



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

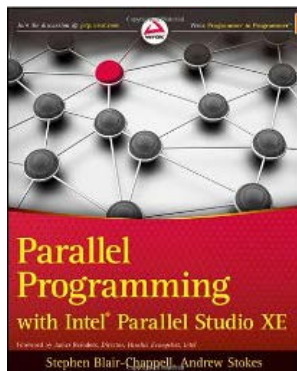
- 1: Parallelism Today
- 2: An Overview of Parallel Studio XE
- 3: Parallel Studio XE for the Impatient

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

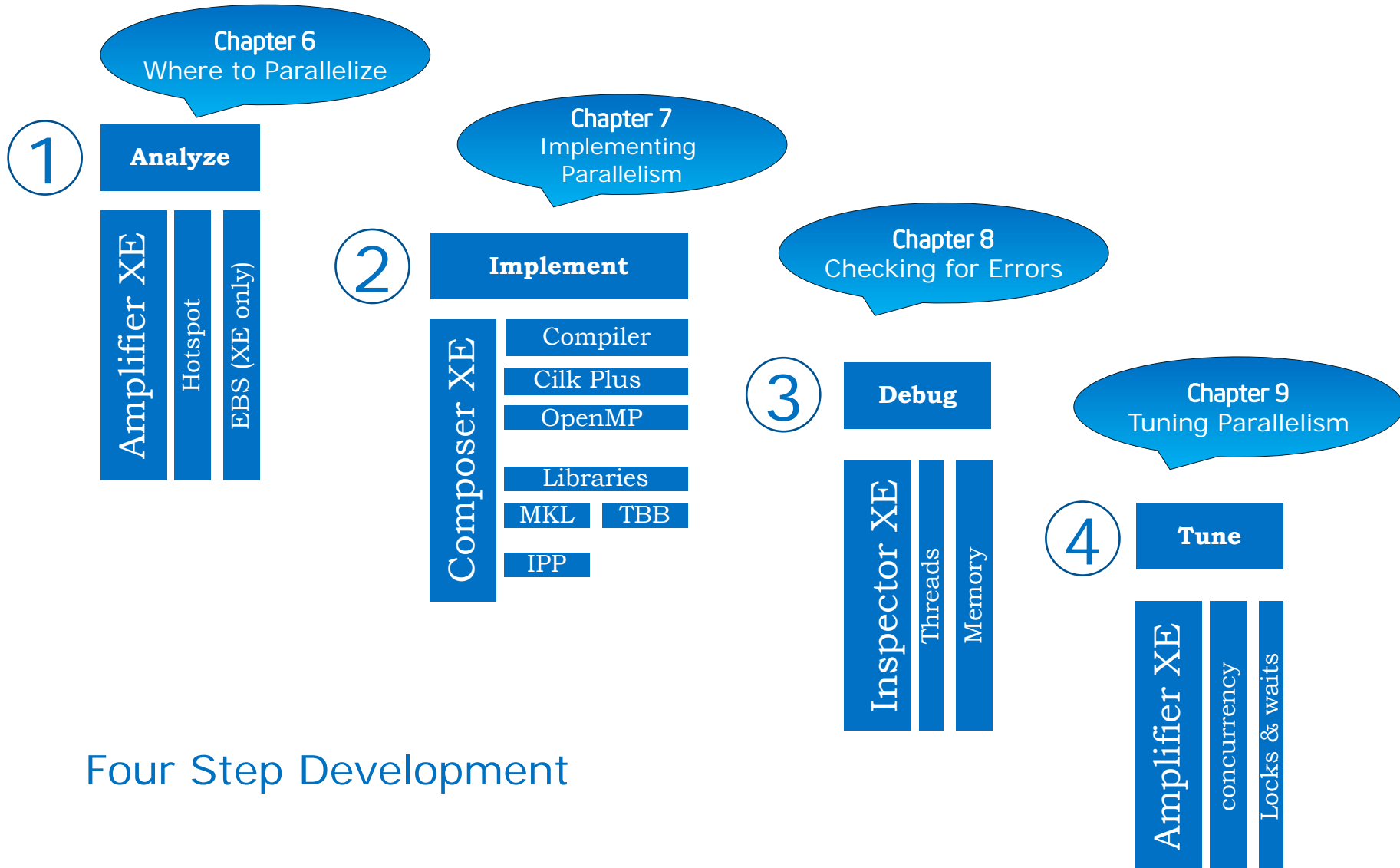
Part III :Case Studies

- 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



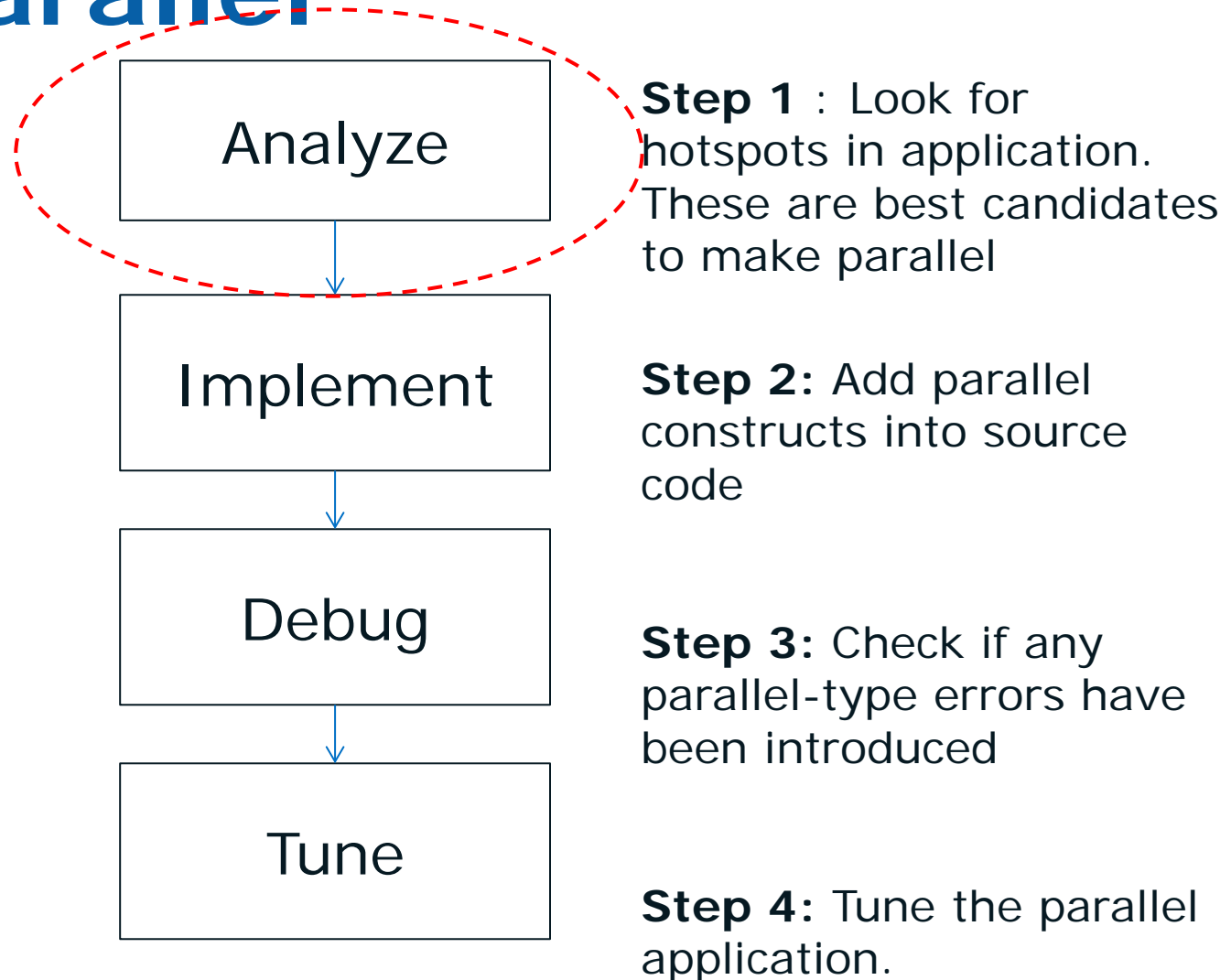
Speedup using parallelism

Parallel Code



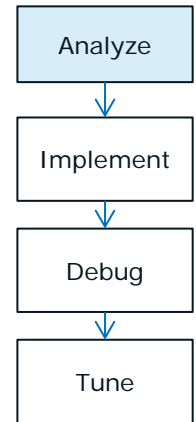
Four Step Development

Steps in moving from Serial to Parallel



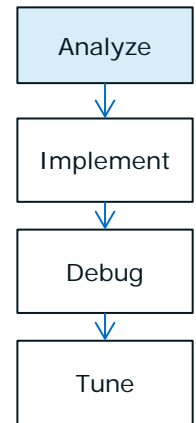
Key Questions - Analyze

- 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**



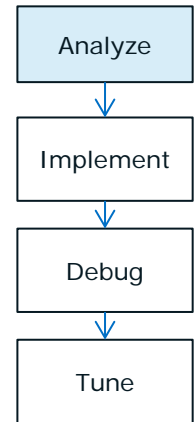
Using the Auto-Parallelizer

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)
```



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

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

** 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.*

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

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Source Location
Total	100.0%	3.3281s	0s	
RegisterWaitForInputIdle	100.0%	3.3281s	0s	
_tmainCRTStartup	100.0%	3.3281s	0s	crtd0.c:327
main	100.0%	3.3281s	0s	duplo.cpp:304
Duplo::run [loop]	89.2%	2.9688s	0s	duplo.cpp:239
Duplo::run [loop]	88.2%	2.9364s	0s	duplo.cpp:243
Duplo::run	88.2%	2.9364s	0s	duplo.cpp:245
Duplo::process [loop]	59.6%	1.9834s	0s	duplo.cpp:103
Duplo::process [loop]	59.1%	1.9677s	0s	duplo.cpp:106
Duplo::process	59.1%	1.9677s	0.7496s	duplo.cpp:106
SourceLine::equals	20.6%	0.6866s	0.6866s	sourceLine.cpp:48
SourceFile::getLine	16.0%	0.5316s	0.5316s	sourcefile.cpp:168

Line	Source	Total Time	%	Loop Time	%
237					
238	// Compare each file with each other				
239	for(int i=0;i<(int)sourceFiles.size();i++)			2.969s	0
240	std::cout << sourceFiles[i]->getFileName()	0.001s			
241	int blocks = 0;				
242					
243	for(int j=0;j<(int)sourceFiles.size();			2.936s	0
244	if(i > j && !isSameFilename(source	0.016s			
245	blocks+=process(sourceFiles[i]	2.921s	0		
246	}				

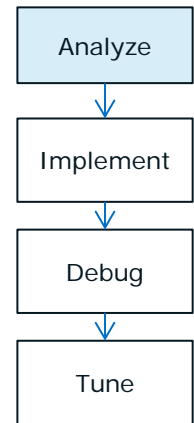
Selected (Total Time): 0s

Call Stack with Loops

- Duplo::run - duplo.cpp:239
- _tmainCRTStartup - crtd0.c:327
- RegisterWaitForInputIdle

Top down list of call sites, lines, & loops

Use Amplifier XE



The screenshot displays the Visual Studio IDE with the 'Source' window showing C++ code and the 'CPU time' window showing performance analysis.

Line	Source	CPU Time
29	<code>float xinc = (float)deltaX/(maxI</code>	
30	<code>float yinc = (float)deltaY/(maxJ</code>	
31	<code>for (int i=0; i<maxI; i++) {</code>	
32	<code>for (int j=0; j<maxJ; j++) {</code>	
33	<code>SetZ(i, j, xinc, yinc);</code>	
34	<code>}</code>	
35	<code>}</code>	
36	<code>}</code>	
37		
38	<code>void SetZ(int i, int j, float xin</code>	

The CPU time window shows the following stack trace:

- 100.0% (3.014s of 3.014s)
- 6-4b.exe!CalcMandelbrot - mandelbrot.cpp
- 6-4b.exe!SetZ - mandelbrot.cpp:42
- 6-4b.exe!Mandelbrot - mandelbrot.cpp:33
- 6-4b.exe!main - main.cpp:15
- 6-4b.exe!_tmainCRTStartup - crt0.c:266
- kernel32.dll!BaseThreadInitThunk+0x1f56c - ..
- ntdll.dll!RtlUserThreadStart+0x33280 - [Unkn..

Advantages & Disadvantages of different hotspot methods

Method	Pro	Con
Loop Profiler	<ul style="list-style-type: none">• Easy to use• All you need is in compiler package• Profiles loops as well as functions	<ul style="list-style-type: none">• Very basic• Code needs instrumenting• No call tree• No comparison function
Auto-parallelizer	<ul style="list-style-type: none">• Easy to use• Quick way of spotting right place• Comparison relatively easy	<ul style="list-style-type: none">• Easily confounded• IPO obscures loops
Amplifier XE	<ul style="list-style-type: none">• Small overhead• Easy to traverse call stack• No special build needed• Multiple options for collection \viewing• Results can be compared	<ul style="list-style-type: none">• (No loop profiler)• No <i>precise</i> call graph

Profile the System

- [-] Algorithm Analysis
 - Advanced Hotspots

Profile Applications

- [-] Algorithm Analysis
 - Basic Hotspots
 - Concurrency
 - Locks and Waits

VTune Amplifier XE

User Mode

- Works on
- Intel
 - non-Intel
 - More overhead than lightweight hotspots

Architectural Analysis

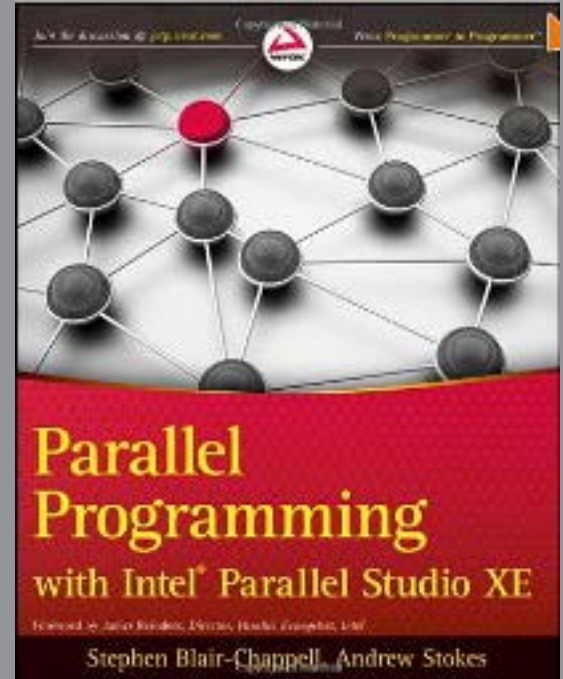
- [-] Advanced Intel(R) Microarchitecture Code Name Sandy
 - General Exploration
 - Bandwidth
 - Access Contention
 - Branch Analysis
 - Client Analysis
 - Core Port Saturation
 - Cycles and uOps
 - Loop Analysis
 - Memory Access
 - Port Saturation

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

/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 **slower** ☹️
- 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.

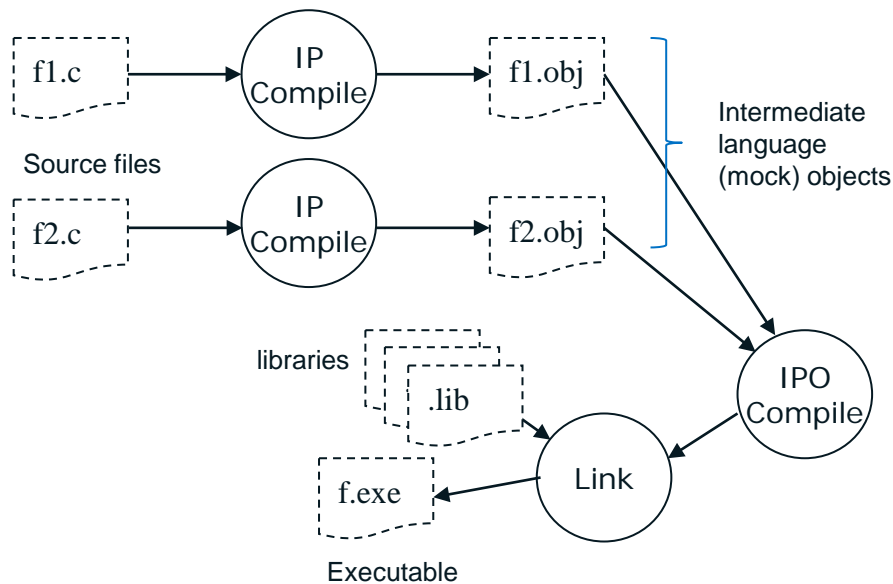
Step 2

/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

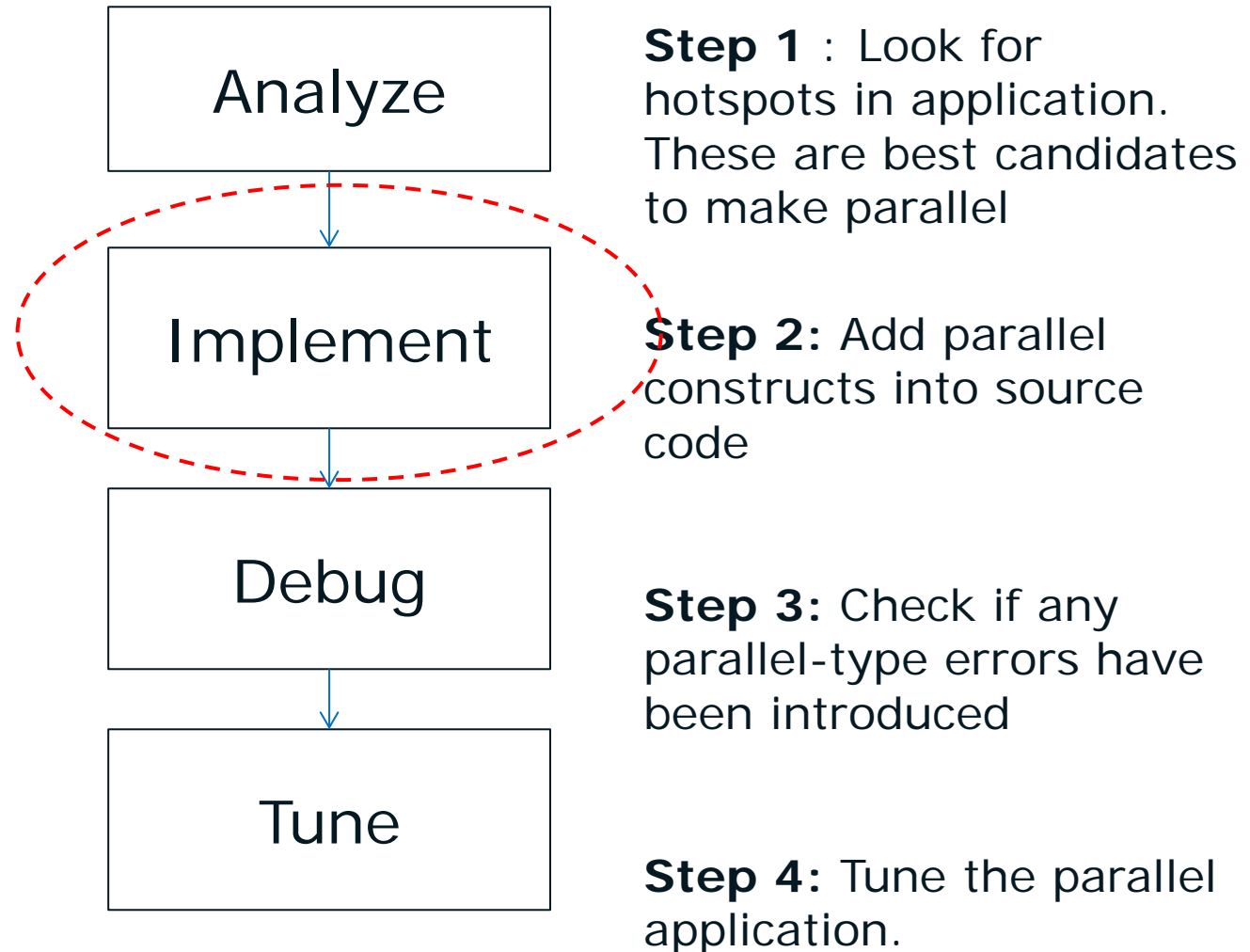
Interprocedural Optimisation



- **/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

Steps in moving from Serial to Parallel



Key Questions – Implement

What's the best method?

How much programming effort is required?

Is my code still working?

Language to help parallelism

Parallel Code

Intel® Cilk™ Plus

OpenMP

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

Intel® Threading Building Blocks

Intel® MPI

Fortran Coarrays

OpenCL

Native Threads

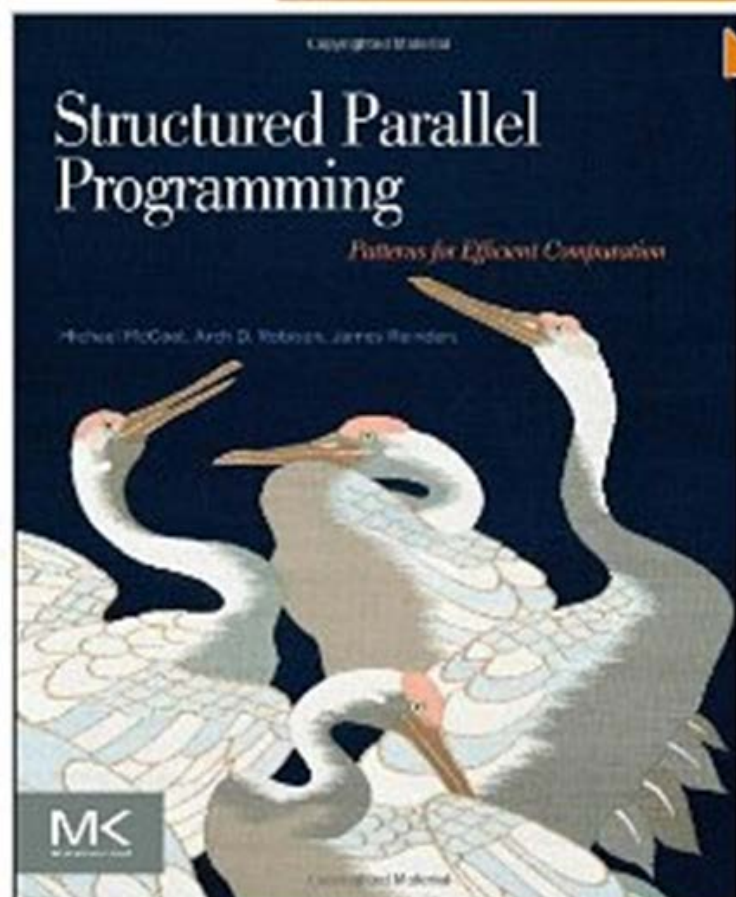
```
cilk_for (int i = 0; i < max_row; i++)
{
    for (int j = 0; j < max_col; j++ )
    {
        p[i][j] = mandel( complex(scale(i), scale(j)));
    }
}
```

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.



Introduction to Concurrency in Programming Languages

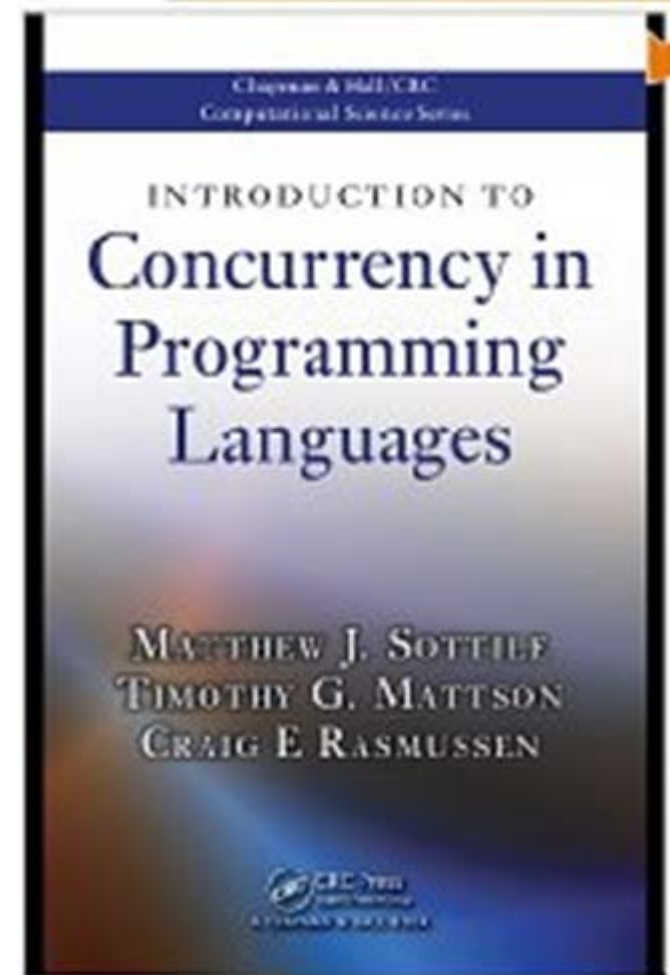
Matthew J. Sottile, Timothy G. Mattson, Craig E. Rasmussen

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.



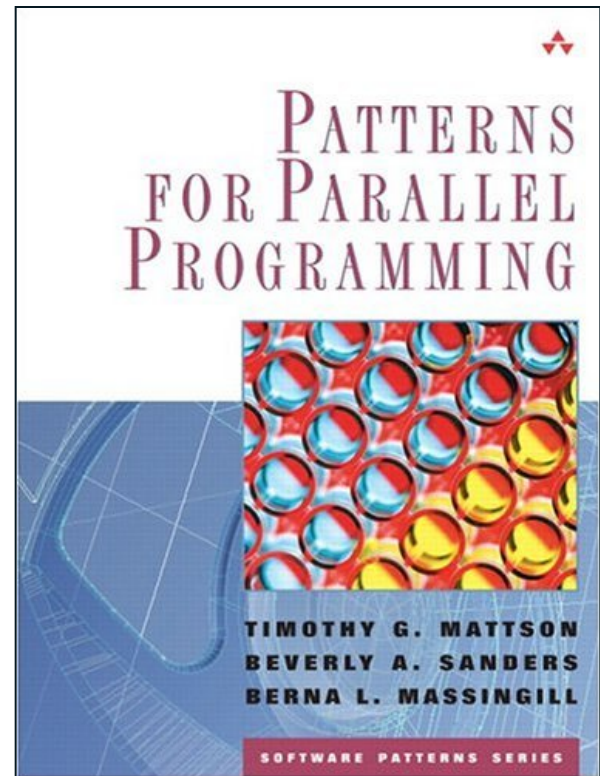
Patterns for Parallel Programming

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



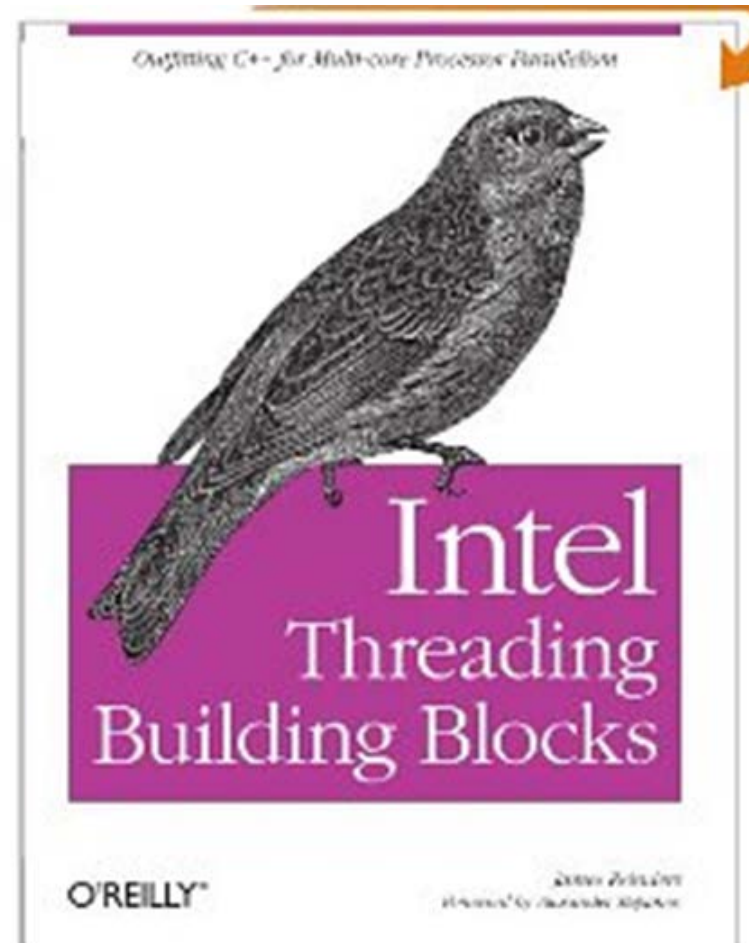
Intel Threading Building Blocks

James Reinders,

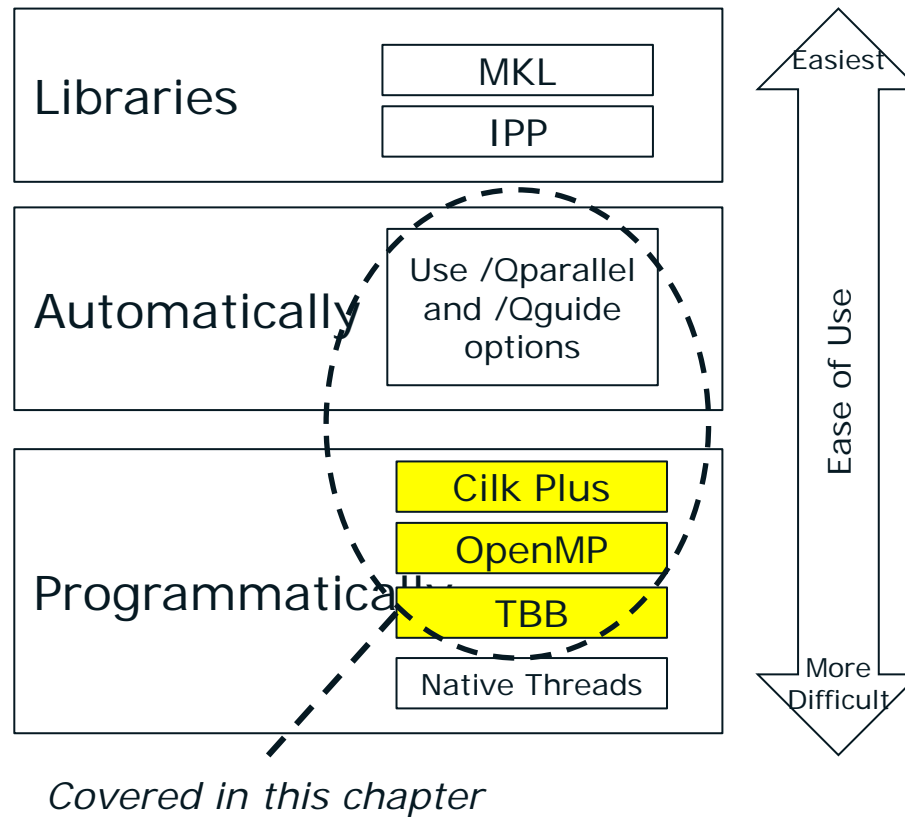
O'Reilly Media; 1 edition (19 July 2007) ISBN-10: 0596514808

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.



Scope of Chapter 6



What Is OpenMP?

Portable, shared-memory threading API

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

Standard

<http://www.openmp.org>

Standard

Support

Current spec is OpenMP 4.0

Standard

(combined C/C++ and Fortran)

directed threading experience

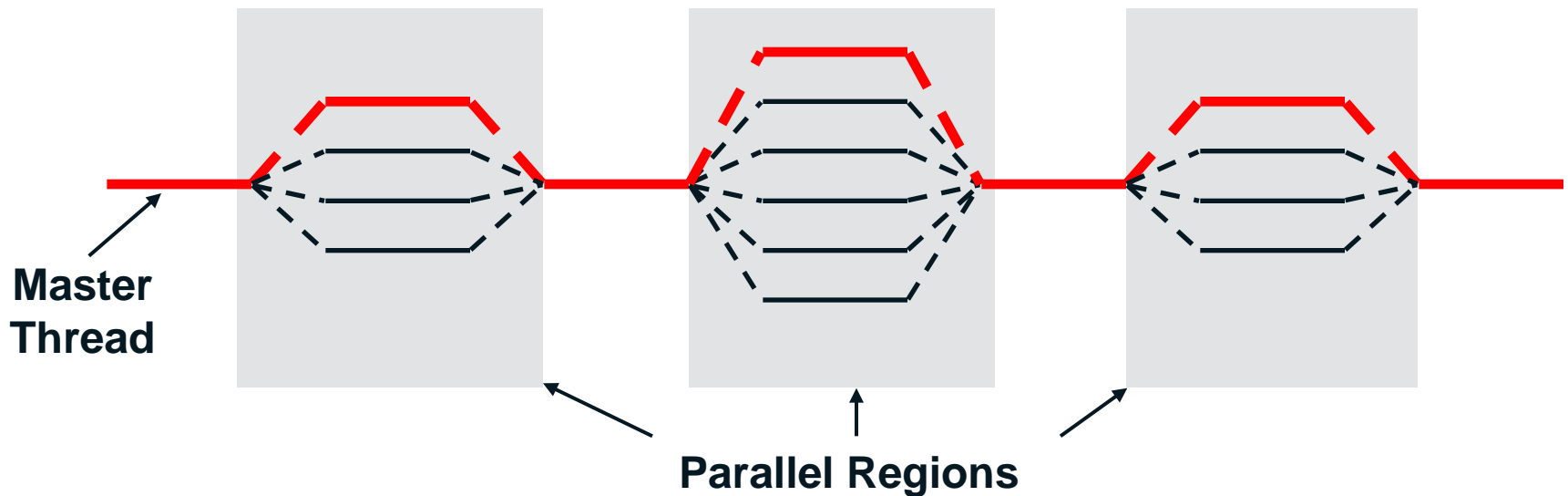
Combines serial and parallel code in single source

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

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

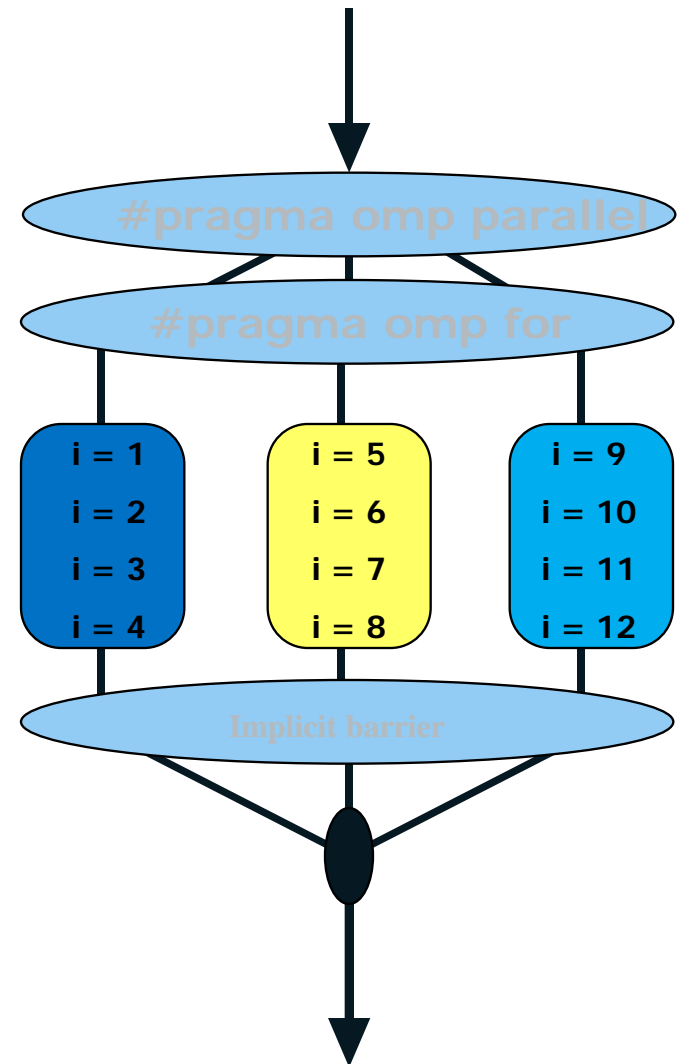


omp for Construct

```
// 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

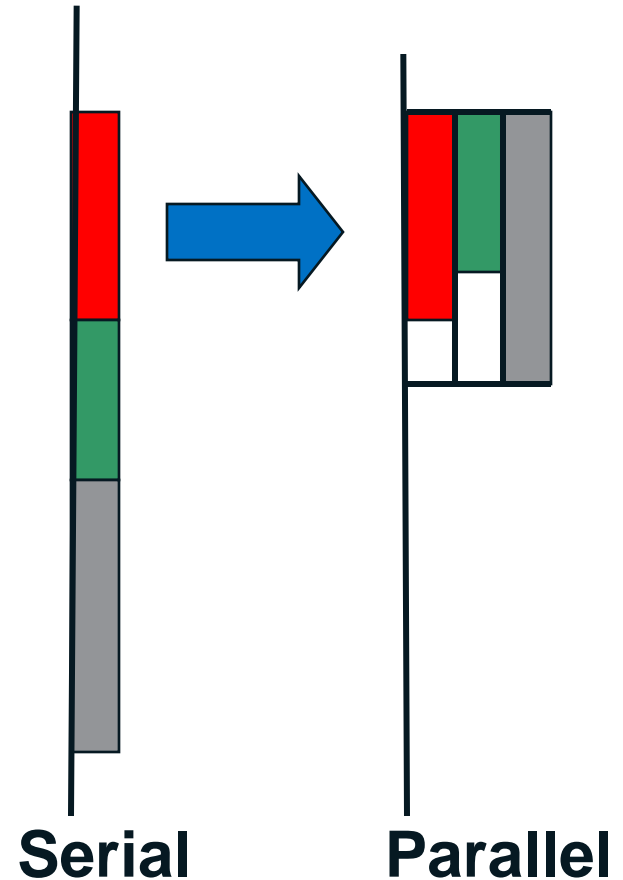
Threads must wait at the end of work-sharing construct



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();  
}
```



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

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

```
1: #include <iostream>
2: #include <algorithm>
3: using namespace std;
4: int main()
5: {
6:     char Message[]="The Beauty of Lambda!";
7:
8:     int NumSpaces = 0;
9:     for_each(
10:         Message,          //
11:         Message + sizeof(Message),
12:         [&NumSpaces] (char c) { if (c == ' ') NumSpaces++;}
13:     );
14:     cout << "There are " << NumSpaces <<" spaces in: '"<<
15:     Message << "' " <<endl;
16: }
```

STL template for_each

```
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);
}
```

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

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

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

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

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]=
        sqrt(( 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]=
            sqrt(( 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

Parallelizing Linked Lists

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

```
void RunThroughLinkedList()
{
    #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 RunThroughLinkedList()
{
    node *pHead = Head;
    while( pHead != NULL)
    {
        cilk_spawn Work( pHead);
        pHead = pHead-> next;
    }
}
```

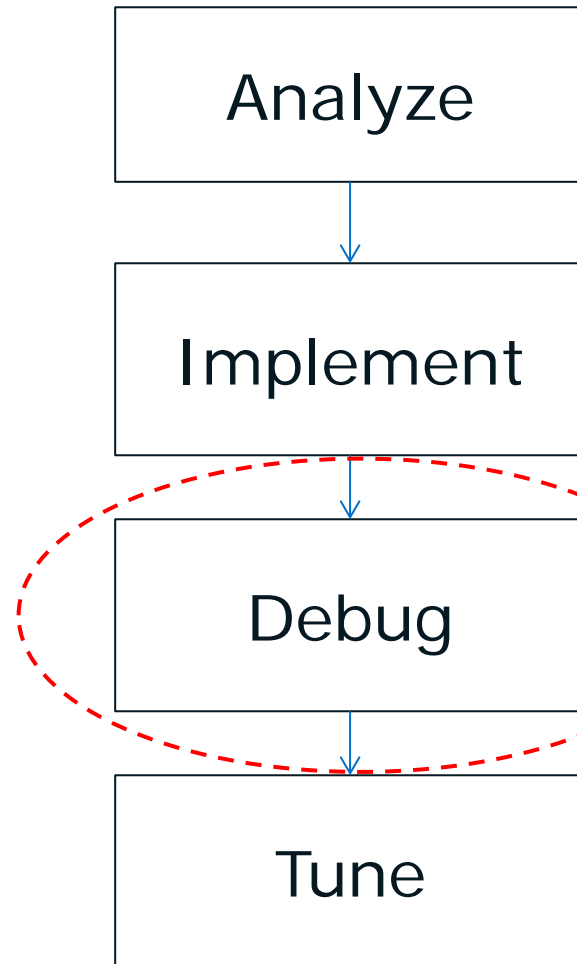
Key

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.

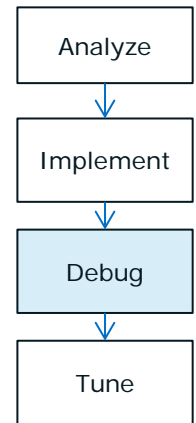
Key Questions - Verify

Is the parallelism correct?

Do I have deadlocks or data races?

Do I have memory errors?

Does my program still work as intended?

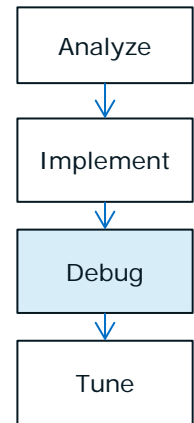


Ways to Find your Parallel Errors

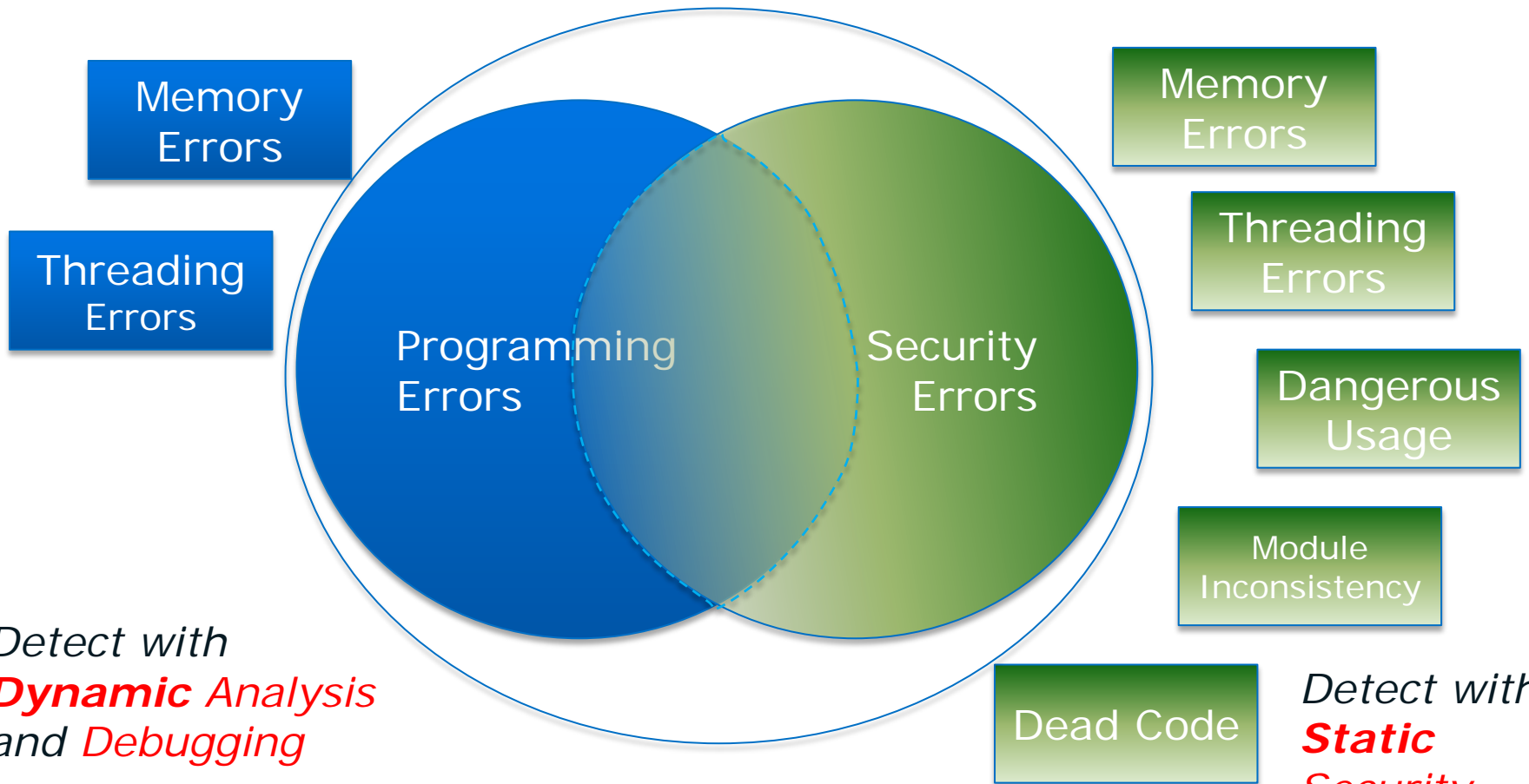
1. Use **Inspector XE**

2. Use **Advisor**

3. Use Intel-provided **Data Race Detector** in GDB



Dynamic vs Static Error Detection



Detect with
Dynamic Analysis
and **Debugging**

Detect with
Static Security Analysis

Intel Inspector XE
and/or
Data Race Detector in GDB

Intel Compiler
and
Intel Inspector XE

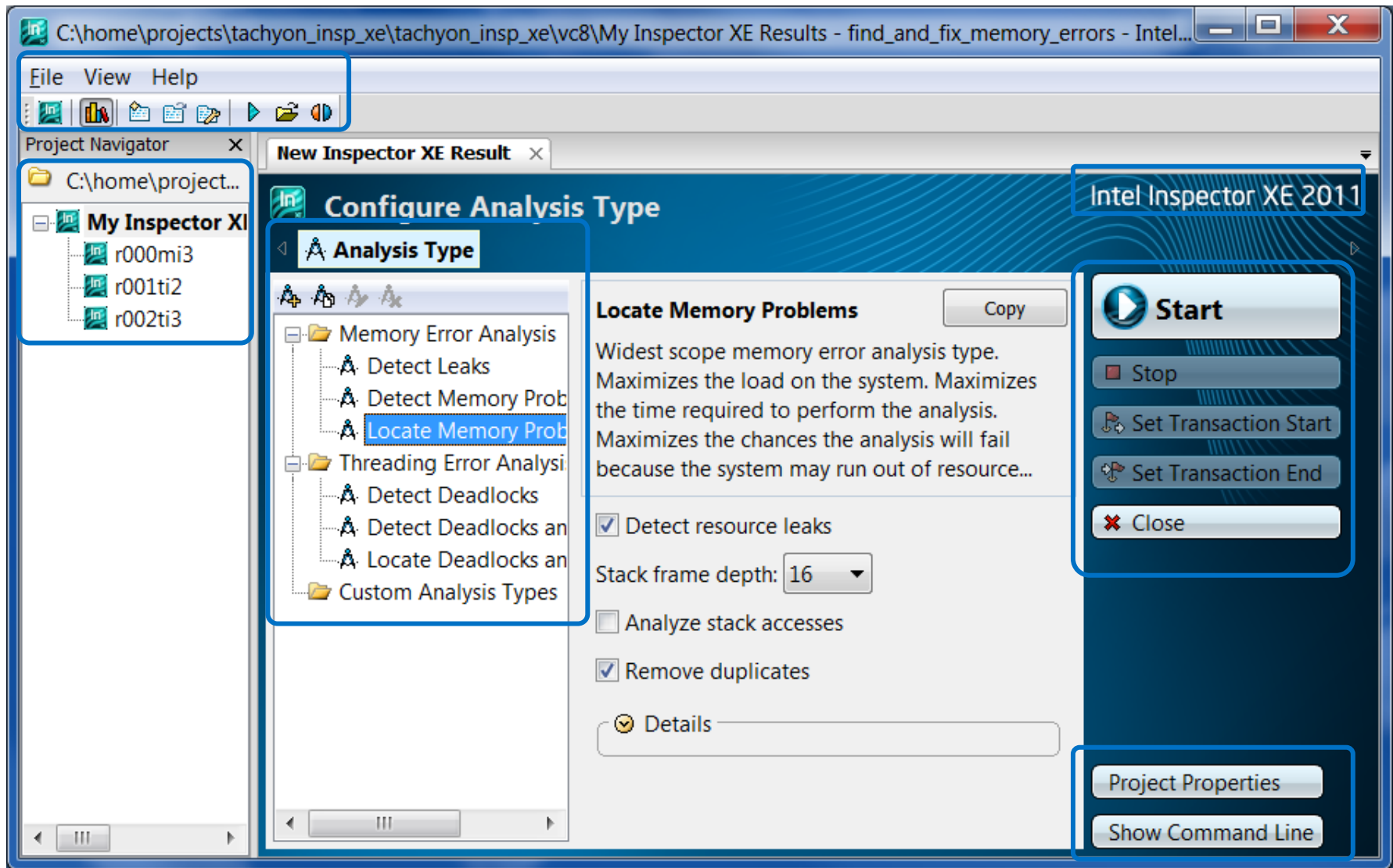
Inspector XE – Threading Errors & warnings

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

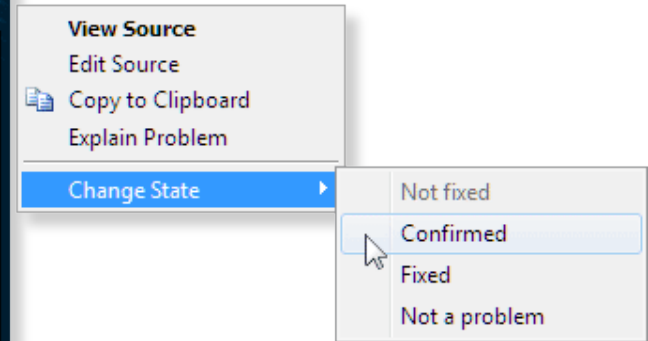
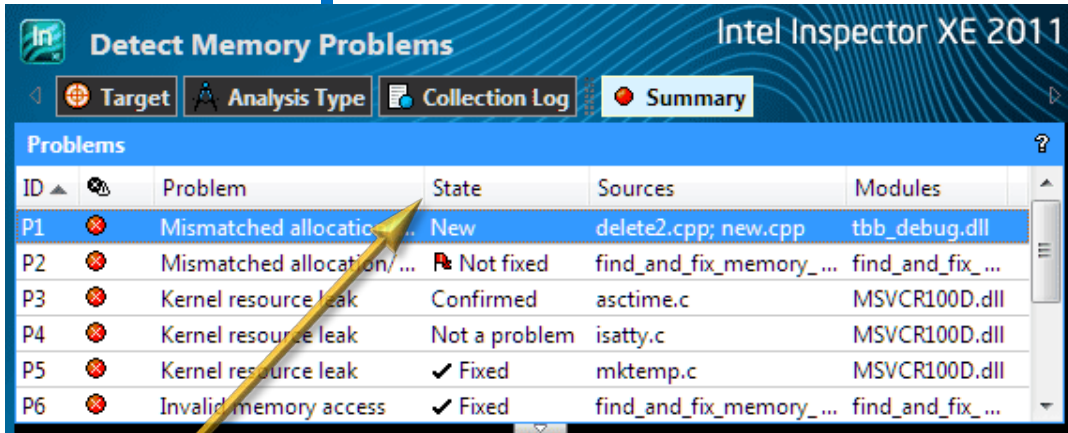
Inspector XE – Memory Errors & warnings

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

Inspector XE - Standalone GUI



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

ID	Description ▲	Source	Function	Module
X5	Read	winvideo.h:201	loop_once	find_and_fix_threading_errors.exe
X6	Read	winvideo.h:202	loop_once	find_and_fix_thr
X8	Read	winvideo.h:270	next_frame	find_and_fix_thr
X7	Write	winvideo.h:202	loop_once	find_and_fix_thr
X9	Write	winvideo.h:270	next_frame	find_and_fix_thr
X11	Write	winvideo.h:270	next_frame	find_and_fix_thr

Context menu options:

- Set as Related Code Location
- Set as Focus Code Location
- Edit Source
- Copy to Clipboard
- Explain Problem
- Suppress...

Be specific or select group of similar problems

	Problem	Code Location ...	Module/Function/Source/Line
<input checked="" type="checkbox"/>	Data race	Read	find_and_fix_threading_errors.exe!loop_once - winvideo.h:201
<input checked="" type="checkbox"/>	Data race	Write	find_and_fix_threading_errors.exe!next_frame - winvideo.h:270

General

Any problem
 Any description

Stack frame

find_and_fix_threading_errors.exe!loop_once - winvideo.h:201

Any module Any function Any source Any line

Suppressions are saved in one or more files

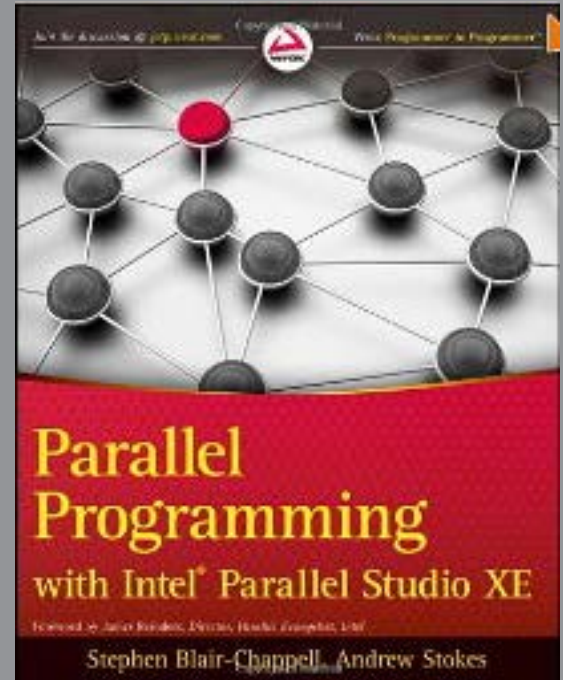
Tool suppresses all files from specified folder(s)

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

2x-20x Detect Leaks
10x-40x Detect Memory Problems
20x-80x Locate Memory Problems

Analysis Time Overhead

Detect Memory Problems Copy

Medium scope memory error analysis type. Increases the load on the system and the time and resources required to perform analysis. Press F1 for more details.

Analyze without debugger
Run an analysis and report all detected problems. Use to view correctness issues without stopping in the debugger to examine them.

Enable debugger when problem detected
Run an analysis under the debugger and stop every time a problem is detected. Use to allow investigation of every problem detected.

Select analysis start location with debugger
Run target application under the debugger with analysis disabled until you choose to turn on analysis. Before starting, set a code breakpoint to stop execution prior to where you want analysis to begin. Sele...

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

Detect Memory Problems

Target Analysis Type Collection Log Summary

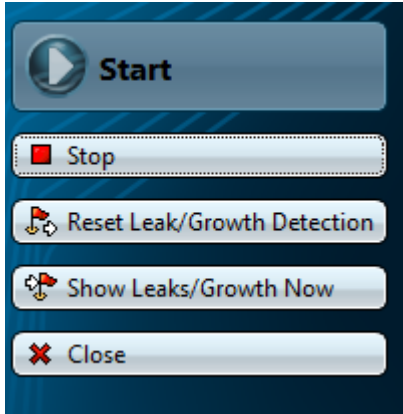
Problems	Type	Sources	Modules	Object ...	State
P	Memory leak	ixe_mem_growth.cpp	ixe_mem_growth.e...	144	New
P	Memory growth	[Unknown]; ixe_mem_gro...	Unknown; ixe_mem...	272	New
	Start memory growth det...	[Unknown]	Unknown		Not fixed
	Memory growth	ixe_mem_growth.cpp:7	ixe_mem_growth.e...	272	New
	End memory growth det...	[Unknown]	Unknown		Not fixed

Code Locations: Memory growth

Description	Source	Function	Module	Object Size	Offset
Allocation site	ixe_mem_growth.cpp:7	transaction	ixe_mem_growth.exe	272	
5	{				ixe_mem_growth.exe!transaction
6	char *str;				ixe_mem_growth.exe!main - ixe_m
7	str = (char*) malloc(16);				ixe_mem_growth.exe!_tmainCRTSta
8	}				ixe_mem_growth.exe!mainCRTStart
9					kernel32.dll!BaseThreadInitThun

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**

The screenshot shows the 'Detect Memory Problems' tool interface. The 'Problems' table lists two issues:

ID	Type	Sources	Modules	Object Size	State
P1	Memory leak	ixemem_growth.cpp	ixemem_growth.exe	192	New
P2	Memory growth	[Unknown]; ixemem_gr...	Unknown; ixemem_gr...	368	New

The 'Code Locations: Memory leak' section shows the following code snippet:

```
5 {
6   char *str;
7   str = (char*) malloc(16);
8
9   malloc(4);
```

The right side of the code editor shows the corresponding memory locations:

- ixemem_growth.exe!transaction
- ixemem_growth.exe!main - ixemem_growth.exe!_tmainCRTSt
- ixemem_growth.exe!mainCRTStar
- kernel32.dll!BaseThreadInitThu

Inspector APIs for memory growth and leak detection.

API	Purpose
<code>__itt_heap_growth</code>	memory growth detection
<code>__itt_heap_leaks</code>	on-demand leak detection
<code>__itt_heap_reset_detection</code>	reset the growth/leak baseline
<code>__itt_heap_record</code>	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

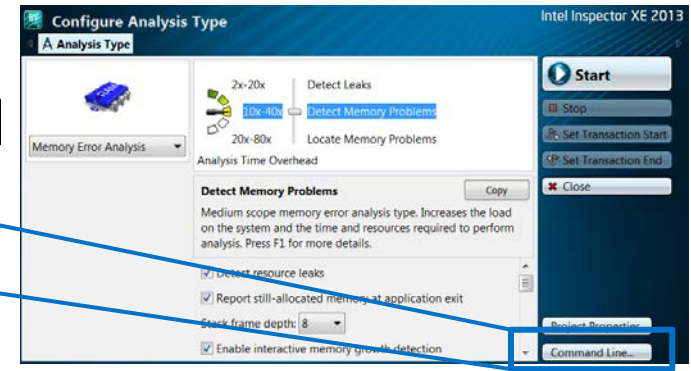
Command Line...

- 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

Reporting

To generate a report:


```
i nspxe-cl -R=<report-type> <results directory name>
```

Sample commands:

```
i nspxe-cl -report-list  
i nspxe-cl -report=summary  
i nspxe-cl -report=problems
```

Example:

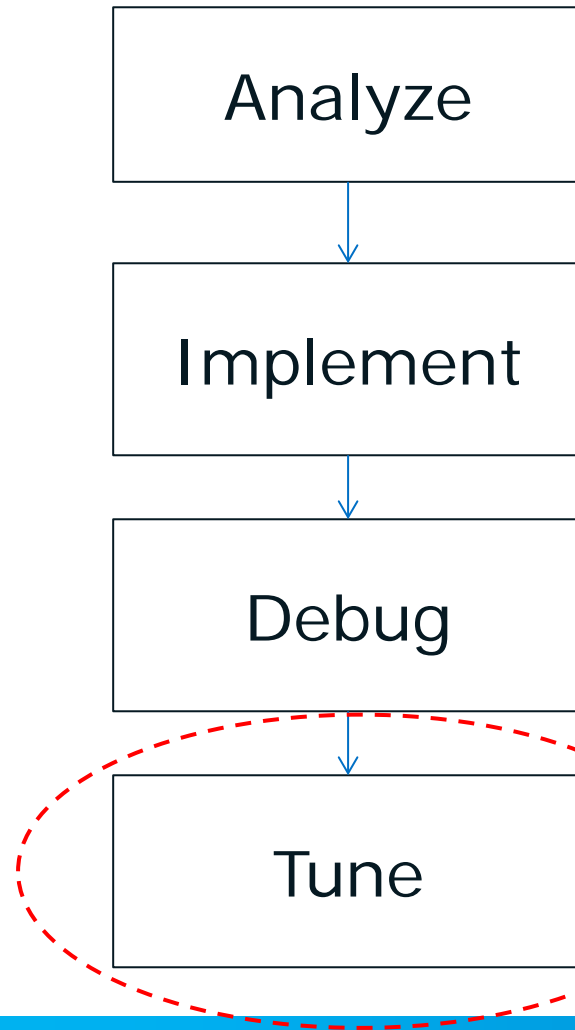
```
cd /home/user/testProgram/r000mi  
cd ..  
i nspxe-cl -R=observations r000mi
```



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

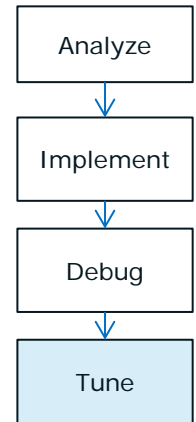
Step 4: Tune the parallel application.

Key Questions -Tune

Do my tasks do equal amounts of work?

Is my application scalable?

Is the threading running efficiently?





Where is my application...

Spending Time?

Function	CPU Time
- Call Stack	
algorithm_2	3.560s
do_xform	3.560s
algorithm_1	1.412s
BaseThreadInitTh	0.000s

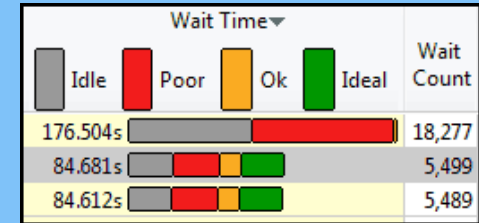
- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

Line		MEM_LOAD... LLC_MISS
475	float rx, ry, rz =	
476	float param1 = (AP	30,000
477	float param2 = (AP	
478	bool neg = (rz < C	

- 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

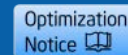
Advanced Profiling For Scalable Multicore Performance



Software & Services Group, Developer Products Division

Copyright © 2013, Intel Corporation. All rights reserved.

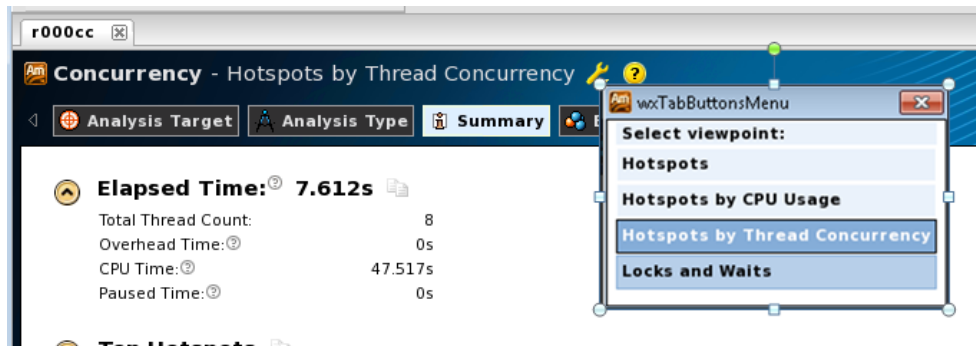
*Other brands and names are the property of their respective owners.



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



Profile the System

- [-] Algorithm Analysis
 - Advanced Hotspots

Profile Applications

- [-] Algorithm Analysis
 - Basic Hotspots
 - Concurrency
 - Locks and Waits

VTune Amplifier XE

User Mode

- Works on
- Intel
 - non-Intel
 - More overhead than lightweight hotspots

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

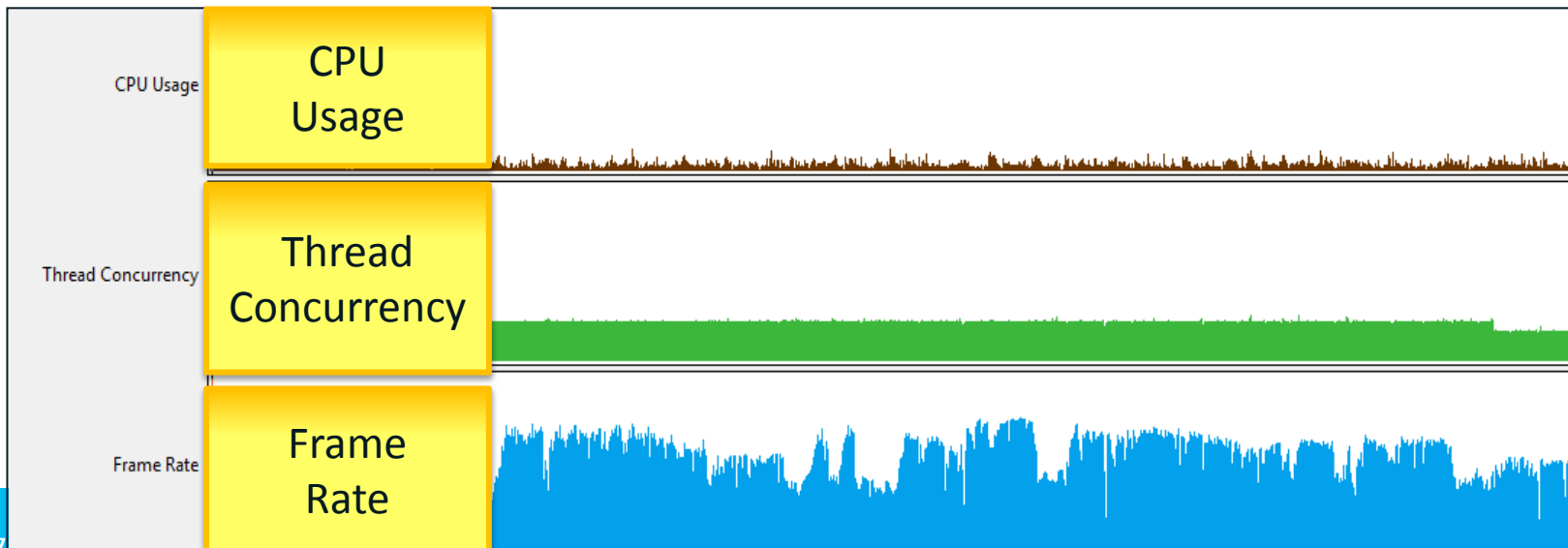
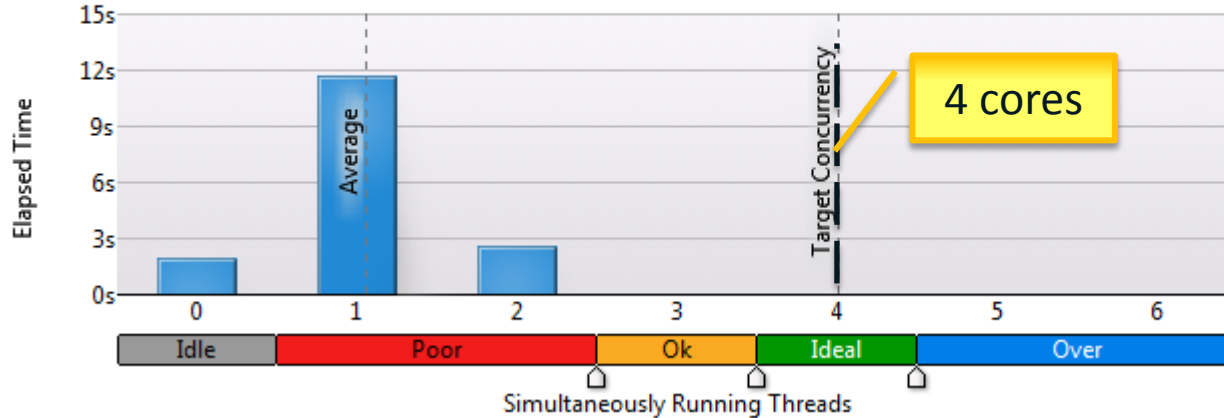
- Works only on
- Intel

Intel® VTune™ Amplifier XE

Get a quick snapshot

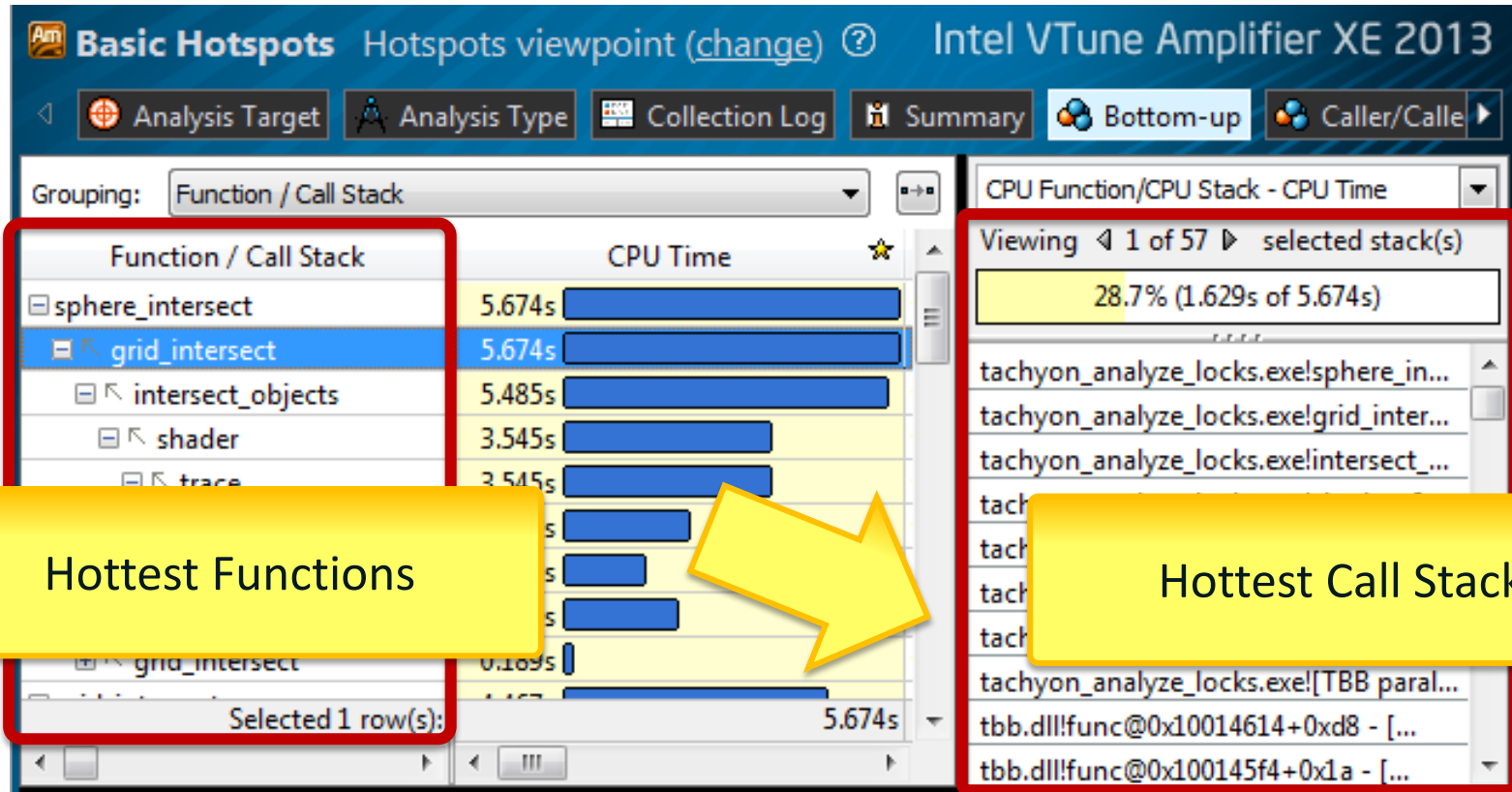
Thread Concurrency Histogram

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 Concurrency that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.



Intel® VTune™ Amplifier XE

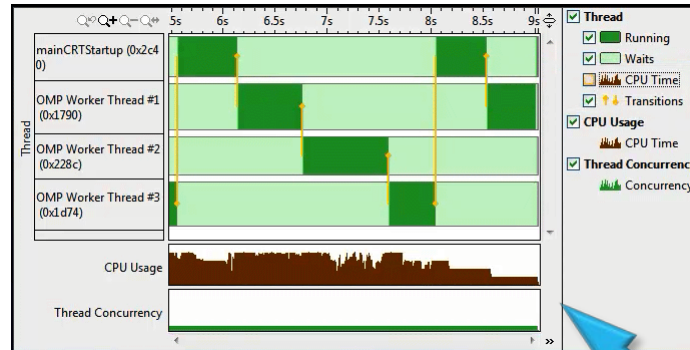
Identify hotspots



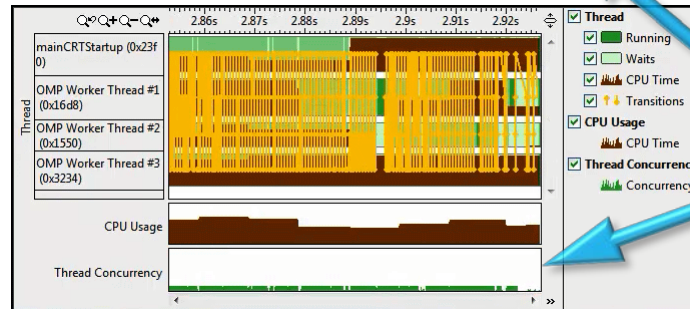
Intel® VTune™ Amplifier XE

Identify threading inefficiency

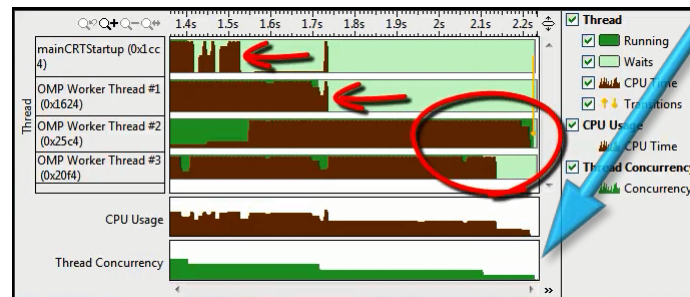
Coarse Grain
Locks



High Lock
Contention



Load
Imbalance



Low
Concurrency

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)

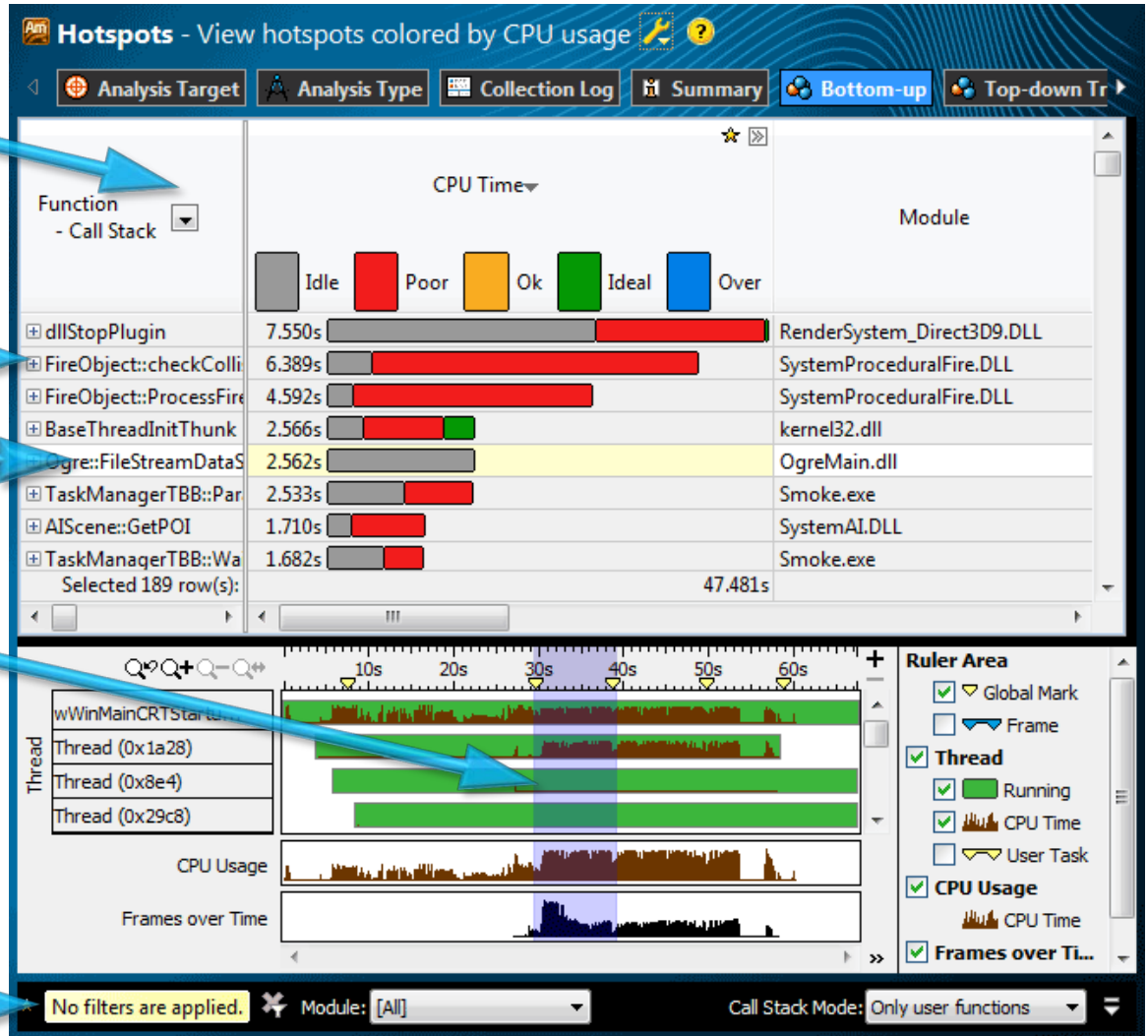
Click [+] for Call Stack

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

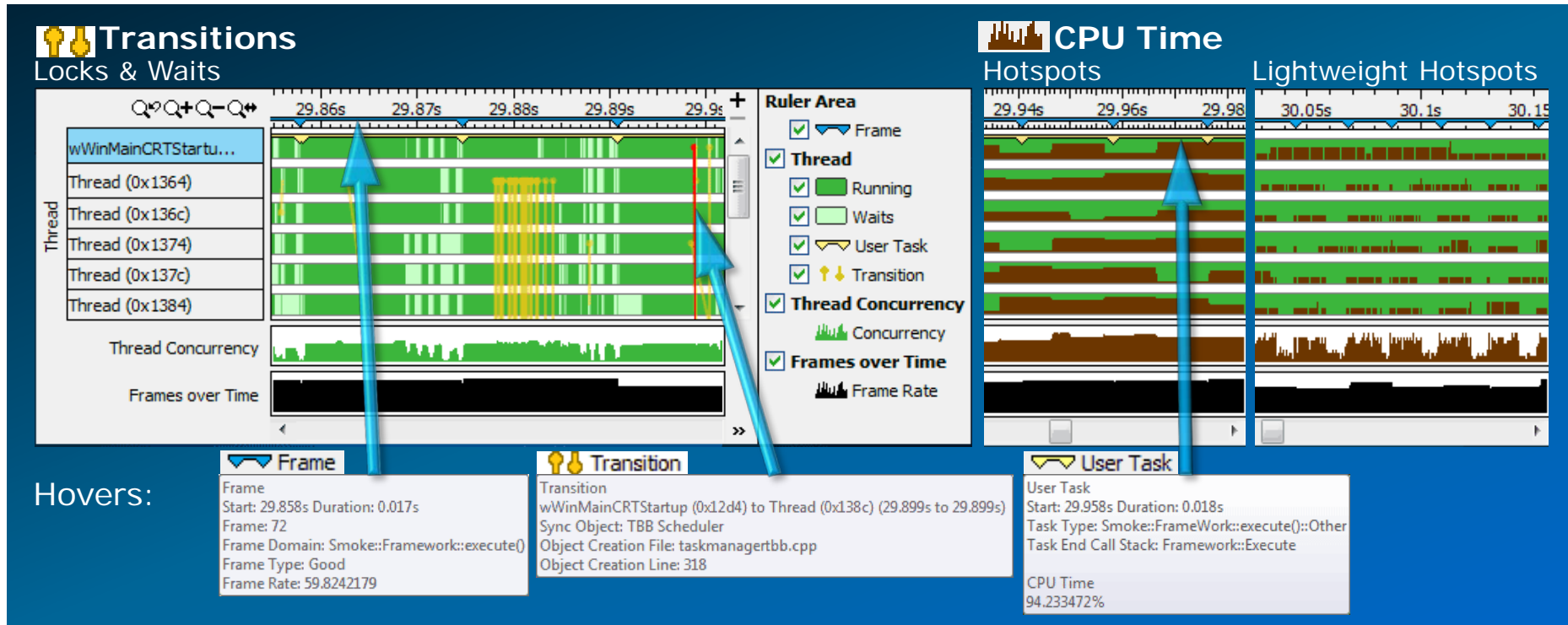
Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters

Filter by Module & Other Controls



Intel® VTune™ Amplifier XE

Timeline Visualizes Thread Behavior



Hovers:

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

Hotspots - View CPU time hotspots and stacks Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | FireObj...

Line	Source	CPU Time	Address	Assembly	CPU Time
469	FireObject::checkCollision(V3 pos,V3 pre	0.476s	0x388c	fld st0, dword ptr [esp+0xc]	0.004s
472	#define FMax std::max<float>	0.561s	0x3890	fld st0, st0	0.993s
	prev	6.846s	0x3892	fmulp st2, st0	0.787s
		3.593s	0x3894	fxch st0, st1	1.465s
477	float param2 = (AABB.zMax - prevPos.	0.830s	0x3896	fstp dword ptr [esp+0x8], st0	0.325s
478	bool neg = (rz < 0.f);	0.615s	0x389a	fld st0, dword ptr [esp+0x40]	0.014s
479	minP = FMax(neg? param2 : param1, mi	3.008s	0x389e	fsubrp st2, st0	0.010s
480	maxP = FMin(neg? param1 : param2, ma	1.875s	0x38a0	fld st0, st0	0.010s
481	if(maxP > minP) {	0.972s	0x38a2	fmulp st2, st0	0.233s
482	rx = 1.f/(pos.x - prevPos.x);	0.252s	0x38a4	fxch st0, st1	0.247s
483	param1 = (AABB.xMin - prevPos.x)	0.264s	0x38a6	fstp dword ptr [esp+0xc], st0	0.326s
484	param2 = (AABB.xMax - prevPos.x)	0.040s	0x38aa	fcomp st0, st2	0.032s
485	neg = (rx < 0.f);	0.047s	0x38ac	fnstsw stx	0.000s
	param1	0.274s	0x38ae	test ah, 0x5	0.000s
	param2	0.164s	0x38b1	jp 0x100038b1	0.000s
488	}	0.612s	0x38b3	Block 2:	0.000s
489	if(maxP > minP) {	0.612s	0x38b3	mov dl, 0x1	0.000s
		0.830s	0x38b5	lea ecx, ptr [esp+0xc]	0.024s
		0.830s	0x38b9	jmp 0x100038c1 <Block 4>	0.000s
		0.830s	0x38bb	Block 3:	0.000s
		0.830s	0x38bb	xor edi, dl	0.159s

Time on Source / Asm →

Quick Asm navigation: Select source to highlight Asm →

Quickly scroll to hot spots. →

Right click for instruction reference manual →

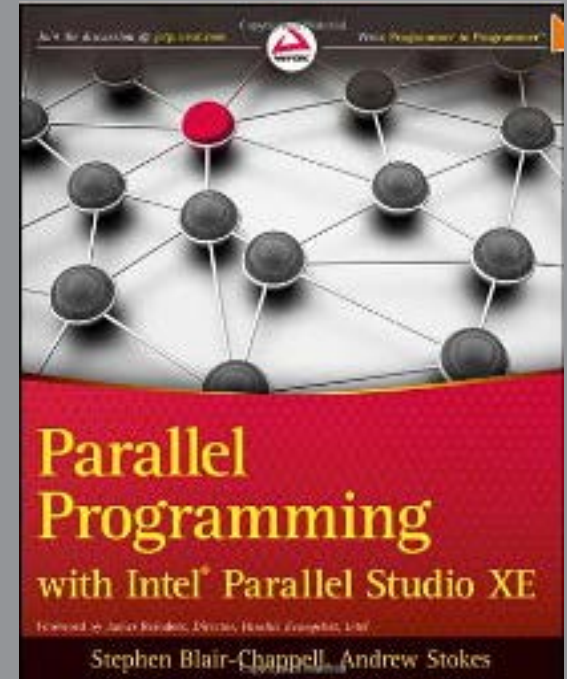
Click jump to scroll Asm →

Selected 1 row(s): 0.830s | Highlighted 6 row(s): 0.830s

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

10.254.198.28 - Remote Desktop Connection

File View Help

Solution Explorer

Solution 'Solution1' (0 p...

Concurrency - Hotspots by Thread Concurrency

Analysis Target Analysis Type Summary Bottom-up Top-down Tree

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization					Ove... Time	Wait Time by Utilization					Module
	Idle	Poor	Ok	Ideal	Over		Idle	Poor	Ok	Ideal	Over	
PrintProgress	48.751s					0.879s	44.569s					9-1.exe
GetPrimes\$omp\$parallel_for@51	3.727s					0.122s	4.117s					9-1.exe
write_nolock	1.767s					0s	0.769s					9-1.exe
SleepEx	0.271s					0s	0s					KERNELBASE.dll
lock	0.150s					0.013s	35.242s					9-1.exe
RtlIntegerToUnicodeString	0.107s					0s	0s					ntdll.dll

Selected 0 row(s):

Thread create stack

2 stack(s) selected. Viewing 1 of 2

Current stack is 91.6% of selection

- 91.6% (50.453s of 55.051s)
- 9-1.exe!GetPrimes(int,int) - ParallelPrime.cpp:51
- 9-1.exe!main - ParallelPrime.cpp:66
- 9-1.exe!_tmainCRTStartup - crt0.c:266
- 9-1.exe!_tmainCRTStartup - crt0.c:291

0.5s 1s 1.5s 2s 2.5s 3s 3.5s 4s 4.5s 5s 5.5s 6s 6.5s 7s 7.5s 8s

mainCRTStartup (0x16b)

OMP Worker Thread #1

OMP Worker Thread #2

OMP Worker Thread #3

OMP Worker Thread #4

OMP Worker Thread #5

OMP Worker Thread #6

OMP Worker Thread #7

OMP Worker Thread #8

OMP Worker Thread #9

OMP Worker Thread #1

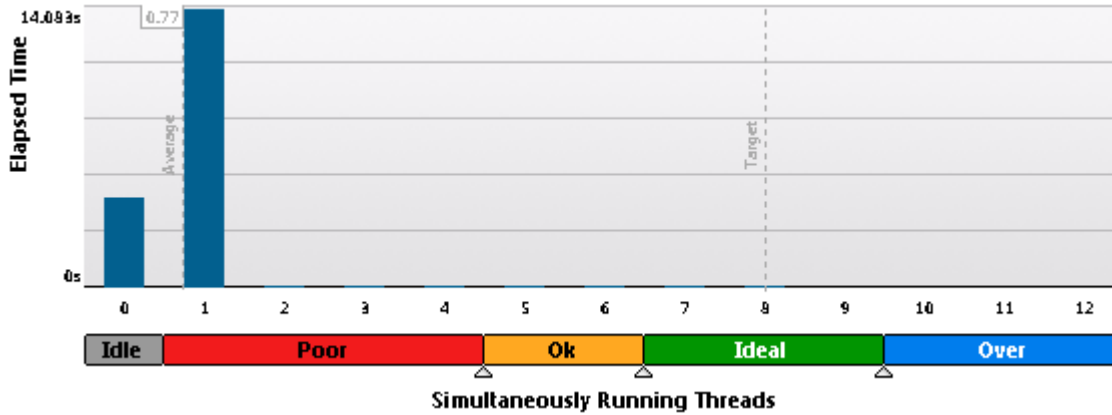
OMP Worker Thread #1

CPU Usage

Thread Concurrency

No filters are applied. Module: [All] Thread: [All] Process: [All] Utilization: [All] Inline Mode: on Call Stack Mode

Concurrency and Performance of Attempt 1



Intel VTune Amplifier XE 20

Concurrency - Locks and Waits

Analysis Target: r003cc

/Sync Object /Function /Call Stack	Wait Time	Wait Count	Spin Time	Module	Object Type	Object Creation Function	Object Creation Module
[Idle]							
OMP Critical_Proc_5:386 0xb65d92	133.814s	599,977	120.451s		OMP Critical	Proc_5	dhystone.exe
OMP Join Barrier_main:147 0xd24	0.247s	8	0.236s		OMP Join Barrier	main	dhystone.exe
Stream 0x7c2fc31d	0.007s	58	0s		Stream	main	dhystone.exe
[Unknown]	0s	0	0s		[Unknown]	[Unknown]	[Unknown]

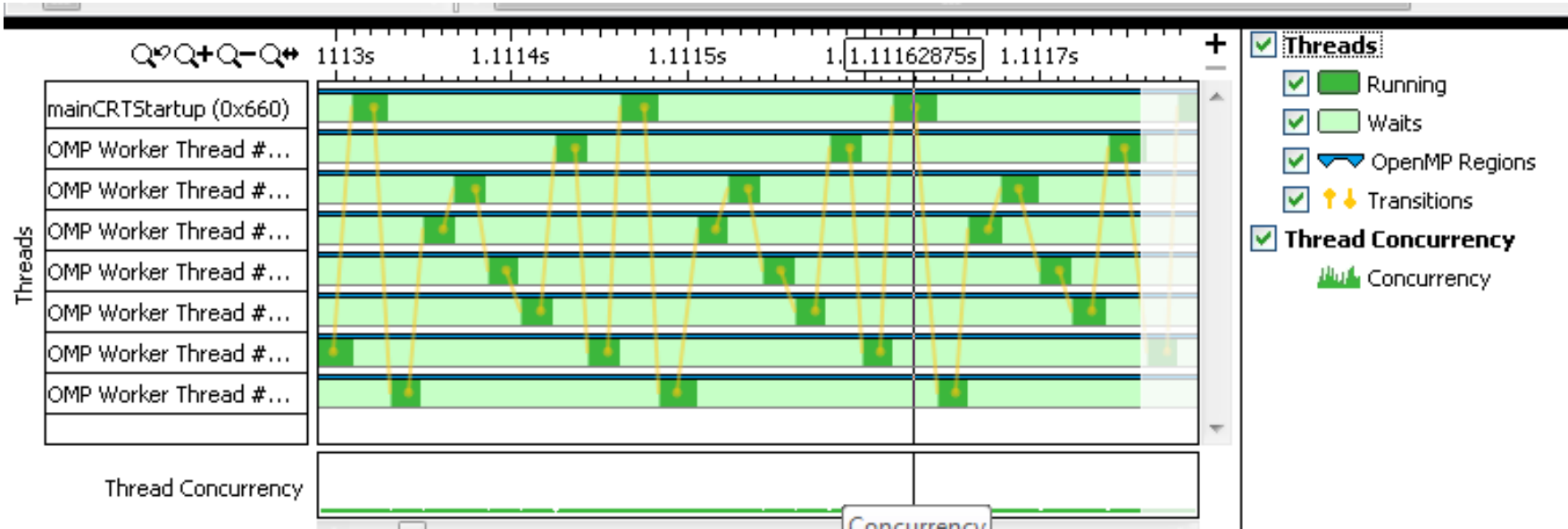
Thread create stack: 1 stack(s) selected. Viewing 1 of 1. Current stack is 0.0% of selection. 0.0% (0s of 0s). [Unknown]

Threads: mainCRTStartup (0x660), OMP Worker Thread #..., OMP Worker Thread #...

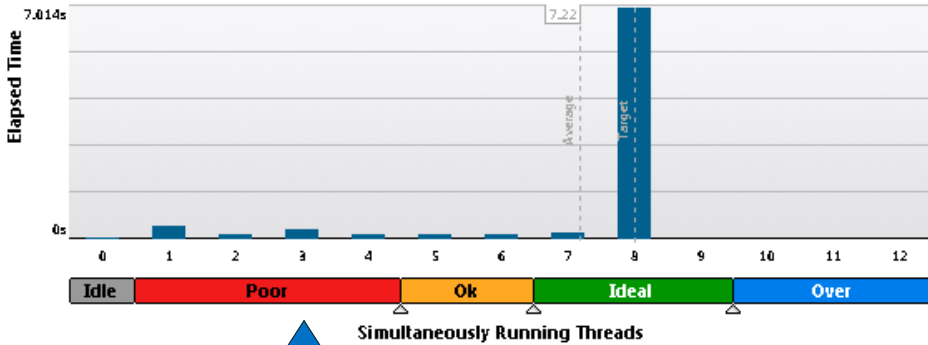
Thread Concurrency: [Graph showing thread activity over time]

Legend: Running (green), Waits (light green), OpenMP R (blue), Transitions (yellow), Thread Concurr (green), Concurrent (light green)

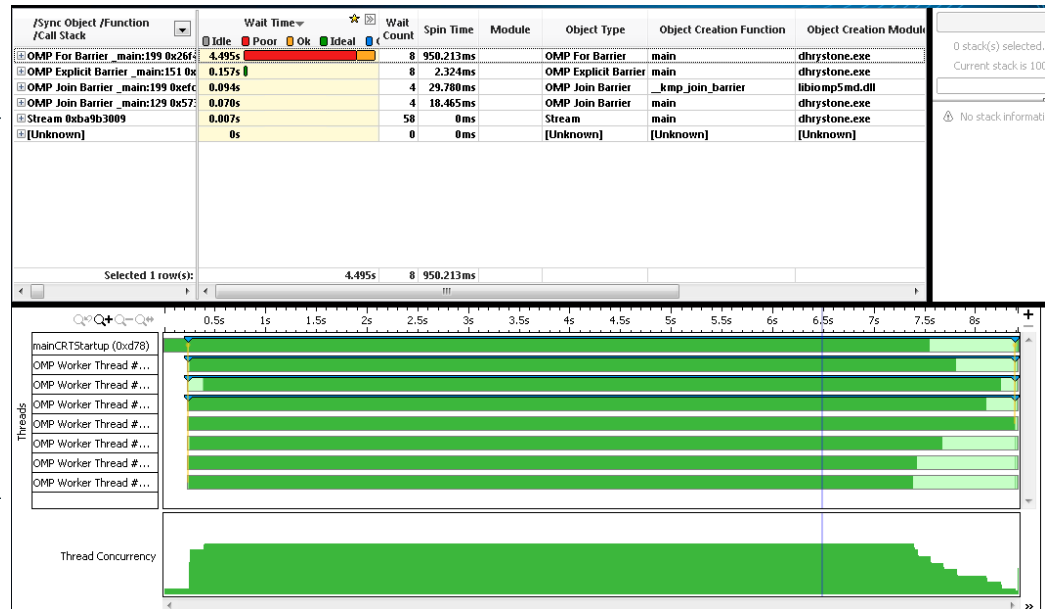
Zoom-in on time line shows reason for poor concurrency



Concurrency and Performance of Attempt 2



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



Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

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

