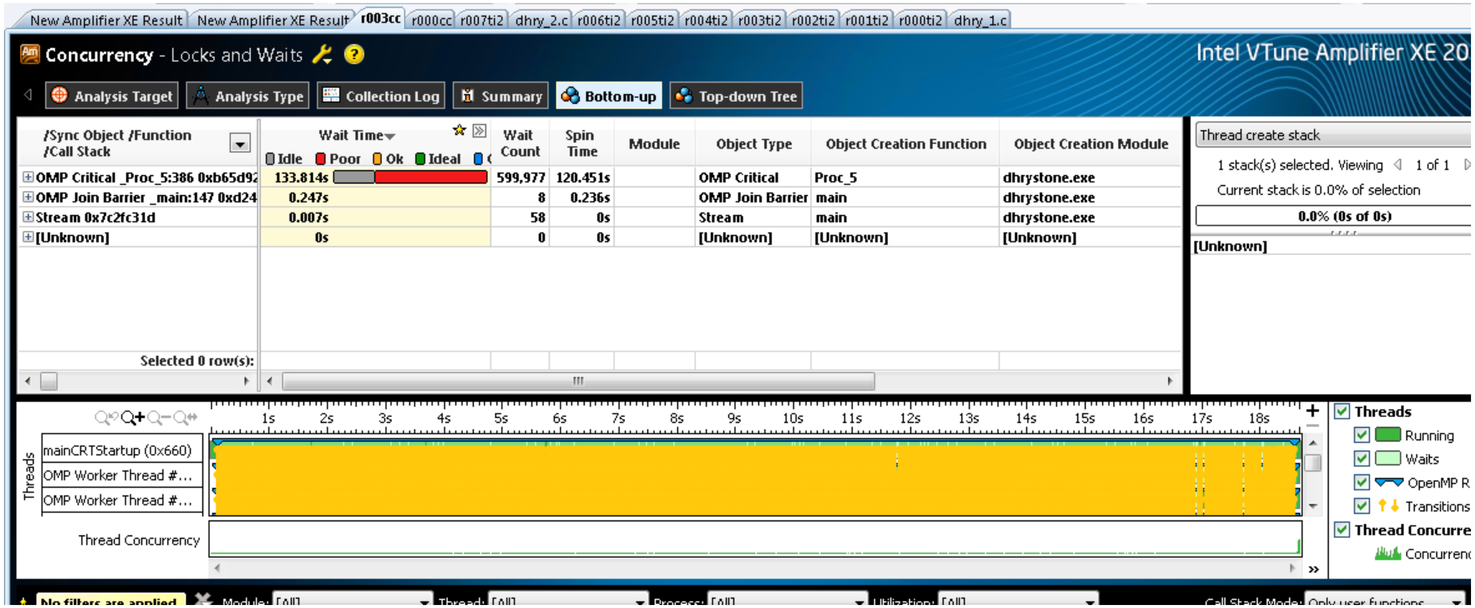
C:\CLASSFILES\Lab Sources\IPS Book\Chapter 9



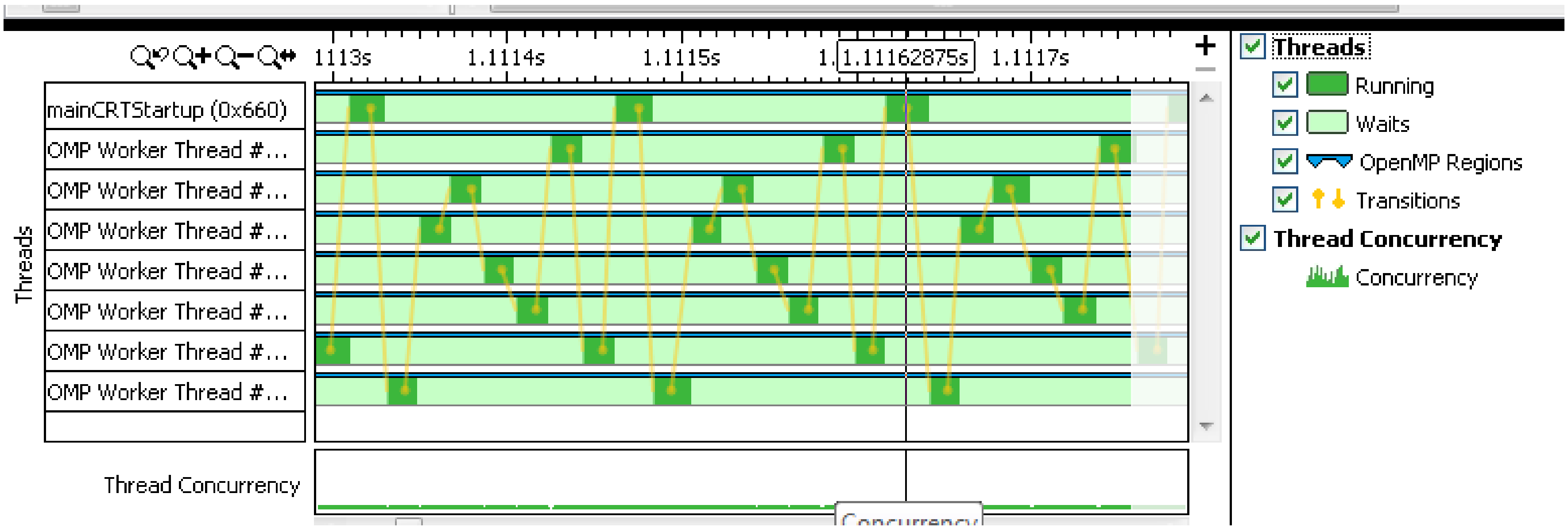
Optimization Notice

Concurrency and Performance of Attempt 1



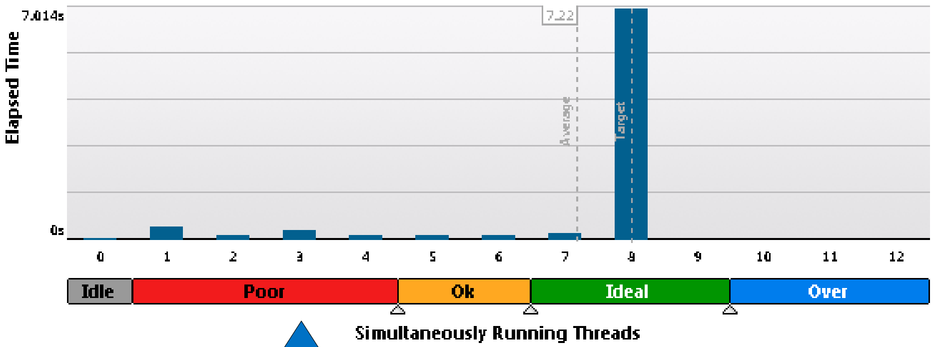


Zoom-in on time line shows reason for poor concurrency

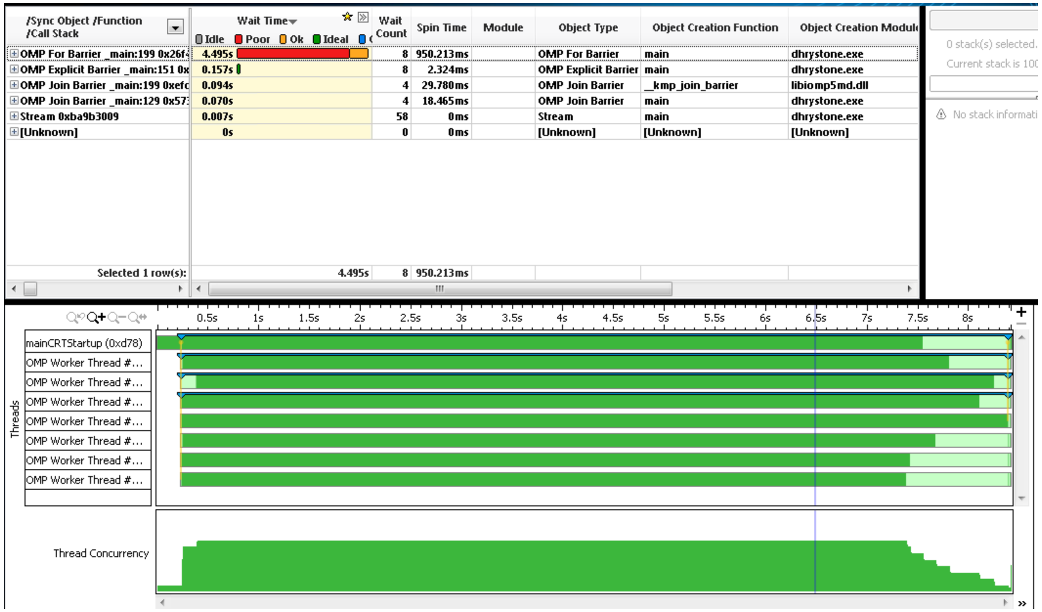


Optimization Notice

Concurrency and Performance of Attempt 2



If you have time, compare two sets of Results from the lab you have just finished



Optimization Notice

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®