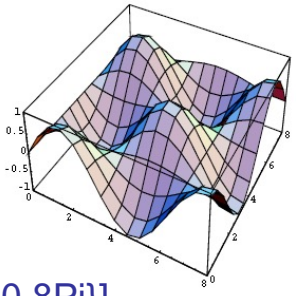
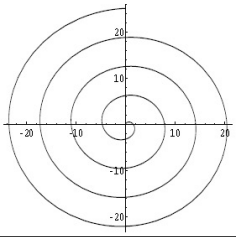
Table of Contents

Visualization – Mathematica Graphics

- Plot3D[Sin[x]*Cos[y], {x,0,8},{y,0,8}]

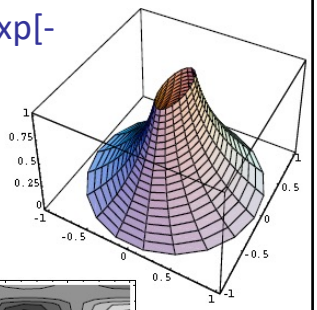


- ParametricPlot[{t*Sin[t],t*Cos[t]},{t,0,8Pi}]

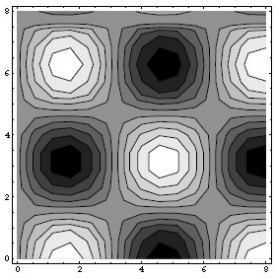


Visualization – Mathematica Graphics

- ParametricPlot3D[{Exp[-2*u]*Sin[t], Exp[-u]*Cos[t],u},{t,0,2Pi},{u,0,1}]



- ContourPlot[Sin[x] Cos[y], {x,0,8}, {y,0,8}]



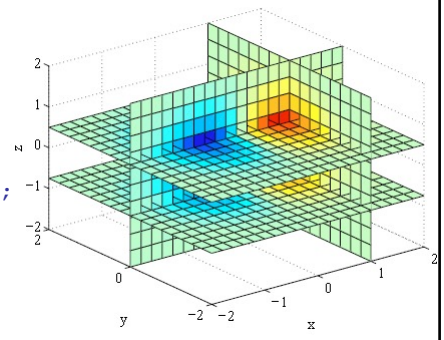
Visualization – Matlab Graphics

- 3D volume visualization of the function

$$f(x,y,z) = xe^{-x^2-y^2-z^2}, \text{where } x,y,z \in [-2,2]$$

- Matlab code:

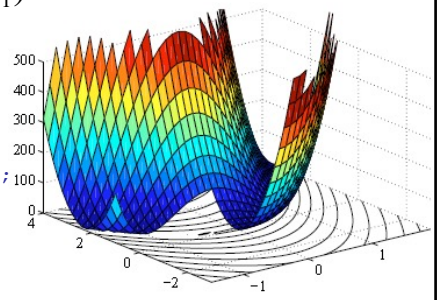
```
r1 = -2:.2:2;
r2 = -2:.25:2;
r3 = -2:.4:2;
[x,y,z] = meshgrid(r1, r2, r3);
v = x.*exp(-x.^2 - y.^2 - z.^2);
colormap(jet);
brighten(0.5);
slice(r1,r2,r3,v, ...
   [1.0],[0.0],[-0.75,0.5])
xlabel('x')
ylabel('y')
zlabel('z')
```



Visualization – Matlab Graphics

- Surface and contour plot of the Rosenbrock function $100(x_2 - x_1^2)^2 + (1 - x_1)^2$
- Matlab code:

```
x1 = -1.5:0.1:1.9;
x2 = -3:0.2:4;
[xx1 xx2] = meshgrid(x1, x2);
z = 100*(xx2-xx1.^2).^2 + (1-xx1).^2;
nan = NaN;
z0 = z;
% removal of points:
z0(z0 > 600) = nan*z0(z0 > 600);
z0(1:20,1:15) = nan(ones(20,15));
% surface plot:
surf(x1,x2,z0)
axis([-1.5 1.9 -3 4 0 500])
```




```
hold on
set(gcf, 'DefaultLineLineWidth', 2)
contour(x1,x2,z,(0:1.4:50).^3);
hold off
caxis([0,500])
```

Foster's Design Methodology

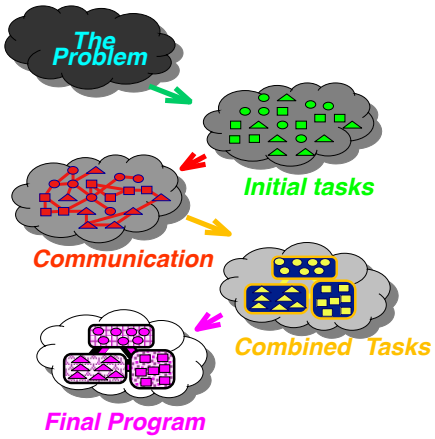
- From *Designing and Building Parallel Programs* by Ian Foster
- Four Steps:
  - **Partitioning**
    - Dividing computation and data
  - **Communication**
    - Sharing data between computations
  - **Agglomeration**
    - Grouping tasks to improve performance
  - **Mapping**
    - Assigning tasks to processors/threads

Parallel Algorithm Design: PCAM

- **Partition**: Decompose problem into fine-grained tasks to maximize potential parallelism
- **Communication**: Determine communication pattern among tasks
- **Agglomeration**: Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs
- **Mapping**: Assign tasks to processors, subject to tradeoff between communication cost and concurrency

Designing Threaded Programs

- **Partition**
  - Divide problem into tasks
- **Communicate**
  - Determine amount and pattern of communication
- **Agglomerate**
  - Combine tasks
- **Map**
  - Assign agglomerated tasks to created threads

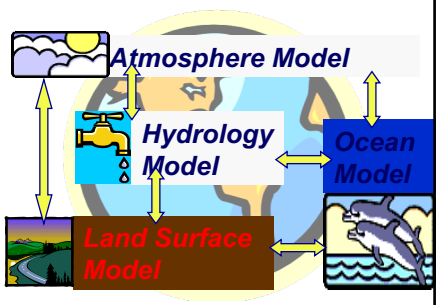


Parallel Programming Models

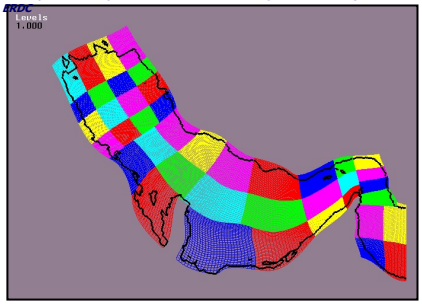
- Functional Decomposition
  - Task parallelism
  - Divide the computation, then associate the data
  - Independent tasks of the same problem
- Data Decomposition
  - Same operation performed on different data
  - Divide data into pieces, then associate computation

Decomposition Methods

- Functional Decomposition
  - Focusing on computations can reveal structure in a problem



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



Domain Decomposition

- *Focus on largest or most frequently accessed data structure*
- *Data Parallelism*
  - *Same operation applied to all data*

Example: Computing Pi

- We want to compute π
- One method: method of darts*
- Ratio of area of square to area of inscribed circle proportional to π

Disclaimer: this is a TERRIBLE way to compute π. Don't even think about doing it this way in real life!!!

Method of Darts

- Imagine dartboard with circle of radius R inscribed in square
- Area of circle $= \pi R^2$
- Area of square $= (2R)^2 = 4R^2$
- Area of circle / Area of square $= \dfrac{\pi R^2}{4R^2} = \dfrac{\pi}{4}$

Method of Darts

- So, ratio of areas proportional to π
- How to find areas?
  - Suppose we threw darts (**completely randomly**) at dartboard
  - Could count number of darts landing in circle and total number of darts landing in square
  - Ratio of these numbers gives approximation to ratio of areas
  - Quality of approximation increases with number of darts
    - $\pi = 4 \times \frac{\text{\# darts inside circle}}{\text{\# darts thrown}}$

Method of Darts

- Okay, but how in the world do we simulate this experiment on a computer?
  - Decide on length R
  - Generate pairs of random numbers (x, y) so that $-R \leq x, y \leq R$
  - If (x, y) within circle (i.e. if $(x^2+y^2) \leq R^2$), add one to tally for inside circle
  - Lastly, find ratio

263

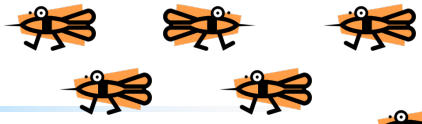
Parallelization Strategies

- What tasks independent of each other?
- What tasks must be performed sequentially?
- Using PCAM parallel algorithm design strategy

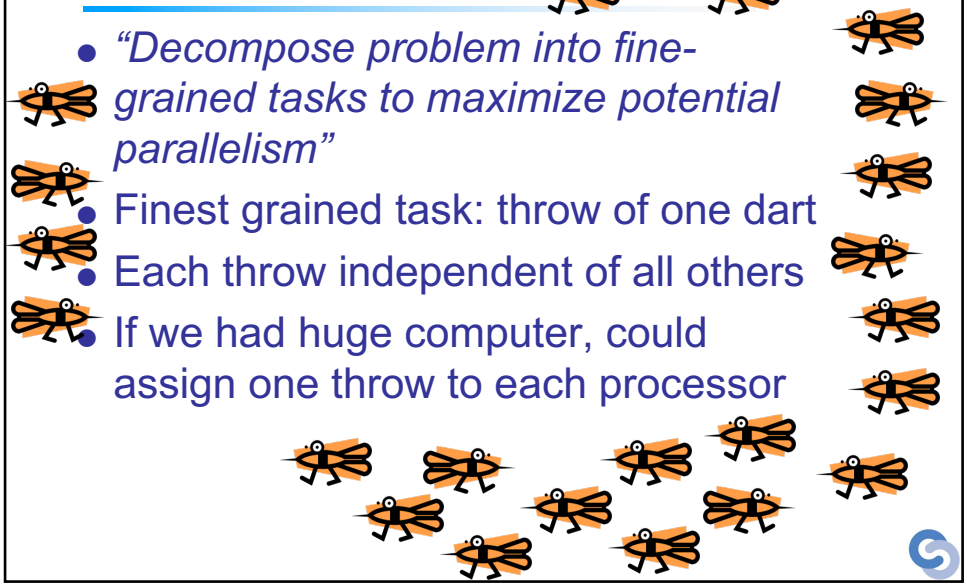
263

264

Partition



- *"Decompose problem into fine-grained tasks to maximize potential parallelism"*
- Finest grained task: throw of one dart
- Each throw independent of all others
- If we had huge computer, could assign one throw to each processor

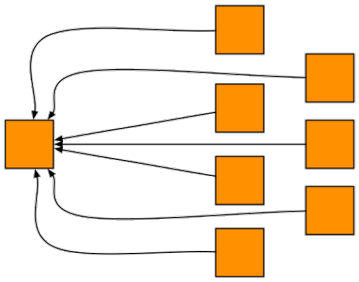


264

Communication

"Determine communication pattern among tasks"

- Each processor throws dart(s) then sends results back to manager process



Agglomeration

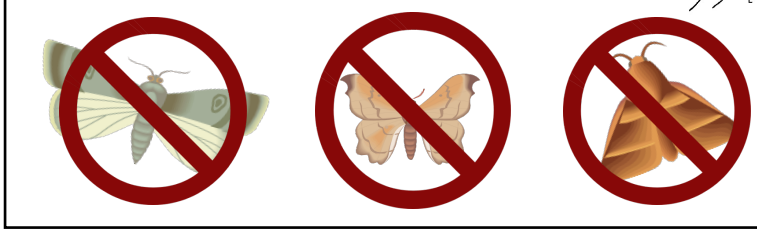
"Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs"

- To get good value of π, must use millions of darts
- We don't have millions of processors available
- Furthermore, communication between manager and millions of worker processors would be very expensive
- Solution: divide up number of dart throws evenly between processors, so each processor does a share of work

Mapping

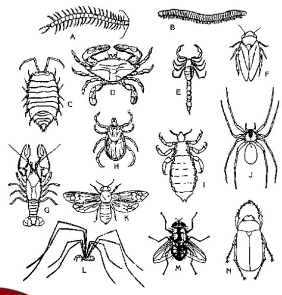
"Assign tasks to processors, subject to tradeoff between communication cost and concurrency"

- Assign role of "manager" to processor 0
- Processor 0 will receive tallies from all the other processors, and will compute final value of π
- Every processor, including manager, will perform equal share of dart throws



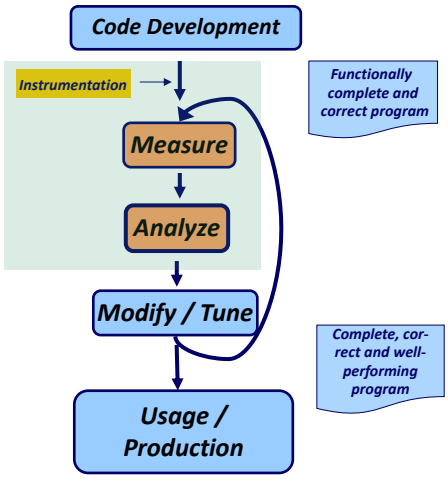
Debugging and Performance Evaluation

- Common errors in parallel programs
- Debugging tools
- Overview of benchmarking and performance measurements

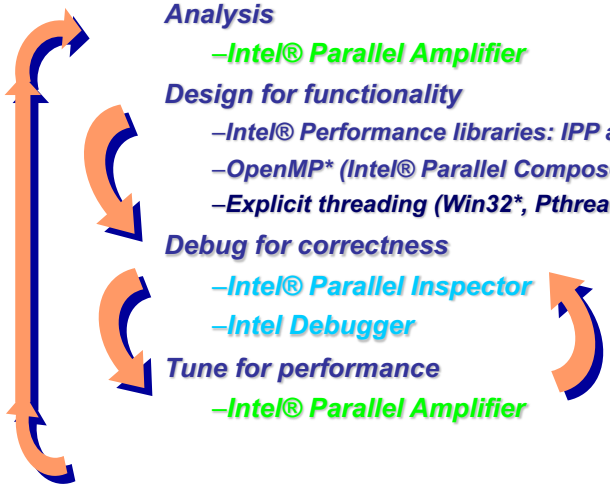


Concepts and Definitions

- The typical performance optimization cycle



Development Cycle



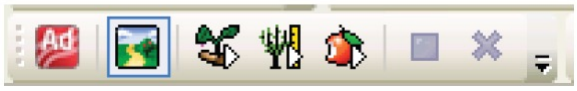
Intel® Parallel Studio

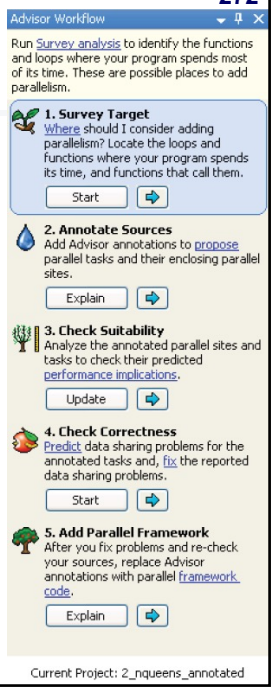
- Decide where to add the parallelism
  - Analyze the serial program
  - Prepare it for parallelism
  - Test the preparations
- Add the parallelism
  - Threads, OpenMP, Cilk, TBB, etc.
- Find logic problems
  - Only fails sometimes
  - Place of failure changes
- Find performance problems

Workflow

- Transforming many serial algorithms into parallel form takes five easy high-level steps
- Often existing algorithms are over-constrained by serial language semantics, and the underlying mathematics has a natural parallel expression if you can just find it

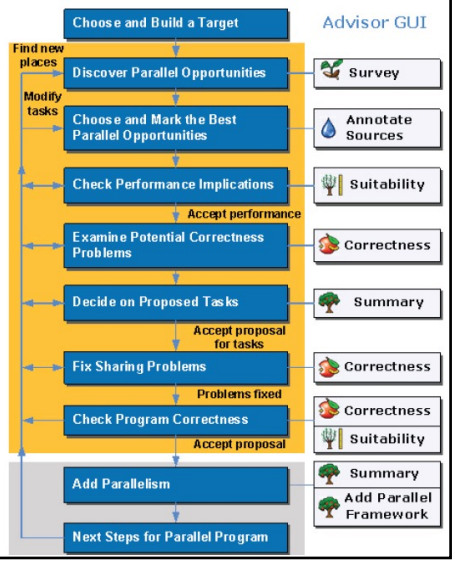
(Advisor toolbar)





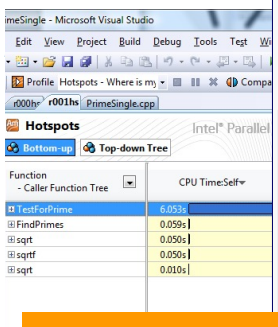
Advisor Overview

- If you look at these steps in more detail, you find decisions you will need to make
- You do not have to choose the perfect answer the first time, so you can go back and modify your choices



Hotspot Analysis

- Use Parallel Am
find hotspots in a



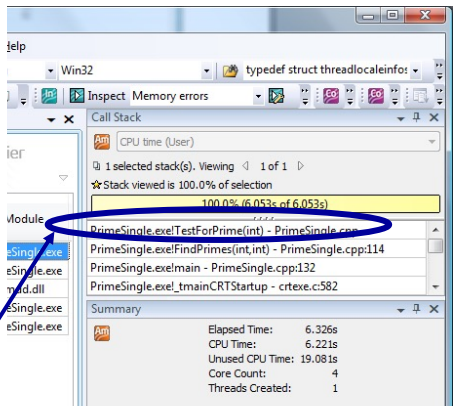

```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor))
            factor ++;

    return (factor > limit);
}
```

```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
```

Identifies the time consuming regions

Analysis - Call Stack

- Inspect the code for the leaf node
- Look for a loop to parallelize
  - If none are found, progress up the call stack until you find a suitable loop or function call to parallelize (FindPrimes)

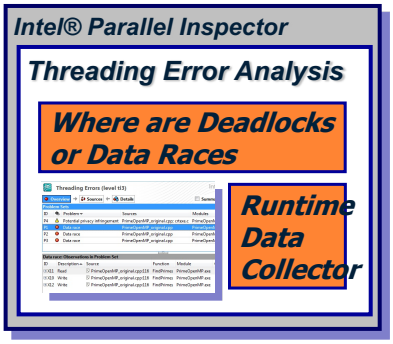


This is the level in the call tree where we need to thread

Used to find proper level in the call-tree to thread

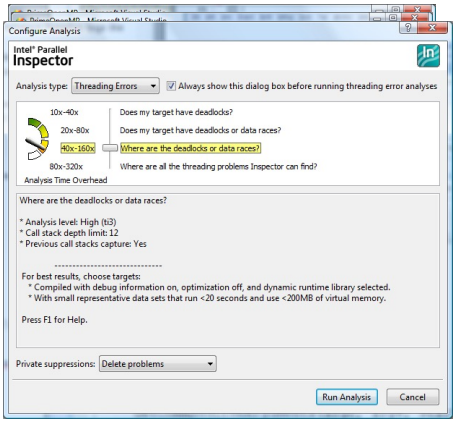
Debugging for Correctness

- Intel® Parallel Inspector pinpoints notorious threading bugs like data races and deadlocks



Intel® Parallel Inspector

- Select info regarding both data races & deadlocks
- View the Overview for Threading Errors
- Select a threading error and inspect the code



277

Motivation

- Developing threaded applications can be a complex task
- New class of problems are caused by the interaction between concurrent threads
  - Data races or storage conflicts
    - More than one thread accesses memory without synchronization
  - Deadlocks
    - Thread waits for an event that will never happen

278

279

Intel® Parallel Inspector

- Debugging tool for threaded software
  - Plug-in to Microsoft* Visual Studio*
- Finds threading bugs in OpenMP*, Intel® Threading Building Blocks, and Win32* threaded software
- Locates bugs quickly that can take days to find using traditional methods and tools
  - Isolates problems, not the symptoms
  - Bug does **not** have to occur to find it!

279

280

Parallel Inspector: Analysis

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

Code path must be executed to be analyzed

280

281

Parallel Inspector: Before You Start

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

Workload selection is important!

281

282

Workload Guidelines

- Execute problem code once per thread to be identified
- Use smallest possible working data set
  - Minimize data set size
    - Smaller image sizes
  - Minimize loop iterations or time steps
    - Simulate minutes rather than days
  - Minimize update rates
    - Lower frames per second

Finds threading errors faster!

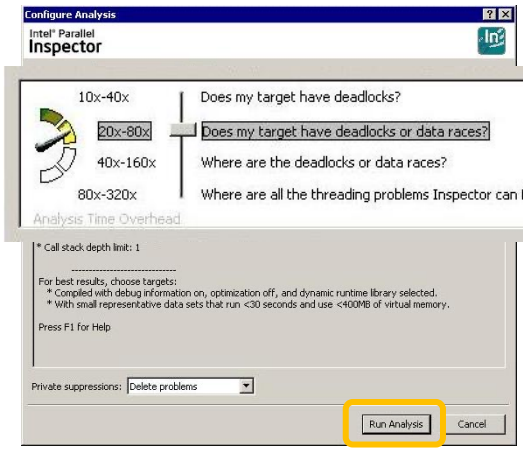
282

Binary Instrumentation

- Build with supported compiler
- Running the application
  - Must be run from within Parallel Inspector
  - Application is instrumented when executed
  - External DLLs are instrumented as used

Starting Parallel Inspector

- The Configure Analysis window pops up



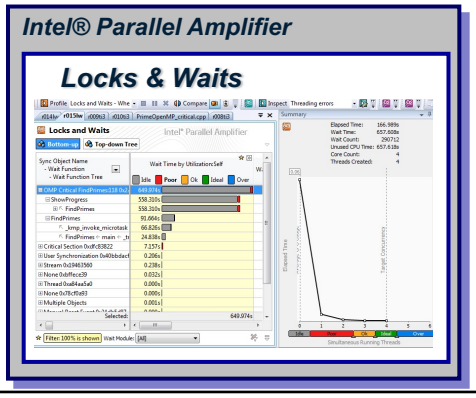
Select the level of analysis to be carried out by Parallel

e analysis, ough the e longer the e

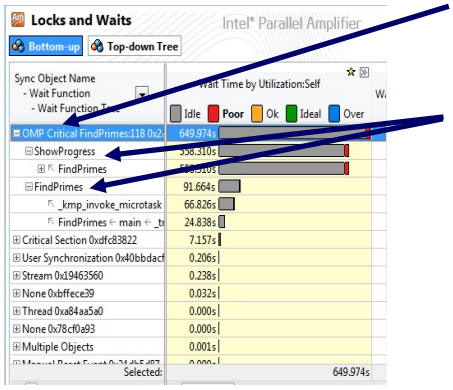
Click Run Analysis

Tuning for Performance

Parallel Amplifier (Locks & Waits) pinpoints performance bottlenecks in threaded applications



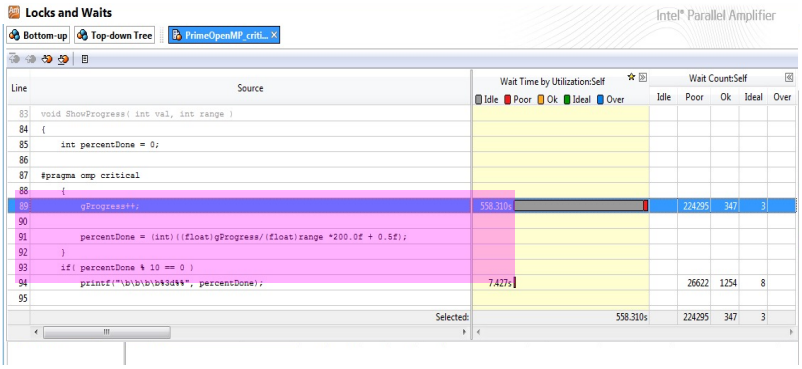
Parallel Amplifier - Locks & Waits



- Graph shows significant portion of time in idle condition as result of critical section
- FindPrimes() & ShowProgress() are both excessively impacted by the idle time occurring in the critical section

Parallel Amplifier - Locks & Waits

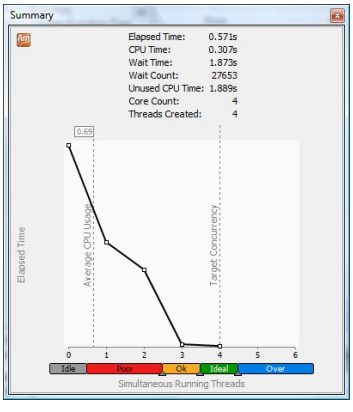
- ShowProgress() consumes 558/657 (85%) of the time idling in a critical section
- Double Click ShowProgress() in largest critical section to



287

Parallel Amplifier Summary

- Elapsed Time shows
  .571 sec
- Wait Time/ Core Count = 1.87/4 =.47 sec
- Waiting 82% of elapsed time in critical section
- Most of the time 1 core and occasionally 2 are occupied

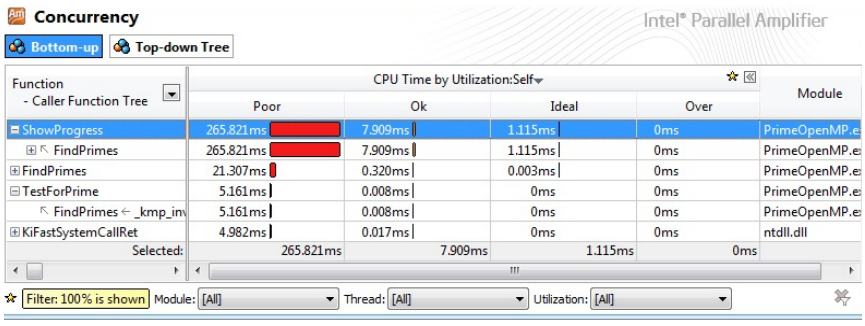


288

289

Parallel Amplifier - Concurrency

- Function - Caller Function Tree
- ShowProgress is called from FindPrimes and represent the biggest reason concurrency is poor

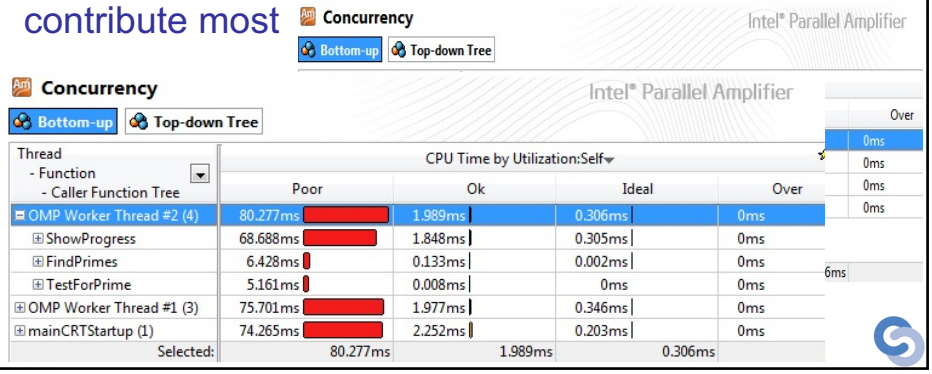


289

290

Parallel Amplifier - Concurrency

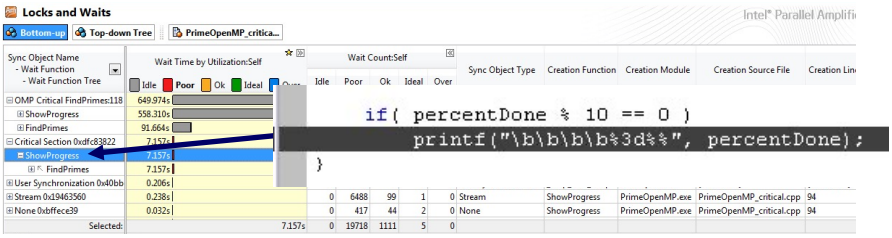
- Thread –Function –Caller Function Tree
- This view shows how each thread contributes to the concurrency issue
- Expanding any thread will reveal the functions that contribute most



290

Performance

- Double Click ShowProgress in second largest critical section
- This implementation has implicit synchronization calls - printf
- This limits scaling performance due to the resulting context switches



Back to the design stage

291