Agenda

• Why Parallel?

• Optimising Applications

• Steps to move from Serial to Parallel
A Message From Steve Lionel
Intel Compiler Labs

BCS Fortran Interest Group 40th Anniversary

The Chartered Institute for IT
Fortran Specialist Group
Moving to Parallel - a view from some developers

Top 5 challenges
- Legacy
- Education
- Tools
- Fear of many cores
- Maintainability
Why Parallel?

Section 1
Why is everyone going multi-core?

Power Density Race

- Sun’s Surface
- Rocket Nozzle
- Nuclear Reactor
- Hot Plate
- Pentium® processors

Power Density (W/cm²)

- '70: 4004
- '80: 8080, 8008
- '90: 286, 8085
- '00: 386, 486
- '10: 8080

Intel® Software Development Products Overview
Moore’s Law reinterpreted

From K. Olukotun, L. Hammond, H. Sutter, and B. Smith

- Speed no longer increasing
- Num transistors still growing
- Num Cores rather than clock speed is doubling every 18 months
Theoretical growth of cores

Growth of Multicore

- 2048 cores
- 512 cores
- 128 cores
- 32 cores
- 8 cores
- 2 cores


Num Cores: 10,000, 1,000, 100, 10

Copyright © 2007, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners
Future: Multicore and Manycore

Connections to memory bank(s), connections between processors, memory coherency models – all come into play. Diversity!

Note: the above pictures don’t necessarily represent any current or future Intel products.
Multi-core: beating the *power\performance* barrier

**Over-clocked (+20%)**

- Dual-Core: 1.73x
- Performance: 1.13x
- Power: 1.00x

**Design Frequency**

- Dual-Core: 1.73x
- Performance: 1.00x
- Power: 0.87x

**Under-clocked (-20%)**

- Dual-Core: 1.73x
- Performance: 1.02x
- Power: 0.51x

*Relative single-core frequency and Vcc*

Copyright © 2007, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
Industry’s First 45 nm High-K + Metal Gate Transistor Technology

- Improved Transistor Density ~2x
- Improved Transistor Switching Speed >20%
- Reduced Transistor Switching Power ~30%
- Reduction in gate oxide leakage power >10x

Enables New Features, Higher Performance, Greater Energy Efficiency
Intel’s Teraflops Research Chip

Podtech Intel Research Day Terascale.flv
# Intel’s Teraflops Research Chip

<table>
<thead>
<tr>
<th>Speed (GHz)</th>
<th>Power (Watts)</th>
<th>Perf. Teraflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>62</td>
<td>1.01</td>
</tr>
<tr>
<td>5.1</td>
<td>175</td>
<td>1.63</td>
</tr>
<tr>
<td>5.7</td>
<td>265</td>
<td>1.81</td>
</tr>
</tbody>
</table>
Larrabee

**CPU**
- Evolving toward throughput computing
- Motivated by energy-efficient performance

**GPU**
- Evolving toward general-purpose computing
- Motivated by higher quality graphics and data-parallel programming

Throughput Performance

Intel® Software Development Products Overview

Copyright © 2007, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
What can we do with faster Computers?

- Solve problems faster
  - Reduce turn-around time of big jobs
  - Increase responsiveness of interactive apps

- Get better solutions in the same amount of time
  - Increase resolution of models
  - Make model more sophisticated
Optimising Code

Section 2
Two points to address before you start parallelising

• Will buying a faster computer solve your problem?

  Dr Yann Golanski, York

  **Just buy a faster machine!**

  *First look at how much it will cost you to make your program parallel. If it will take say 2 months of coding, can you just buy a faster machine that will give you the speedup you want? Of course once you reach the limits of a machine’s speed, you are going to have to then do some parallelisation.*

• Maybe Serial Optimisation will be sufficient.
## A Three-Tiered Tuning Model

<table>
<thead>
<tr>
<th>Tuning Level</th>
<th>Question being asked</th>
<th>Examples of issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>System wide</td>
<td>Can my system be ‘tuned’ to improve the performance of my application</td>
<td>Network, disk and memory performance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intrusion by 3rd party programs such as virus scanners.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bad \ missing threading implementation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Failing to use latest generation optimised instructions.</td>
</tr>
</tbody>
</table>
Compiler generated Optimisations

Global Compiler Options
Inter-procedural Optimisations
Profile Guided Optimisations
Vectorisation
Parallelisation
Example of hand-crafted SSE instructions

```c
1: bool SSEHasNumber(SUDOKU *pPuzzle, __m128i BinArray[], int i, int j)
2: {
3:     __m128i Tmp1 = ( _mm_and_si128(pPuzzle->BinNum[j-1], BinArray[i]));
4:     __m128i Tmp2 = _mm_setzero_si128();
5:     Tmp2 = _mm_cmpeq_epi32(Tmp2, Tmp1);
6:     Tmp2 = _mm_cmpeq_epi32(Tmp2, Tmp1);
7:     unsigned int p[4]; _mm_storeu_si128((__m128i*)p, Tmp2);
8:     if(p[0] == 0 || p[1] == 0 || p[2] == 0) return true;
9:     return false;
10: }
```

<table>
<thead>
<tr>
<th>Time Taken</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>No SSE</td>
<td>4.55 sec</td>
</tr>
<tr>
<td>With SSE</td>
<td>0.19 sec</td>
</tr>
</tbody>
</table>

Copyright © 2007, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
Modern Architectures have lots of features to help speed up code.
Intel® VTune™ Performance Analyzer

Graphical tool
Helps characterise runtime performance
System-wide View of application environment
Use to tune serial and parallel code
Use to identify Hot Spots in Code
Use to generate a call graph
Call graph: Application workflow

The red lines show the critical path. The critical path is the most time-consuming call path. It is based on self time.

Bright orange nodes indicate functions with the highest self time.

Filter view by self time
The life of a program instruction

1. Instruction read from memory
2. Instruction fed to Decoder
3. Micro-ops (uops) generated
4. Uops queued in RS
5. Uops dispatched
6. Results sent to ROB
7. Instruction marked – all uops executed
8. Instruction sent for retirement

Memory Sub-system
Inst. Fetch Branch Pred
Decoder
Reservation Station
Execution Units
Reorder Buffer
Retirement
Hardware Performance Events

- BUS_TRANS_ANY.ALL_AGENTS
- RS_UOPS_DISPATCHED.CYCLES_NONE
- MEM_LOAD RETIRED.L2_MISS
- CPU_CLK_UNHALTED.CORE
- INST RETIRED.ANY
Demo 0 - Using Intel® VTune™ Performance Analyzer

From 1 to 1,000,000
Notice the System Wide View – Can you see any problems?

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcshield.exe</td>
<td>1,146</td>
</tr>
<tr>
<td>cidaemon.exe</td>
<td>1,067</td>
</tr>
<tr>
<td>02_Hotspot Analysis.exe</td>
<td>838</td>
</tr>
<tr>
<td>csrss.exe</td>
<td>807</td>
</tr>
<tr>
<td>pid_0x0</td>
<td></td>
</tr>
<tr>
<td>Process=02_Hotspot Ana</td>
<td></td>
</tr>
<tr>
<td>cisvc.exe</td>
<td>178</td>
</tr>
<tr>
<td>rundll32.exe</td>
<td>155</td>
</tr>
<tr>
<td>pid_0x4</td>
<td>111</td>
</tr>
<tr>
<td>services.exe</td>
<td>97</td>
</tr>
</tbody>
</table>
What is the problem here?

- VTune Sample-over-time view
The Hotspot

VTune Performance Environment - Sampling Hotspots - [Sun Jun 06 23:12:43 2010 - Sampling Results [ECWMOSBLAIRCH]]

From Serial to Parallel

Copyright © 2008, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners
Four Steps in Moving to Parallel

Section 2
Steps in moving from Serial to Parallel

1. Architectural Analysis
2. Introducing Parallelism
3. Validating Correctness
4. Performance Tuning
5. Parallel
Key Questions

Design
- 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?

Code & Debug
- How?
- How difficult?
- Is my code still working?

Verify
- Is the parallelism correct?
- Do I have deadlocks or data races?
- Do I have memory errors?
- Does my program still work as intended?

Tune
- Do my tasks do equal amounts of work?
- Is my application scalable?
- Is the threading running efficiently?
Tools

Serial
- Architectural Analysis
- Introducing Parallelism
- Validating Correctness
- Performance Tuning
- Parallel

Existing Intel Software

Intel® VTune™ Performance Analyzer

Intel Parallel Studio

Advisor/Amplifier

Intel Compilers
Parallel Libraries

Composer

Intel® Thread Checker

Inspector

Intel® Thread Profiler

Amplifier
For Microsoft Visual Studio* C++ architects, developers, and software innovators creating parallel Windows* applications.

Intel® Parallel Studio includes:
- Intel® Parallel Advisor Lite **
- Intel® Parallel Composer
- Intel® Parallel Inspector
- Intel® Parallel Amplifier

** Beta – from whatif.intel.com
Step 1

Which part of my code should I make Parallel?
Key Questions - Design

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?
Identifying best parts to Parallelize

We need to Identify Hotspots
Why use parallelism?
Amdahl’s Law

Describes the upper bound of parallel execution speedup

Serial code limits speedup

\[ T_{\text{parallel}} = \left\{ (1-P) + \frac{P}{n} \right\} T_{\text{serial}} \]

\[ \text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \]

\[ \frac{1.0}{0.35} = 2.857 \]

Copyright © 2008, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
Some code is not worth making parallel...

Don’t parallelise code
  – just because it’s clever
  – With low CPU utilisation
  – I/O bound

• Do parallelise code that
  – Eats significant CPU cycles

• You need to get visibility of the runtime behaviour
Architectural Extensions can speed up your code

- Always optimise your code

- Even if you don’t go parallel, some architectural features can still give significant speed-up

- Example, SSE extensions
bool TestForPrime(int val) {
    // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
        factor ++;
    return (factor > limit);
}

void FindPrimes(int start, int end) {
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
Demo 1 – Getting the Benchmark

From 1 to 1,000,000
Our Application - Prime Number Generator

```c
bool TestForPrime(int val) {
    // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
        factor ++;
    return (factor > limit);
}
```

```c
void FindPrimes(int start, int end)
{
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
```

Running 00_Benchmark Serial.exe
100%

78498 primes found between 0 and 1000000 in 2.48 secs

```c
int range = end - start + 1;
for( int i = start; i <= end; i += 2 )
{
    if( TestForPrime(i) )
        globalPrimes[gPrimesFound++] = i;
    ShowProgress(i, range);
}
```
Optimise the serial code first

- Using Intel Compiler to automatically generate SSE instructions.
  - Code ran twice as fast
  - No change made to original code

<table>
<thead>
<tr>
<th>Calculating Pi</th>
<th>Time Secs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>No SSE</td>
<td>1.29</td>
<td>1.00</td>
</tr>
<tr>
<td>With SSE</td>
<td>0.66</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Example of SSE compiler-generated instructions
Demo 2 - Using the Intel Compiler

From 1 to 1,000,000
Swapping compilers.

- From solution drop-down menu

- Action is reversible
Program built and run with Intel compiler

Running 00_Benchmark Serial.exe
100%
78498 primes found between 0 and 1000000 in 2.48 secs

Running 01_Converted To Intel.exe
100%
78498 primes found between 0 and 1000000 in 2.28 secs

Speedup 1.09
Identifying Hotspots

Pinpointing places where an application could be parallelised
The Big Question

“How can I make my code run faster?”
Today’s Question

“Where do I split up my code to take advantage of multiple CPU cores?”
The task, Identifying the Hot Spot...
... and Splitting up the Work.
Demo 2 - Finding the Hotspots
Finding a Hot Spot
## Where to Parallelise - Amplifier

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>CPU Time:Total</th>
<th>CPU Time:Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>BaseProcessStart</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>_mainCRTStartup</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>main</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>FindPrimes</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>TestForPrime</td>
<td>24.4%</td>
<td>0.328s</td>
</tr>
<tr>
<td>ShowProgress</td>
<td>75.6%</td>
<td>1.016s</td>
</tr>
</tbody>
</table>

**Call Stack**

**Hotspot**
Design: What’s the expected speedup?

- **Use Amdhals Law**

  \[
  \text{Speedup} = \frac{1}{[s + (1-s)/n + H(n)]}
  \]

  - \(s\) is serial part (fraction of 1)
  - \(H\) is parallel overhead (ignore)
  - \(n\) is number of cores

  \[
  S = 0 \\
  \text{Speedup} = \frac{1}{[0 + (1-0)/2]} \\
  = \frac{1}{[0 + 0.5]}
  \]

  **Speedup = 2** (i.e. new speed ~ 0.672 seconds)
Alternate Calculation

\[
\text{Speedup} = \frac{1}{s + \frac{(1-s)}{n} + H(n)}
\]

- \(s\) is serial part (fraction of 1)
- \(H\) is parallel overhead (ignore)
- \(n\) is number of cores

\[
S = 1 - \frac{(1.688 - 0.012)}{1.688} = 0.007
\]

\[
\text{Speedup} = \frac{1}{0.007 + (1 - 0.007)/2} = \frac{1}{0.007 + 0.4965}
\]

\[
\text{Speedup} = 1.986 \quad \text{(i.e. CPU Time \sim 0.850 seconds)}
\]
Step 2
Implement Parallelism in code
Key Questions – Code & Debug

How?
How difficult?
Is my code still working?
Common types of parallelism

• Functional or Task Parallelism
• Data Parallelism
• Software Pipelining
Task and Data Parallelism

- Different job for each thread
  - e.g. one thread prints, another reads keyboard

- Splitting workload between multiple identical threads
  - e.g. three identical threads perform calculations on data array

Task parallelism

Data parallelism
Software Pipeline

Core 1: Collect A, Collect B, Collect C, Collect D, ...
Core 2: Transfer A, Transfer B, Transfer C, Transfer D, ...
Core 3: Polish A, Polish B, Polish C, ...
Core 4: Produce A, Produce B, ...

Time
Question

• How many different ways can you think of to implement parallelism?

– *E.g* OpenMP, ..., ...
Intel’s Family of Parallel Models

**Most Composable Parallel Models**

- Intel® Threading Building Blocks (TBB)
- Intel® Cilk
- Intel® Ct Technology
- Array Notations and SIMD pragmas

**Other Supported Standards**

- Fixed Function Libraries
- Intel® Math Kernel Library (MKL)
- Intel® Integrated Performance Primitives (IPP)
- Intel® OpenCL SDK

**Research Initiatives**

- MPI
- OpenMP*
- OpenMP
- Intel® Cluster OpenMP

**Intel® Concurrency Collections**

- Software Transactional Memory

---

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

Loop-level parallelism automatically supplied by the compiler
Auto-parallelization

- Auto-parallelization: Automatic threading of loops without having to manually insert OpenMP* directives.

<table>
<thead>
<tr>
<th>Windows*</th>
<th>Linux*</th>
<th>Mac*</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Qparallel</td>
<td>-parallel</td>
<td>-parallel</td>
</tr>
</tbody>
</table>

- Compiler can identify “easy” candidates for parallelization, but large applications are difficult to analyze.
<table>
<thead>
<tr>
<th>Optimisation</th>
<th>Time Taken (secs)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>0.938</td>
<td>1</td>
</tr>
<tr>
<td>auto-vectorisation</td>
<td>0.375</td>
<td>2.5</td>
</tr>
<tr>
<td>auto-parallelism</td>
<td>0.516</td>
<td>1.8</td>
</tr>
<tr>
<td>auto-vec. &amp; auto-par.</td>
<td>0.203</td>
<td>4.6</td>
</tr>
</tbody>
</table>
OpenMP Architecture

- Fork-Join Model
- Worksharing constructs
- Synchronization constructs
- Directive/pragma-based parallelism
- Extensive API for finer control
OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance are met: i.e. the sequential program evolves into a parallel program.

Parallel Regions

Sequential Parts

A Nested Parallel region

Master Thread in red

Parallel Regions

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

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

OpenMP

Divide iterations of the for loop

Create threads here for this parallel region
Demo 3 : Adding parallelism using #pragma omp for
Results – Open MP

Amazing! We Have a speed up!
Code: Is my code still working?

Running 01_Converted To Intel.exe
100%
78498 primes found between 0 and 1000000 in 2.28 secs

Running 03_Naive OpenMP.exe
50%
36964 primes found between 0 and 1000000 in 1.45 secs

Bother !!!!!!!!!!!!!!
Number of primes is wrong
Questions

Are the results right?

Was the run quicker?
Architectural Analysis
Introducing Parallelism
Validating Correctness
Performance Tuning
Parallel

Step 3
Check for any problems
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?
New paradigm requires new tools

• Using traditional debugging tools is difficult/impossible
  - Printf – not re-entrant
  - Debugging several threads is notoriously hard
  - Many debuggers/profilers are not multi-core enabled

• Multi-core tools are available
Non deterministic Error Sources in parallel Applications

- Shared Resources require locks

- Locks can
  - ‘serialize’ a program
  - lead to Deadlocks

Wrong Result
(X should be 2)
Demo 4 – Checking for threading errors
Checking for Errors with Parallel Inspector

### Problem Sets

<table>
<thead>
<tr>
<th>ID</th>
<th>Problem</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Data race</td>
<td>PrimeOpenMP.cpp</td>
</tr>
<tr>
<td>P2</td>
<td>Data race</td>
<td>PrimeOpenMP.cpp</td>
</tr>
<tr>
<td>P3</td>
<td>Potential privacy infringement</td>
<td>PrimeOpenMP.cpp</td>
</tr>
<tr>
<td>P4</td>
<td>Potential privacy infringement</td>
<td>PrimeOpenMP.cpp</td>
</tr>
<tr>
<td>P5</td>
<td>Potential privacy infringement</td>
<td>PrimeOpenMP.cpp</td>
</tr>
</tbody>
</table>
The Offending Sources

Focus Observation: PrimeOpenMP.cpp:116 - Read

111
112 #pragma omp parallel for
113 for( int i = start; i <= end; i += 2 )
114 {
115   if( TestForPrime(i) )
116     globalPrimes[gPrimesFound++] = i;
117     ShowProgress(i, range);
118 }
119 }
120 }
121
122 int main(int argc, char **argv)

Related Observation: PrimeOpenMP.cpp:116 - Write

111
112 #pragma omp parallel for
113 for( int i = start; i <= end; i += 2 )
114 {
115   if( TestForPrime(i) )
116     globalPrimes[gPrimesFound++] = i;
117     ShowProgress(i, range);
118 }
119 }
120 }
121
122 int main(int argc, char **argv)

From Serial to Parallel

Copyright © 2008, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners
Protecting shared variables

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

Will create a critical section for this reference

```c
#pragma omp critical
{
gProgress++;  // Will create a critical section for both these references
percentDone = (int)(gProgress/range *200.0f+0.5f)
}
```

Copyright © 2008, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners*
Demo 5 - Fixing the threading errors
Data Races Have Disappeared!

<table>
<thead>
<tr>
<th>Problem Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
</tbody>
</table>
Number of Primes is Correct

78498 primes found between 1 and 1000000 in 3.53 secs

Number of primes is correct
Step 4
Tune for best performance
Key Questions - Tune

Is the threading running efficiently?

Do my tasks do equal amounts of work?

Is my application scalable?
Performance Issues

Load Balancing

Synchronisation Overhead

Scalability

Difficult to examine without the right tools
A Reminder – Where are we?

Number of primes is correct

Almost as slow as the serial version
Demo 6 – Find the Threading Performance Issues
## Hotspot Analysis

### Hotspots

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>CPU Time:Total</th>
<th>CPU Time:Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>100.0%</td>
<td>0s</td>
</tr>
<tr>
<td>BaseThreadStart</td>
<td>50.8%</td>
<td>0s</td>
</tr>
<tr>
<td>_kmp_launch_worker</td>
<td>50.8%</td>
<td>0s</td>
</tr>
<tr>
<td>_kmpc_invoke_task_func</td>
<td>50.8%</td>
<td>0s</td>
</tr>
<tr>
<td>_kmpc_invoke_task_func</td>
<td>50.8%</td>
<td>0s</td>
</tr>
<tr>
<td>_kmp_invoke_microtask</td>
<td>50.8%</td>
<td>0s</td>
</tr>
<tr>
<td><code>L_findPrimes@@YAXHH</code></td>
<td>50.8%</td>
<td>0.023s</td>
</tr>
<tr>
<td><code>ShowProgress</code></td>
<td>42.1%</td>
<td>1.231s</td>
</tr>
<tr>
<td>TestForPrime</td>
<td>7.9%</td>
<td>0.231s</td>
</tr>
<tr>
<td>BaseProcessStart</td>
<td>49.2%</td>
<td>0s</td>
</tr>
<tr>
<td>_mainCRTStartup</td>
<td>49.2%</td>
<td>0s</td>
</tr>
<tr>
<td><code>FindPrimes</code></td>
<td>49.2%</td>
<td>0.016s</td>
</tr>
<tr>
<td><code>L_findPrimes@@YAXHH</code></td>
<td>48.7%</td>
<td>0.012s</td>
</tr>
<tr>
<td><code>ShowProgress</code></td>
<td>43.9%</td>
<td>1.283s</td>
</tr>
<tr>
<td>TestForPrime</td>
<td>4.3%</td>
<td>0.127s</td>
</tr>
</tbody>
</table>
```
81 }  
82  
83 void ShowProgress( int val, int range ) 
84 {  
85       int percentDone = 0;  
86  
87 #pragma omp critical 
88 {  
89       gProgress++; 
90  
91       percentDone = (int)((float)gProgress/(float)range *200.0f +  
92 }  
93  
94  
95     if( percentDone % 10 == 0 )  
96         printf("\\b\\b\\b\\b\\b\%3d\n", percentDone);  
97  
98 }  
99 
```
void ShowProgress( int val, int range )
{
    int percentDone;
    static int lastPercentDone = 0;

    gProgress++;
    percentDone = (int)((float)gProgress/(float)range*200.0f+0.5f);
}

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

if( percentDone % 10 == 0 && lastPercentDone < percentDone / 10)
{
    printf("\b\b\b\b%3d\%", percentDone);
    lastPercentDone++;
}

The algorithm has many more updates than the 10 needed for showing progress

This change should fix the contention issue
Demo 7 – Fixing the Synchronisation issues
Superb Speedup ... ???

Running 01_Converted To Intel.exe
100%
78498 primes found between 0 and 1000000 in 2.28 secs

Running 07 Reducing Printfs.exe
100%
78498 primes found between 0 and 1000000 in 0.31 secs

Speedup 7.36
On a dual core?
Demo 8 – Getting New Serial Benchmark
That’s better (but disappointing)...

Running 08_New Serial Benchmark.exe
100%
78498 primes found between 0 and 1000000 in 0.48 secs

Running 07_Reducing Printfs.exe
100%
78498 primes found between 0 and 1000000 in 0.31 secs

Speedup 1.55
Demo 9 – Correcting the Synchronisation Issue
### Hotspots

#### Function - Caller Function Tree

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time: Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestForPrime</td>
<td>388.224ms</td>
</tr>
<tr>
<td>L:?FindPrimes@YAXHH@Z_119_par_loop0_2.61</td>
<td>388.224ms</td>
</tr>
<tr>
<td>_kmp_invoke_microtask</td>
<td>247.670ms</td>
</tr>
<tr>
<td>_kmp_invoke_task_func</td>
<td>140.555ms</td>
</tr>
<tr>
<td>FindPrimes ← main ← _tmainCRTStartup ← BaseProcessStart</td>
<td>52.530ms</td>
</tr>
<tr>
<td>FindPrimes ← main ← _tmainCRTStartup ← BaseProcessStart</td>
<td>47.070ms</td>
</tr>
<tr>
<td>_L:?FindPrimes@YAXHH@Z_119_par_loop0_2.61</td>
<td>23.560ms</td>
</tr>
<tr>
<td>_kmp_invoke_microtask</td>
<td>23.510ms</td>
</tr>
<tr>
<td>_kmp_invoke_task_func</td>
<td>23.488ms</td>
</tr>
<tr>
<td>ShowProgress</td>
<td>46.924ms</td>
</tr>
<tr>
<td>L:?FindPrimes@YAXHH@Z_119_par_loop0_2.61</td>
<td>46.924ms</td>
</tr>
<tr>
<td>_kmp_invoke_microtask</td>
<td>23.435ms</td>
</tr>
<tr>
<td>_kmp_invoke_task_func</td>
<td>9.970ms</td>
</tr>
<tr>
<td>main</td>
<td>9.970ms</td>
</tr>
<tr>
<td>_tmainCRTStartup ← BaseProcessStart</td>
<td>9.970ms</td>
</tr>
</tbody>
</table>
## Locks & Waits

![Image of Locks and Waits visualization]

### Sync Object Name
- Wait Function
- Wait Function Tree

<table>
<thead>
<tr>
<th>Sync Object Name</th>
<th>Wait Time by Utilization: Self</th>
<th>Wait Count: Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP Critical FindPrimes:3</td>
<td>25.070s</td>
<td>479605</td>
</tr>
<tr>
<td>ShowProgress</td>
<td>21.982s</td>
<td>414666</td>
</tr>
<tr>
<td>_L?FindPrimes@</td>
<td>21.982s</td>
<td>414666</td>
</tr>
<tr>
<td>_kmp_invoke</td>
<td>11.160s</td>
<td>208993</td>
</tr>
<tr>
<td>FindPrimes</td>
<td>10.822s</td>
<td>205673</td>
</tr>
<tr>
<td>L?FindPrimes@@YAX</td>
<td>3.088s</td>
<td>64939</td>
</tr>
</tbody>
</table>

From Serial to Parallel
```c
84  {
85      int percentDone = 0;
86      static int lastPercentDone = 0;
87
88      #pragma omp critical
89      {
90          gProgress++;
91
92          percentDone = (int)((float)gProgress);
93      }
94
95      if( percentDone % 10 == 0 & lastPercentDone != percentDone )
96          printf("%d\r", percentDone);
97      lastPercentDone++;
98  }
99
100 }
101 ```
Fixing the synchronisation issue -1

This fix removes the need for a critical section

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

    #pragma omp parallel for
    for( int i = start; i <= end; i += 2 ) {
        if( TestForPrime(i) ) {
            globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;
            ShowProgress(i, range);
        }
    }
}
```
Fixing the synchronisation issue - 2

This fix removes the need for a critical section

```c
void ShowProgress( int val, int range )
{
    long percentDone, localProgress;
    static int lastPercentDone = 0;

    localProgress = InterlockedIncrement(&gProgress);
    percentDone = (int)((float)localProgress/
        (float)range*200.0f+0.5f);

    if( percentDone % 10 == 0 && lastPercentDone < percentDone / 10)
        printf("\b\b\b\b\b%3d\%", percentDone);
    lastPercentDone++;
}
```
That’s better (but still disappointing)...

Running 08_New Serial Benchmark.exe
100%
78498 primes found between 0 and 1000000 in 0.48 secs

Running 10_Fixing Synchronisation Issue.exe
100%
78498 primes found between 0 and 1000000 in 0.30 secs

Speedup 1.6
Demo 9 – Improving the Load Balancing
Threads are not doing equal work
Fixing a Load Imbalance

Distribute the work more evenly

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

#pragma omp parallel for schedule(static, 8)
    for( int i = start; i <= end; i += 2 ) {
        if( TestForPrime(i) )
            globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;
        ShowProgress(i, range);
    }
}
```
That’s better

Running 08_New Serial Benchmark.exe
100%
78498 primes found between 0 and 1000000 in 0.48 secs

Running 11_Load Balancing.exe
100%
78498 primes found between 0 and 1000000 in 0.25 secs

Speedup 1.92
A Finely Balanced threaded Program!

Concurrency

Thread
- Function
- Caller Function Tree
OMP Worker Thread #1 (1)
mainCRTStartup (2)

CPU Time by Utilization: Self

Elapsed Time: 0.288s
CPU Time: 0.472s
Wait Time: 0.010s
Wait Count: 29
Unused CPU Time: 0.037s
Core Count: 2
Threads Created: 2

Simultaneous Running Threads

1.87
Key Questions - Tune

Is the threading running efficiently?

Do my tasks do equal amounts of work?

Is my application scalable?
Moving to Parallel – a view from some developers

Top 5 challenges

• Legacy
• Education
• Tools
• Fear of many cores
• Maintainability
### Step 3: Summary

<table>
<thead>
<tr>
<th>Session Name</th>
<th>prime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upload zip file</td>
<td>11_Load Balancing.zip</td>
</tr>
<tr>
<td>Command Line</td>
<td>11_Load Balancing.exe 1 1000000</td>
</tr>
</tbody>
</table>

Click on DONE to get the reports of the execution.
The Results

Your job has completed!

paralleluniverse@intel.com

Sent: Mon 07/06/2010 05:05
To: Blair-chappel, Stephan

We're sending you this email because your job "prime" has successfully completed.

Please click the link below to access the results page:

http://paralleluniverse.intel.com/Job/Summary/878388be-a821-46f9-a911-0592309db43b

Thanks for using Intel® Parallel Universe Portal!
Without the printfs

<table>
<thead>
<tr>
<th>Cores</th>
<th>Elapsed Time (secs)</th>
<th>Number of Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.17</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.17</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0.12</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0.10</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>0.11</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cores</th>
<th>Average Concurrency</th>
<th>Number of Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1.55</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2.89</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3.94</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3.31</td>
<td>3</td>
</tr>
</tbody>
</table>
A run of 10 million
## Tools

### Serial
- Architectural Analysis
- Introducing Parallelism
- Validating Correctness
- Performance Tuning
- Parallel

### Existing Intel Software
- Intel® VTune™ Performance Analyzer
- Intel Compilers
- Parallel Libraries
- Intel® Thread Checker
- Intel® Thread Profiler

### Intel Parallel Studio
- Advisor/Amplifier
- Composer
- Inspector
- Amplifier
thank you

intel.com / go / parallel
Q&A

thank you