Debugging, Profiling, Performance Analysis, Optimization, Load Balancing

APP – 13.12.2010

Emil Slusanschi
emil.slusanschi@cs.pub.ro

Foster's Design Methodology

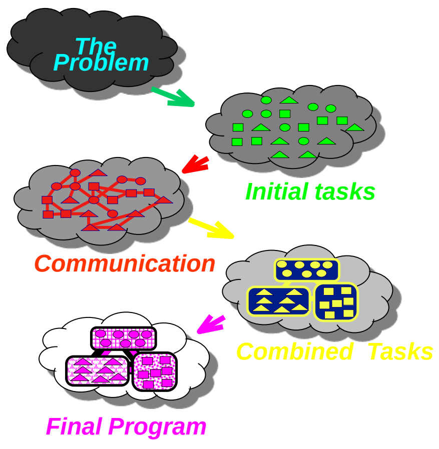
- From *Designing and Building Parallel Programs* by Ian Foster
- Four Steps:
  - **Partitioning**
    - Dividing computation and data
  - **Communication**
    - Sharing data between computations
  - **Agglomeration**
    - Grouping tasks to improve performance
  - **Mapping**
    - Assigning tasks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

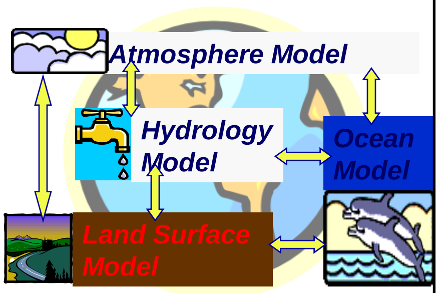


Parallel Programming Models

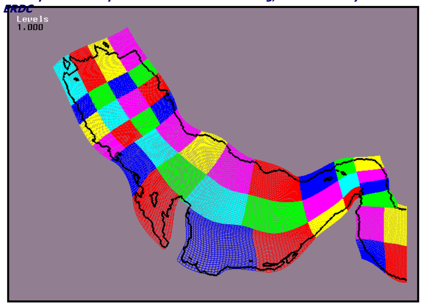
- Functional Decomposition
  - Task parallelism
  - Divide the computation, then associate the data
  - Independent tasks of the same problem
- Data Decomposition
  - Same operation performed on different data
  - Divide data into pieces, then associate computation

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 $= (2R)^2 = 4R^2$
- $\dfrac{\text{Area of circle}}{\text{Area of square}} = \dfrac{\pi R^2}{4R^2} = \dfrac{\pi}{4}$

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
  - $\pi = 4 \times \dfrac{\text{\# darts inside circle}}{\text{\# darts thrown}}$

Method of Darts

- Okay, Rebecca, 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

Parallelization Strategies

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

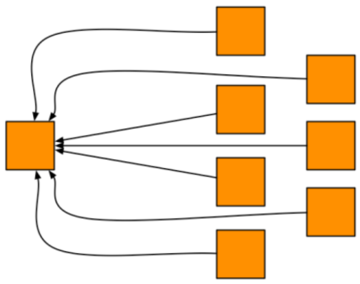
Partition

- *"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

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"

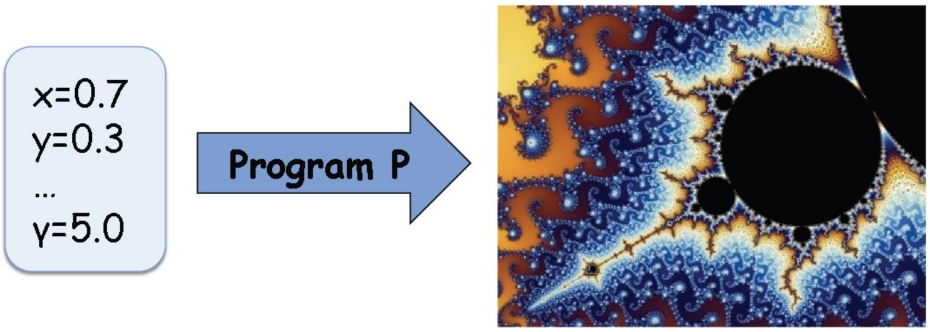
- Assign role of "manager" to processor 0
- Processor 0 will receive tallies from all the other processors, and will compute final value of π
- Every processor, including manager, will perform equal share of dart throws

Parallel Correctness Challenges

- Parallel programming presents a number of new challenges to writing correct software
  - New kinds of bugs: data races, deadlocks, etc.
  - More difficult to test programs and find bugs
  - More difficult to reproduce errors
- Key Difficulty: Potential non-determinism
  - Order in which threads execute can change from run to run
  - Some runs are correct while others hit bugs

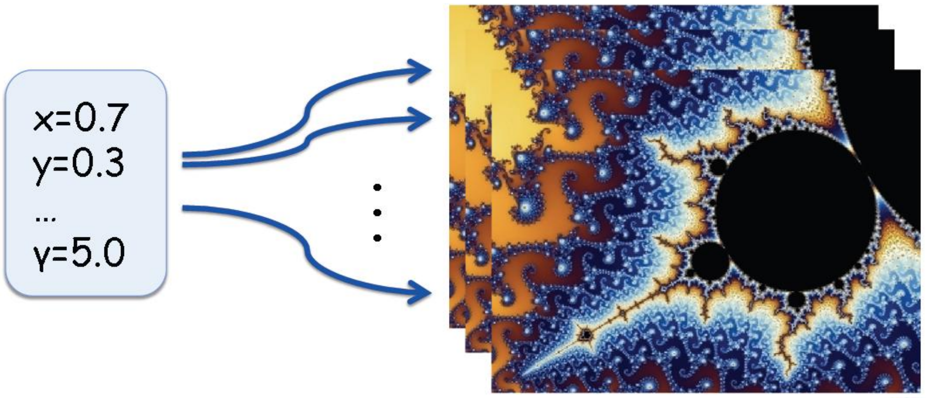
Parallel Correctness Challenges

- For sequential programs, we typically expect that same input → same output



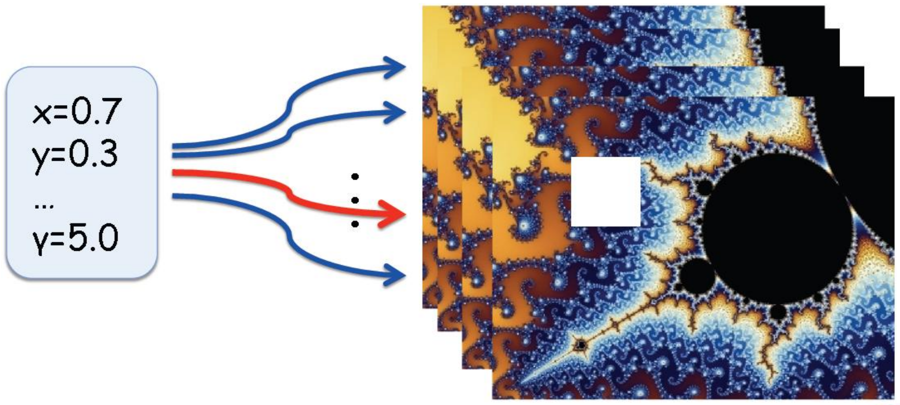
Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run



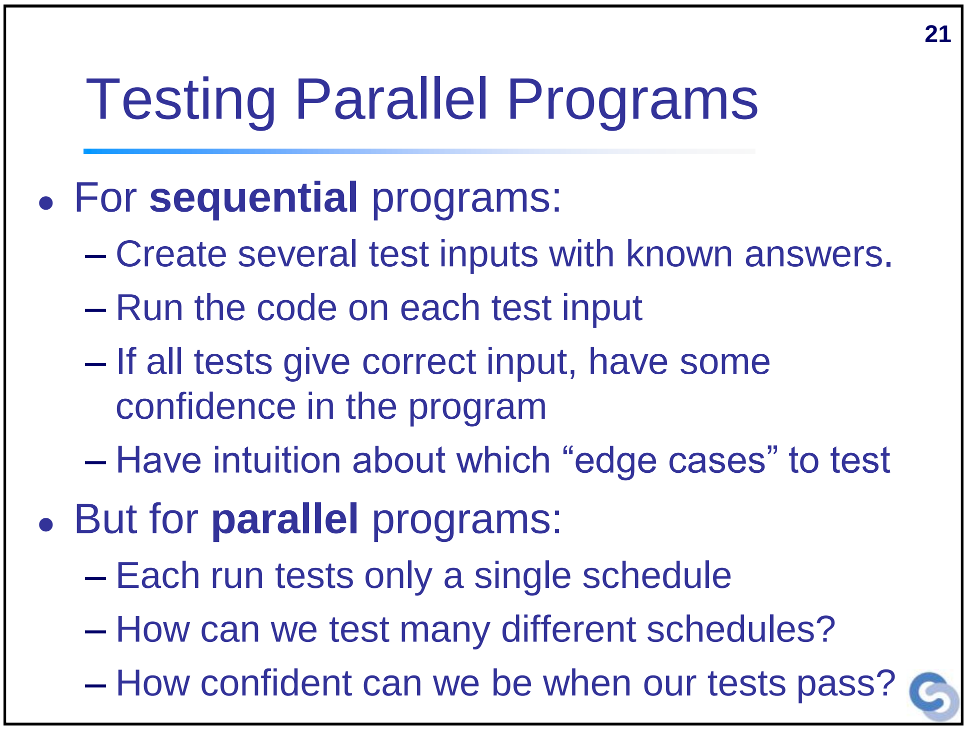
Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run



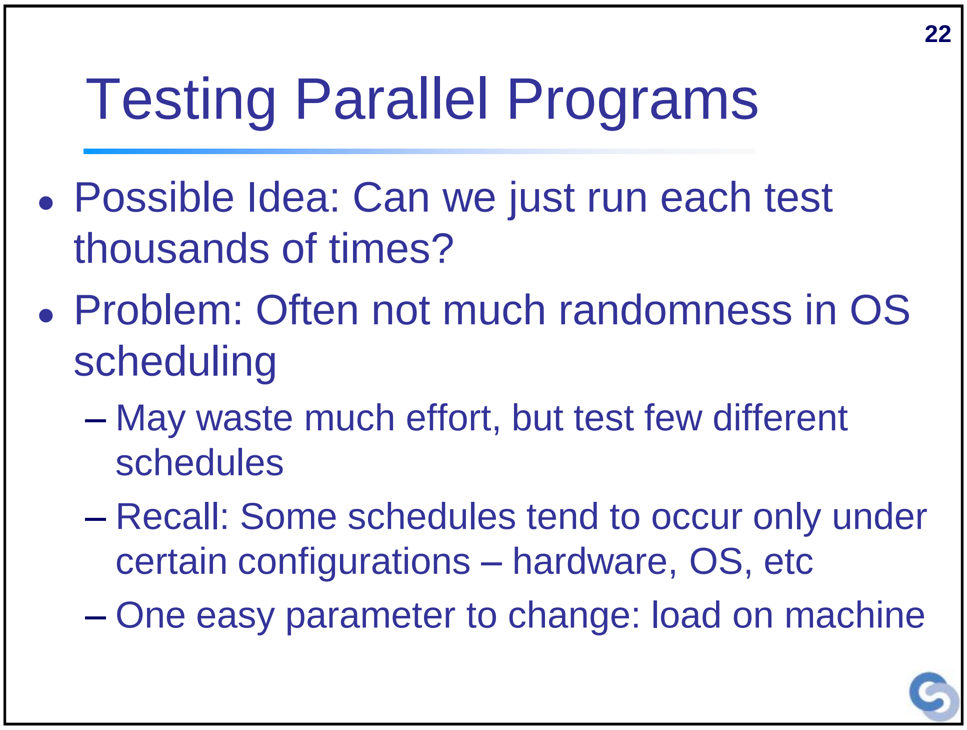
Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run
- A bug may occur under only rare schedules.
  – In 1 run in 1000 or 10,000 or …
- May occur only under some configurations:
  – Particular OS scheduler
  – When machine is under heavy load.
  – Only when debugging/logging is turned off!



Testing Parallel Programs

- For **sequential** programs:
  - – Create several test inputs with known answers.
  - – Run the code on each test input
  - – If all tests give correct input, have some confidence in the program
  - – Have intuition about which “edge cases” to test
- But for **parallel** programs:
  - – Each run tests only a single schedule
  - – How can we test many different schedules?
  - – How confident can we be when our tests pass?



Testing Parallel Programs

- Possible Idea: Can we just run each test thousands of times?
- Problem: Often not much randomness in OS scheduling
  - – May waste much effort, but test few different schedules
  - – Recall: Some schedules tend to occur only under certain configurations – hardware, OS, etc
  - – One easy parameter to change: load on machine

Stress Testing

- Idea: Test parallel program while oversubscribing the machine
  - On a 4-core system, run with 8 or 16 threads
  - Run several instances of the program at a time
  - Increase size to overflow cache/memory
  - Effect: Timing of threads will change, giving different thread schedules
- Pro: Very simple idea, easy to implement
  - And often works!

Noise Making / Random Scheduling

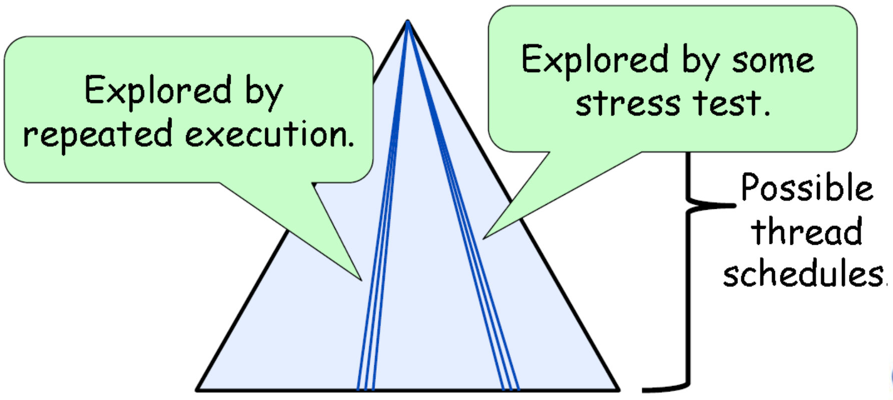
- Idea: Run with random thread schedules
  - E.g., insert code like:
    - if (rand() < 0.01) usleep(100)
    - if (rand() < 0.01) yield()
  - Can add to only “suspicious” or “tricky” code.
  - Or use tool to seize control of thread scheduling.
- Pros: Still fairly simple and often effective.
  - Explores different schedules than stress testing.
  - Many tools can perform this automatically

Noise Making / Random Scheduling

- IBM’s ConTest: Noise-making for Java
  - Clever heuristics about where to insert delays
- Berkeley’s Thrille (C + pthreads) and
- CalFuzzer (Java) do simple random scheduling
  - Extensible: Write testing scheduler for your app
- Microsoft Research’s Cuzz (for .NET)
  - New random scheduling algorithm with probabilistic guarantees for finding bugs – available soon
- Many of these tools provide replay – same random number seed → same schedule

Limitations of Random Scheduling

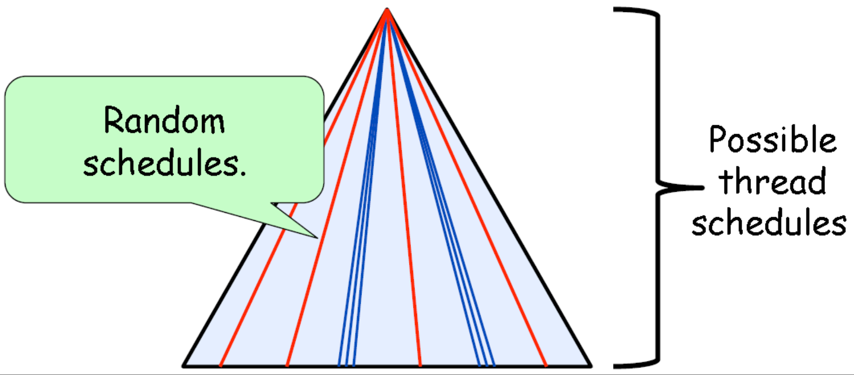
- Parallel programs have huge number of schedules – exponential in length of a run


Explored by repeated execution.
Explored by some stress test.
Possible thread schedules.

Limitations of Random Scheduling

- Parallel programs have huge number of schedules – exponential in length of a run

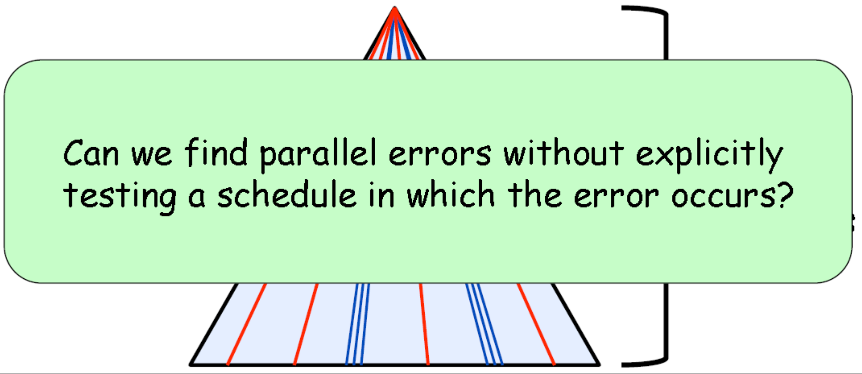
Vast majority of schedules will never be tested.



Limitations of Random Scheduling

- Parallel programs have huge number of schedules – exponential in length of a run

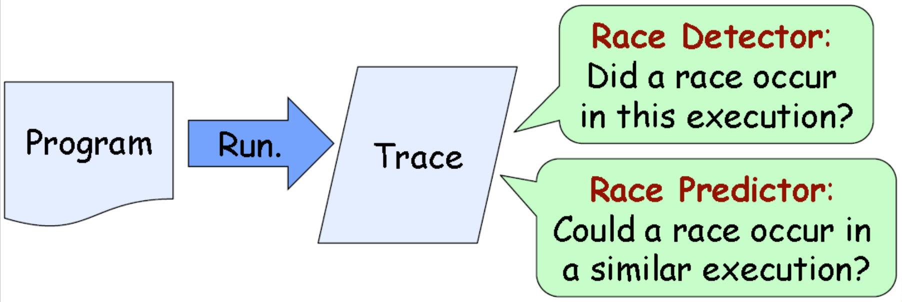
Vast majority of schedules will never be tested.



Can we find parallel errors without explicitly testing a schedule in which the error occurs?

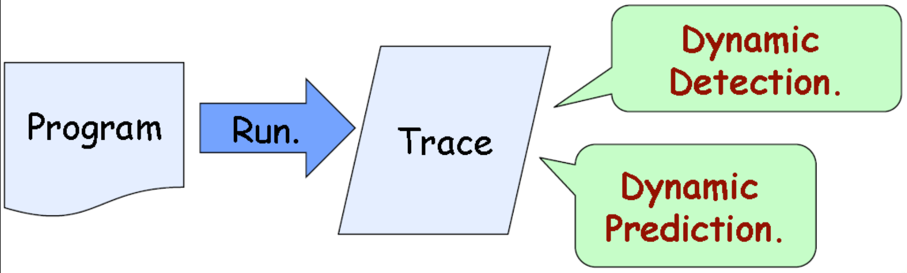
Detecting/Predicting Parallel Bugs

- Say we observe a test run of a parallel program that doesn't obviously fail
- **Key Question**: Can we find possible parallel bugs by examining the execution?


Program
Run.
Trace
Race Detector: Did a race occur in this execution?
Race Predictor: Could a race occur in a similar execution?

Detecting/Predicting Parallel Bugs

- Techniques/tools exist for:
  - Data races
  - Atomicity violations
  - Deadlocks
  - Memory consistency errors



Data Race Detection / Prediction

- 20+ years of research on race detection
- Happens-Before Race Detection [Schonberg ’89]:
  - Do two accesses to a variable occur, at least one a write, with no intervening synchronization?
  - No false warnings
- Lockset Race Prediction [Savage, et al., ’97]:
  - Does every access to a variable hold a common lock?
  - Efficient, but many false warnings
- Hybrid Race Prediction [O’Callahan, Choi, 03]:
  - Combines Lockset with Happens-Before for better performance and fewer false warnings vs. Lockset

Coverage vs. False Warnings

- **False Warning**: Tool reports a data race, but the race cannot happen in a real run
- **Coverage**: How many of the real data races does a tool report?
- Hybrid race prediction:
  - Better coverage but more false warnings
- Happens-Before race detection:
  - Fewer false warnings (still some, in practice) and less coverage

Dynamic Data Race Tools

- Intel Thread Checker for C + pthreads
  - Happens-Before race detection
- Valgrind-based tools for C + pthreads
  - Helgrind and DRD (Happens-Before)
  - ThreadSanitizer (Hybrid)
- CHESS performs race detection for .NET
- CalFuzzer and Thrille: hybrid race detection for Java and C + pthreads

Static Analysis

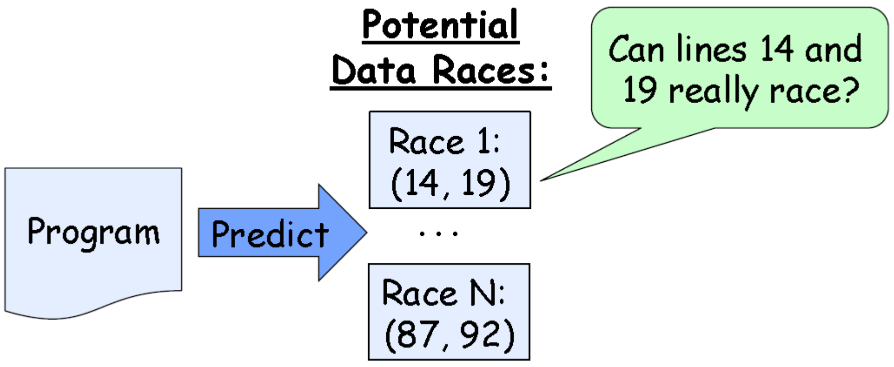
- Have only discussed dynamic analyses
  - Examine a real run/trace of a program
- Static analyses predict data races, deadlocks, etc., **without** running a program
  - Only examine the source code
  - Area of active research for ~20 years
  - Potentially much better coverage than dynamic analysis – examines all possible runs
  - But typically also more false warnings
- CHORD: static race and deadlock prediction

Active Random Testing Overview

- **Problem**: Random testing can be very effective for parallel programs, but can miss many potential bugs
- **Problem**: Predictive analyses find many bugs, but can have false warnings
  - Time consuming and difficult to examine reported bugs and determine whether or not they are real
- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug

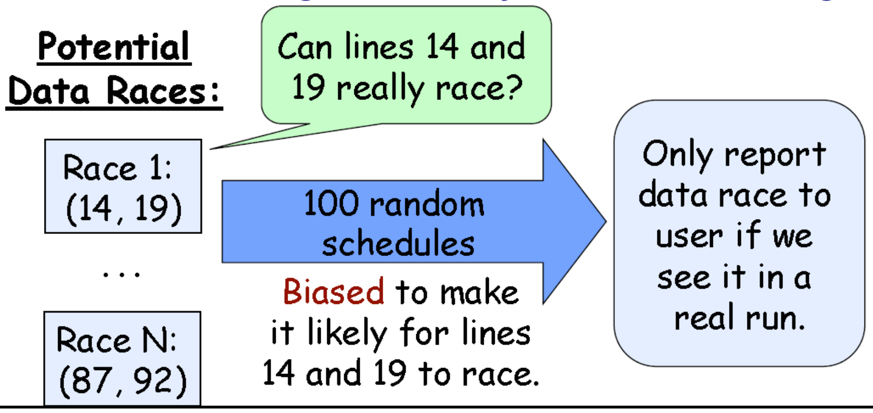
Active Random Testing

- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug



Active Random Testing

- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug

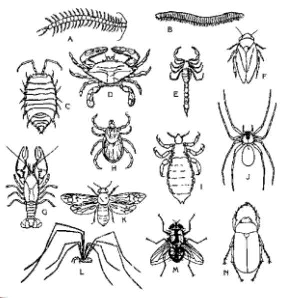

Potential Data Races:
Can lines 14 and 19 really race?
Race 1: (14, 19)
. . .
Race N: (87, 92)
100 random schedules
Biased to make it likely for lines 14 and 19 to race.
Only report data race to user if we see it in a real run.

Active Random Testing

- CalFuzzer is an extensible, open-source tool for active testing of Java programs
  - For data races, atomicity bugs, and deadlocks.
  - RaceFuzzer is the active testing algorithm for data races – will show by example.
- Thrille for C + pthreads.
  - For data races.
- And UPC-Thrille for Unified Parallel C.
  - Part of the Berkeley UPC system

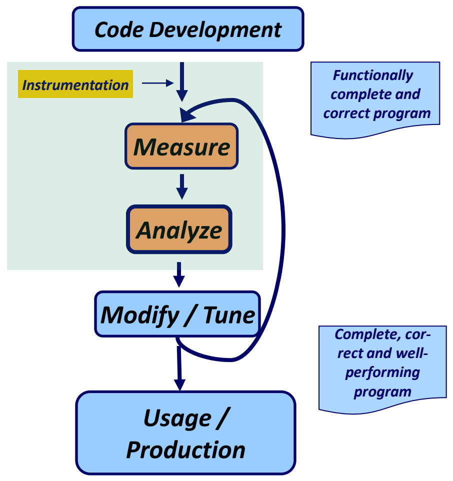
Debugging and Performance Evaluation

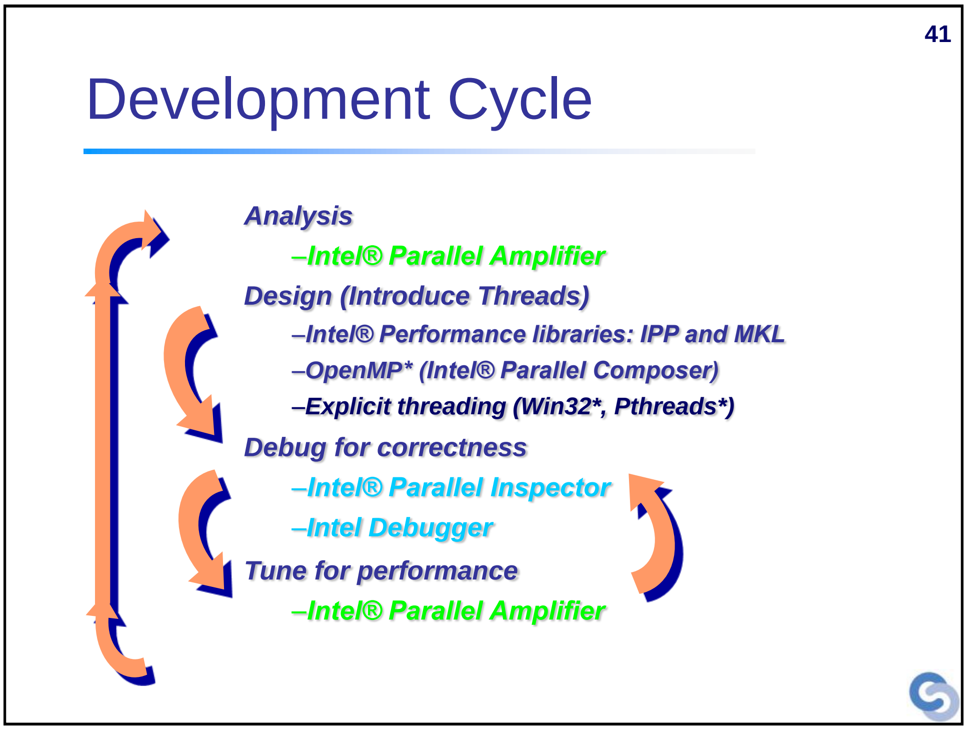
- Common errors in parallel programs
- Debugging tools
- Overview of benchmarking and performance measurements



Concepts and Definitions

- The typical performance optimization cycle


Code Development
Instrumentation
Functionally complete and correct program
Measure
Analyze
Modify / Tune
Complete, correct and well-performing program
Usage / Production



Development Cycle

Analysis

–Intel® Parallel Amplifier

Design (Introduce Threads)

–Intel® Performance libraries: IPP and MKL

–OpenMP (Intel® Parallel Composer)*

–Explicit threading (Win32, Pthreads*)*

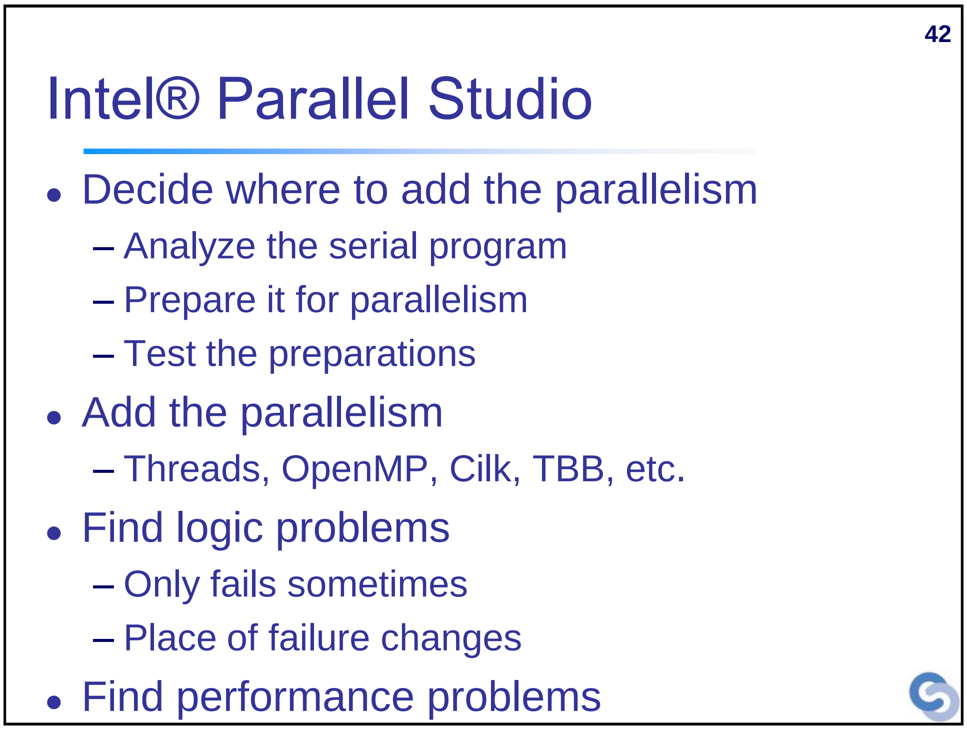
Debug for correctness

–Intel® Parallel Inspector

–Intel Debugger

Tune for performance

–Intel® Parallel Amplifier



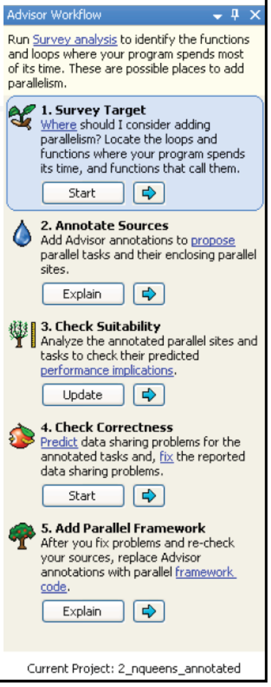
Intel® Parallel Studio

- 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 Workflow

Run Survey analysis to identify the functions and loops where your program spends most of its time. These are possible places to add parallelism.

1. Survey Target
Where should I consider adding parallelism? Locate the loops and functions where your program spends its time, and functions that call them.
Start

2. Annotate Sources
Add Advisor annotations to propose parallel tasks and their enclosing parallel sites.
Explain

3. Check Suitability
Analyze the annotated parallel sites and tasks to check their predicted performance implications.
Update

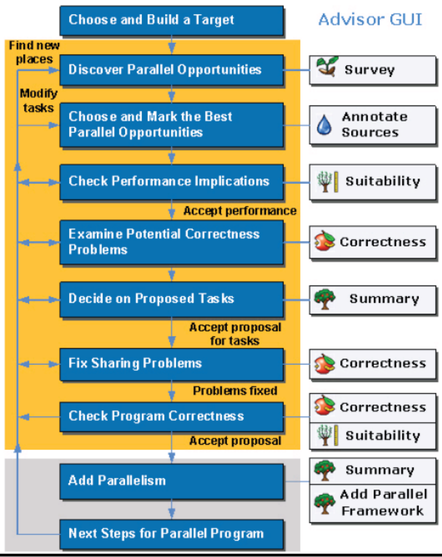
4. Check Correctness
Predict data sharing problems for the annotated tasks and, fix the reported data sharing problems.
Start

5. Add Parallel Framework
After you fix problems and re-check your sources, replace Advisor annotations with parallel framework code.
Explain

Current Project: 2_nqueens_annotated

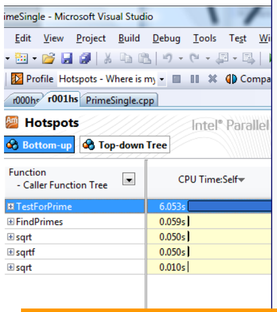
Advisor Overview

- 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 ap



imeSingle - Microsoft Visual Studio

Profile Hotspots - Where is my

r000hs r001hs PrimeSingle.cpp

Hotspots

Function - Caller Function Tree	CPU Time:Self
TestForPrime	6.053s
FindPrimes	0.059s
sqrt	0.050s
sqrtf	0.050s
sqrt	0.010s

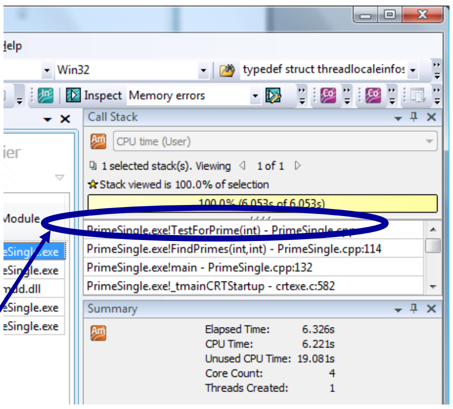
```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor))
            factor ++;

    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;
        ShowProgress(i, range);
```

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)



Call Stack

CPU time (User)

1 selected stack(s). Viewing 1 of 1

Stack viewed is 100.0% of selection

100.0% (6.053s of 6.053s)

```
PrimeSingle.exe!TestForPrime(int) - PrimeSingle.cpp
PrimeSingle.exe!FindPrimes(int,int) - PrimeSingle.cpp:114
PrimeSingle.exe!main - PrimeSingle.cpp:132
PrimeSingle.exe!_tmainCRTStartup - crtexe.c:582
```

Summary

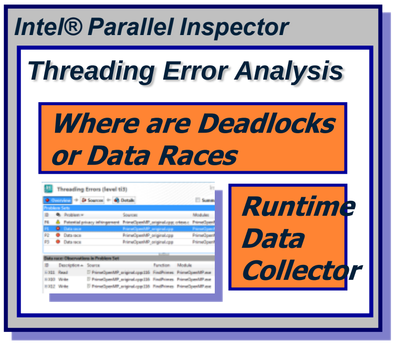
```
Elapsed Time:        6.326s
CPU Time:            6.221s
Unused CPU Time:    19.081s
Core Count:              4
Threads Created:         1
```

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

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

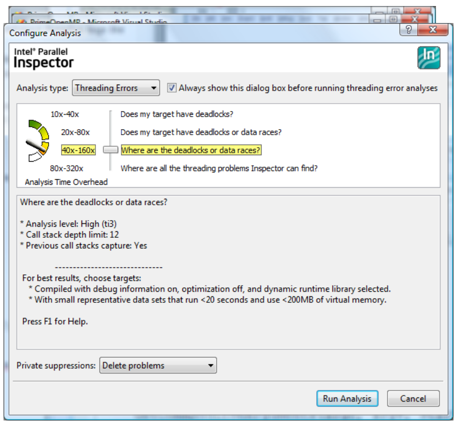
Debugging for Correctness

- 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



Analysis type: Threading Errors

[✓] Always show this dialog box before running threading error analyses

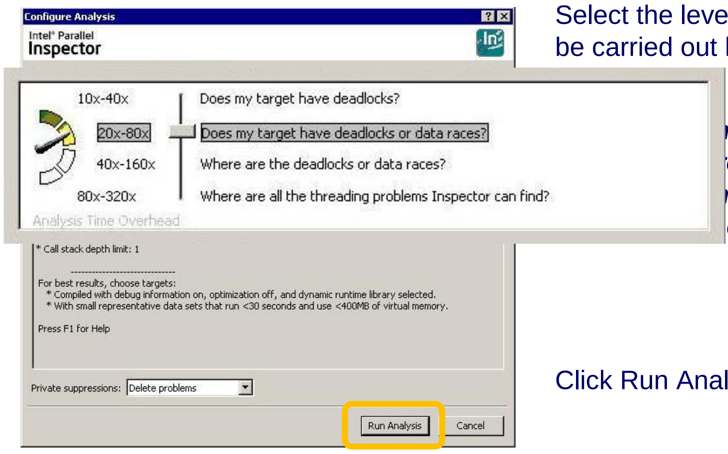
Does my target have deadlocks?
Does my target have deadlocks or data races?
Where are the deadlocks or data races?
Where are all the threading problems Inspector can find?

```
Where are the deadlocks or data races?

* Analysis level: High (ti3)
* Call stack depth limit: 12
* Previous call stacks capture: Yes

          ----------------------------
For best results, choose targets:
  * Compiled with debug information on, optimization off, and dynamic runtime library selected.
  * With small representative data sets that run <20 seconds and use <200MB of virtual memory.

Press F1 for Help.
```

Private suppressions: Delete problems

Starting Parallel Inspector

- The Configure Analysis window pops up



Configure Analysis

Intel® Parallel Inspector

Analysis Time Overhead	
10x-40x	Does my target have deadlocks?
20x-80x	Does my target have deadlocks or data races?
40x-160x	Where are the deadlocks or data races?
80x-320x	Where are all the threading problems Inspector can find?

* Call stack depth limit: 1

For best results, choose targets:
 * Compiled with debug information on, optimization off, and dynamic runtime library selected.
 * With small representative data sets that run <30 seconds and use <400MB of virtual memory.

Press F1 for Help

Private suppressions: Delete problems

Select the level of analysis to be carried out by Parallel

e analysis, ough the e longer the e

Click Run Analysis

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

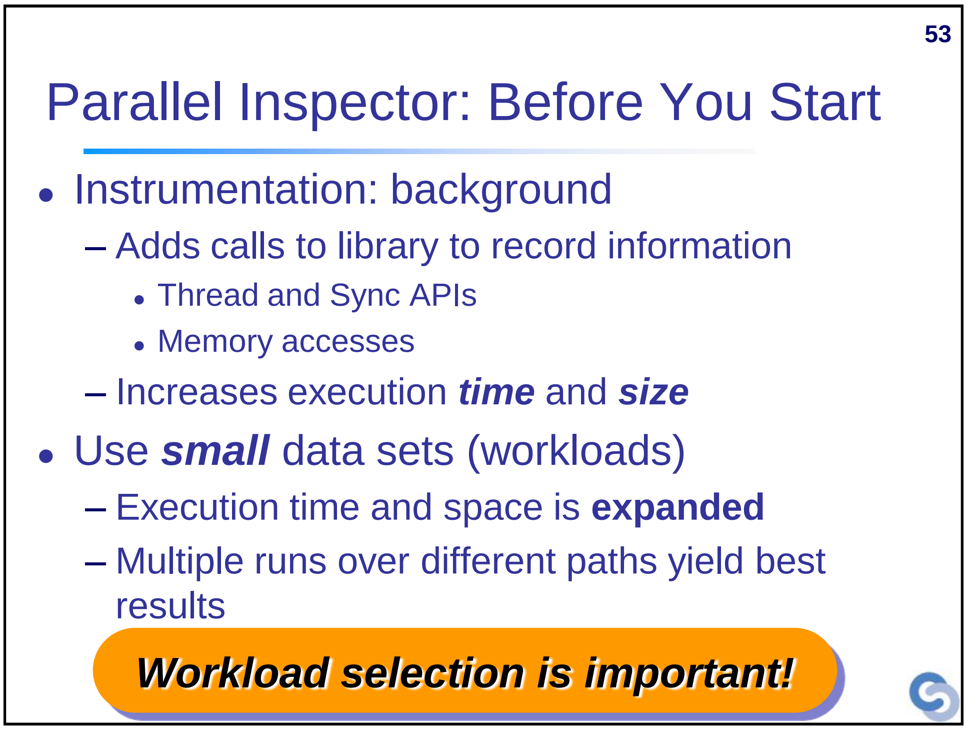
Intel® Parallel Inspector

- 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!

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

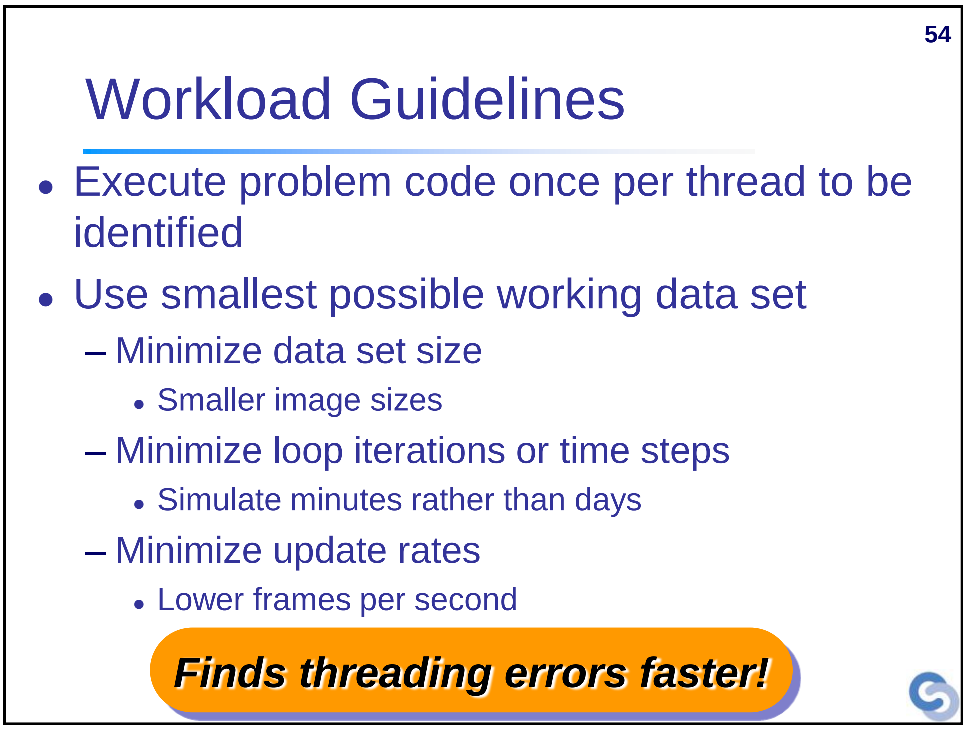


53

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!



54

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

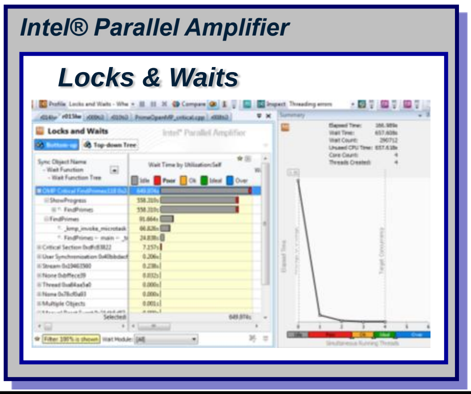
Finds threading errors faster!

Binary Instrumentation

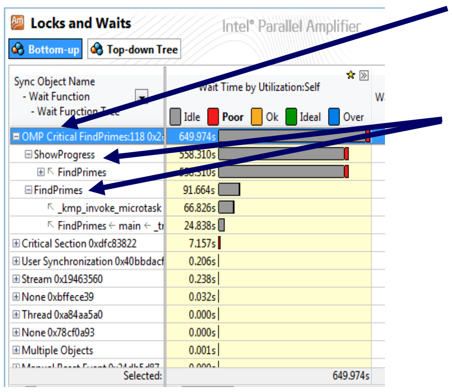
- Build with supported compiler
- Running the application
  - Must be run from within Parallel Inspector
  - Application is instrumented when executed
  - External DLLs are instrumented as used

Tuning for Performance

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



Parallel Amplifier - Locks & Waits



Locks and Waits — Intel® Parallel Amplifier

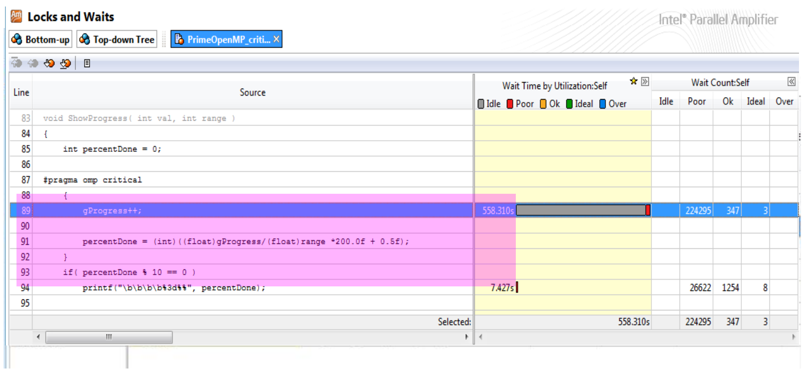
Bottom-up | Top-down Tree

Sync Object Name - Wait Function - Wait Function Tree	Wait Time by Utilization:Self (Idle, Poor, Ok, Ideal, Over)
⊟ OMP Critical FindPrimes:118 0x2	649.974s
⊟ ShowProgress	558.310s
⊞ ↖ FindPrimes	558.310s
⊟ FindPrimes	91.664s
↖ _kmp_invoke_microtask	66.826s
↖ FindPrimes ← main ← _tr	24.838s
⊞ Critical Section 0xdfc83822	7.157s
⊞ User Synchronization 0x40bbdacf	0.206s
⊞ Stream 0x19463560	0.238s
⊞ None 0xbffece39	0.032s
⊞ Thread 0xa84aa5a0	0.000s
⊞ None 0x78cf0a93	0.000s
⊞ Multiple Objects	0.001s
Selected:	649.974s

- 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



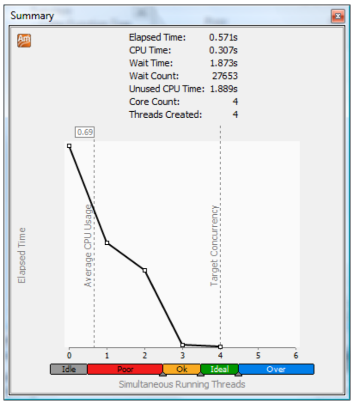
Locks and Waits — Intel® Parallel Amplifier

Bottom-up | Top-down Tree | PrimeOpenMP_criti... ✕

Line	Source	Wait Time by Utilization:Self	Wait Count:Self Idle	Poor	Ok	Ideal	Over
83	`void ShowProgress( int val, int range )`						
84	`{`						
85	`    int percentDone = 0;`						
86							
87	`#pragma omp critical`						
88	`    {`						
89	`        gProgress++;`	558.310s		224295	347	3	
90							
91	`        percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);`						
92	`    }`						
93	`    if( percentDone % 10 == 0 )`						
94	`        printf("\b\b\b\b%3d%%", percentDone);`	7.427s		26622	1254	8	
95							
	Selected:	558.310s		224295	347	3	

Parallel Amplifier Summary

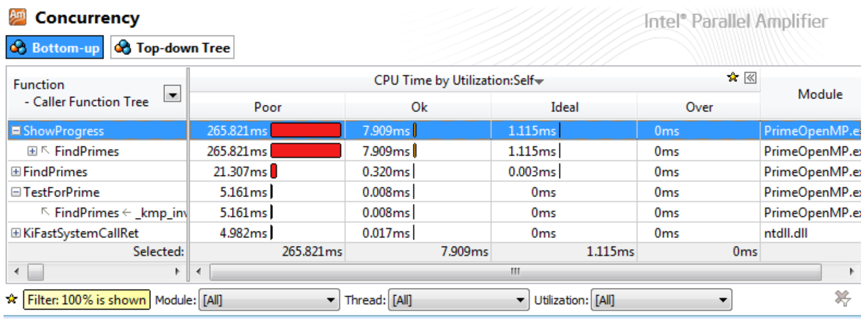
- Elapsed Time shows
  .571 sec
- Wait Time/ Core Count = 1.87/4 =.47 sec
- Waiting 82% of elapsed time in critical section
- Most of the time 1 core and occasionally 2 are occupied


Summary

Elapsed Time:	0.571s
CPU Time:	0.307s
Wait Time:	1.873s
Wait Count:	27653
Unused CPU Time:	1.889s
Core Count:	4
Threads Created:	4

Parallel Amplifier - Concurrency

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

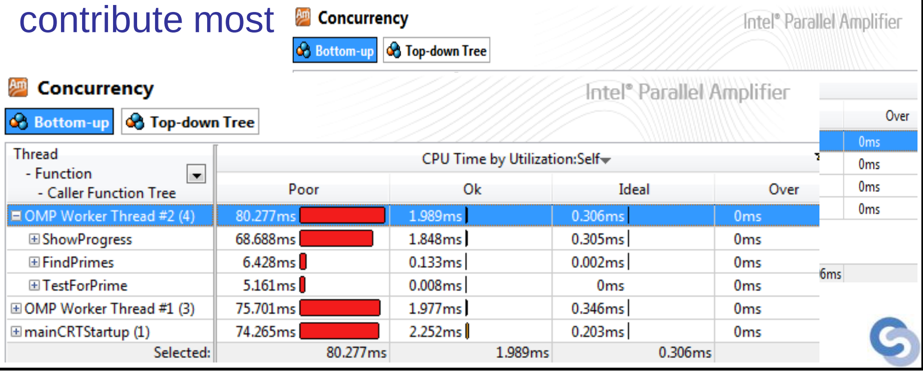

Concurrency

Function - Caller Function Tree	CPU Time by Utilization:Self Poor	Ok	Ideal	Over	Module
⊟ShowProgress	265.821ms	7.909ms	1.115ms	0ms	PrimeOpenMP.e
⊞ ↖ FindPrimes	265.821ms	7.909ms	1.115ms	0ms	PrimeOpenMP.e
⊞FindPrimes	21.307ms	0.320ms	0.003ms	0ms	PrimeOpenMP.e
⊟TestForPrime	5.161ms	0.008ms	0ms	0ms	PrimeOpenMP.e
↖ FindPrimes ← _kmp_inv	5.161ms	0.008ms	0ms	0ms	PrimeOpenMP.e
⊞KiFastSystemCallRet	4.982ms	0.017ms	0ms	0ms	ntdll.dll
Selected:	265.821ms	7.909ms	1.115ms	0ms	

Filter: 100% is shown Module: [All] Thread: [All] Utilization: [All]

Parallel Amplifier - Concurrency

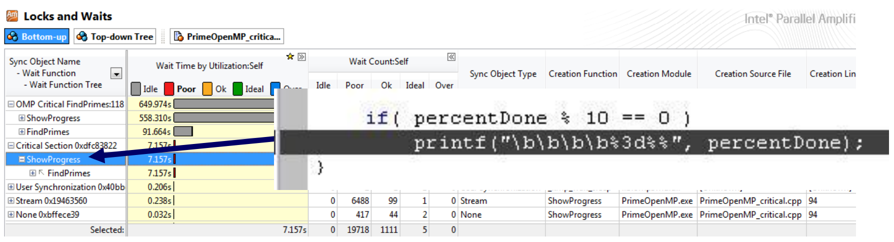
- Thread –Function –Caller Function Tree
- This view shows how each thread contributes to the concurrency issue
- Expanding any thread will reveal the functions that contribute most



Thread - Function - Caller Function Tree	CPU Time by Utilization:Self			
	Poor	Ok	Ideal	Over
⊟ OMP Worker Thread #2 (4)	80.277ms	1.989ms	0.306ms	0ms
⊞ ShowProgress	68.688ms	1.848ms	0.305ms	0ms
⊞ FindPrimes	6.428ms	0.133ms	0.002ms	0ms
⊞ TestForPrime	5.161ms	0.008ms	0ms	0ms
⊞ OMP Worker Thread #1 (3)	75.701ms	1.977ms	0.346ms	0ms
⊞ mainCRTStartup (1)	74.265ms	2.252ms	0.203ms	0ms
Selected:	80.277ms	1.989ms	0.306ms	

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



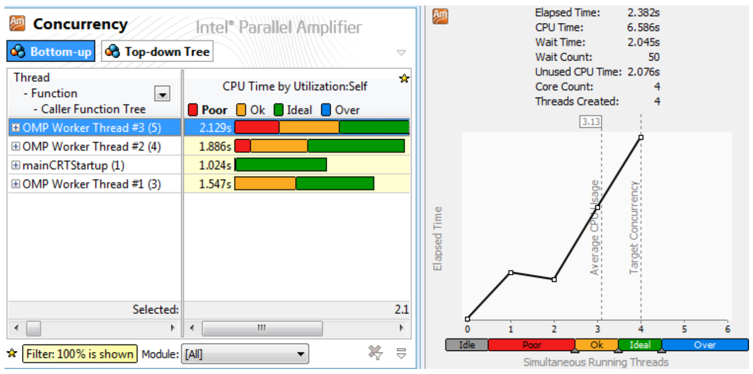
Sync Object Name - Wait Function - Wait Function Tree	Wait Time by Utilization:Self	Wait Count:Self					Sync Object Type	Creation Function	Creation Module	Creation Source File	Creation Lin
	🟥 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Idle	Poor	Ok	Ideal	Over					
⊟ OMP Critical FindPrimes:118	649.974s										
⊞ ShowProgress	558.310s										
⊞ FindPrimes	91.664s										
⊟ Critical Section 0xdfc83822	7.157s										
⊟ ShowProgress	7.157s										
⊞ FindPrimes	7.157s										
⊞ User Synchronization 0x40bb	0.206s										
⊞ Stream 0x19463560	0.238s	0	6488	99	1	0	Stream	ShowProgress	PrimeOpenMP.exe	PrimeOpenMP_critical.cpp	94
⊞ None 0xbffece39	0.032s	0	417	44	2	0	None	ShowProgress	PrimeOpenMP.exe	PrimeOpenMP_critical.cpp	94
Selected:	7.157s	0	19718	1111	5	0					

```
    if( percentDone % 10 == 0 )
        printf("\b\b\b\b%3d%%", percentDone);
}
```

Back to the design stage

Load Balance Analysis

- Use Parallel Amplifier – Concurrency Analysis
- Select the “Thread –Function -Caller Function Tree”
- Observe that the 4 threads do unequal amounts of work



Thread - Function - Caller Function Tree	CPU Time by Utilization:Self 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
⊞ OMP Worker Thread #3 (5)	2.129s
⊞ OMP Worker Thread #2 (4)	1.886s
⊞ mainCRTStartup (1)	1.024s
⊞ OMP Worker Thread #1 (3)	1.547s
Selected:	2.1

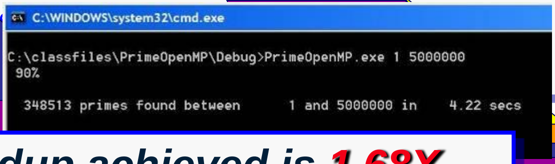
Filter: 100% is shown Module: [All]

Elapsed Time:	2.382s
CPU Time:	6.586s
Wait Time:	2.045s
Wait Count:	50
Unused CPU Time:	2.076s
Core Count:	4
Threads Created:	4

Fixing the Load Imbalance

- Distribute the work more evenly

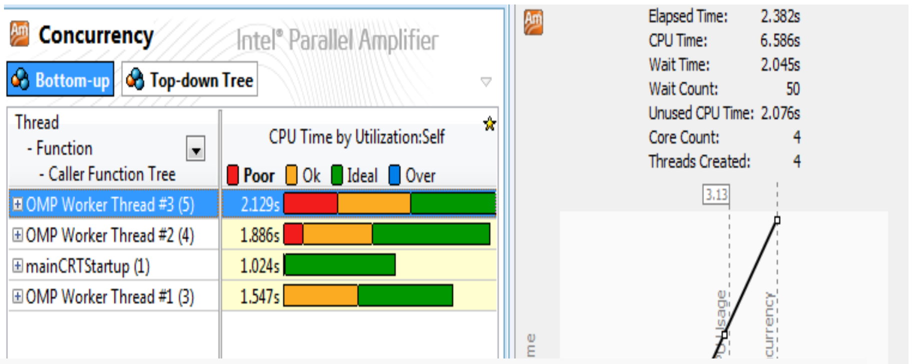
```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;

 pragma omp parallel for sched    (static,8)
    for( int i = start; i <= enc       = 2 )

             ( TestForPrime(i) )
             lobalPrimes[Interlock        gPrimesFound)] = i;
```



```
C:\classfiles\PrimeOpenMP\Debug>PrimeOpenMP.exe 1 5000000
 90%

  348513 primes found between       1 and 5000000 in     4.22 secs
```

Speedup achieved is 1.68X

Comparative Analysis



Thread - Function - Caller Function Tree	CPU Time by Utilization:Self
	🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over
OMP Worker Thread #3 (5)	2.129s
OMP Worker Thread #2 (4)	1.886s
mainCRTStartup (1)	1.024s
OMP Worker Thread #1 (3)	1.547s

Elapsed Time:	2.382s
CPU Time:	6.586s
Wait Time:	2.045s
Wait Count:	50
Unused CPU Time:	2.076s
Core Count:	4
Threads Created:	4

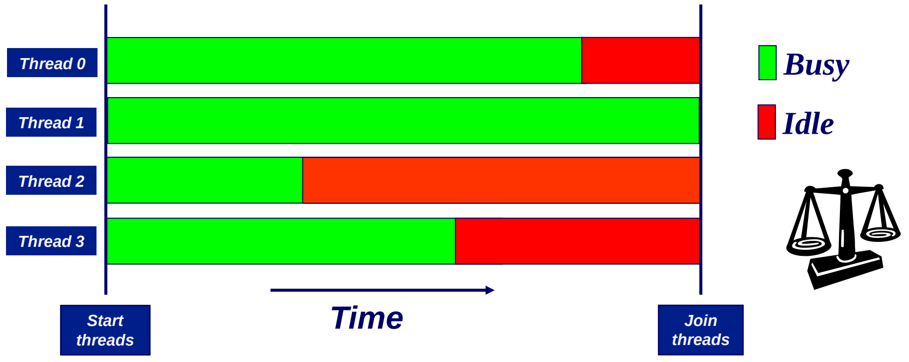
Threading applications require multiple iterations of going through the software development cycle

Common Performance Issues

- Load balance
  - Improper distribution of parallel work
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- Parallel Overhead
  - Due to thread creation, scheduling..
- Granularity
  - Not sufficient parallel work

Load Imbalance

- Unequal work loads lead to idle threads and wasted time



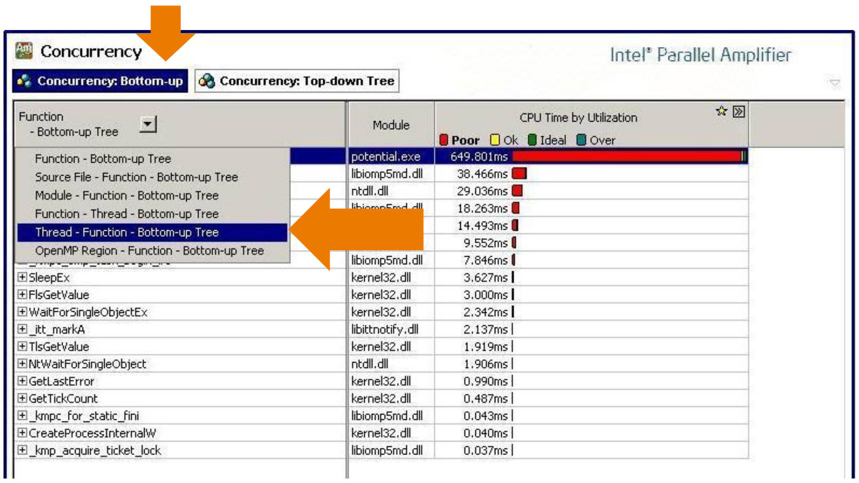
Redistribute Work to Threads

- Static assignment
  - –Are the same number of tasks assigned to each thread?
  - –Do tasks take different processing time?
    - Do tasks change in a predictable pattern?
      - ◆Rearrange (static) order of assignment to threads
    - Use dynamic assignment of tasks

Redistribute Work to Threads

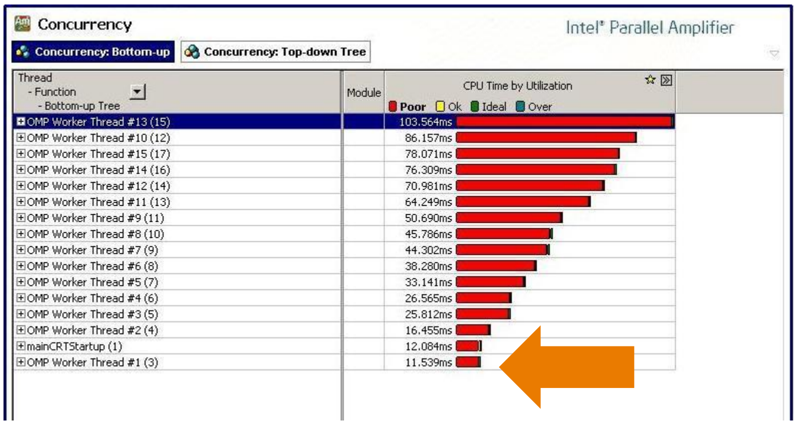
- Dynamic assignment
  - Is there one big task being assigned?
    - Break up large task to smaller parts
  - Are small computations agglomerated into larger task?
    - Adjust number of computations in a task
    - More small computations into single task?
    - Fewer small computations into single task?
    - Bin packing heuristics

Unbalanced Workloads


Concurrency
Intel® Parallel Amplifier
Concurrency: Bottom-up
Concurrency: Top-down Tree
Function - Bottom-up Tree
Source File - Function - Bottom-up Tree
Module - Function - Bottom-up Tree
Function - Thread - Bottom-up Tree
Thread - Function - Bottom-up Tree
OpenMP Region - Function - Bottom-up Tree

Function - Bottom-up Tree	Module	CPU Time by Utilization (🟥 Poor 🟨 Ok 🟩 Ideal 🟦 Over)
	potential.exe	649.801ms
	libiomp5md.dll	38.466ms
	ntdll.dll	29.036ms
	libiomp5md.dll	18.263ms
		14.493ms
		9.552ms
	libiomp5md.dll	7.846ms
SleepEx	kernel32.dll	3.627ms
FlsGetValue	kernel32.dll	3.000ms
WaitForSingleObjectEx	kernel32.dll	2.342ms
_itt_markA	libittnotify.dll	2.137ms
TlsGetValue	kernel32.dll	1.919ms
NtWaitForSingleObject	ntdll.dll	1.906ms
GetLastError	kernel32.dll	0.990ms
GetTickCount	kernel32.dll	0.487ms
_kmpc_for_static_fini	libiomp5md.dll	0.043ms
CreateProcessInternalW	kernel32.dll	0.040ms
_kmp_acquire_ticket_lock	libiomp5md.dll	0.037ms

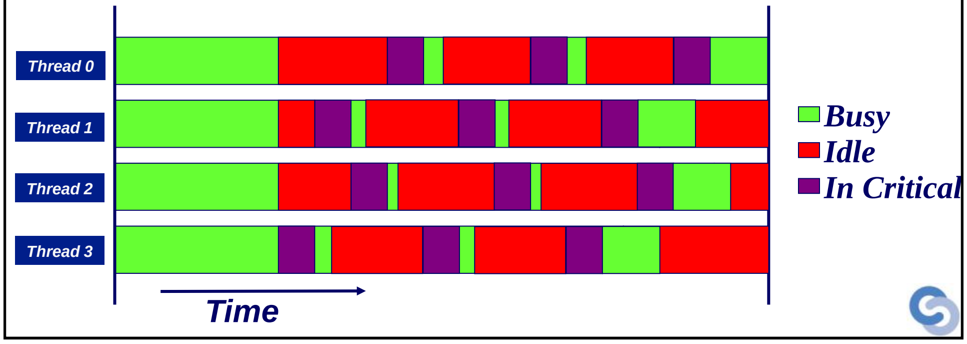
Unbalanced Workloads



Thread - Function - Bottom-up Tree	Module	CPU Time by Utilization (🟥 Poor 🟨 Ok 🟩 Ideal 🟦 Over)
OMP Worker Thread #13 (15)		103.564ms
OMP Worker Thread #10 (12)		86.157ms
OMP Worker Thread #15 (17)		78.071ms
OMP Worker Thread #14 (16)		76.309ms
OMP Worker Thread #12 (14)		70.981ms
OMP Worker Thread #11 (13)		64.249ms
OMP Worker Thread #9 (11)		50.690ms
OMP Worker Thread #8 (10)		45.786ms
OMP Worker Thread #7 (9)		44.302ms
OMP Worker Thread #6 (8)		38.280ms
OMP Worker Thread #5 (7)		33.141ms
OMP Worker Thread #4 (6)		26.565ms
OMP Worker Thread #3 (5)		25.812ms
OMP Worker Thread #2 (4)		16.455ms
mainCRTStartup (1)		12.084ms
OMP Worker Thread #1 (3)		11.539ms

Synchronization

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

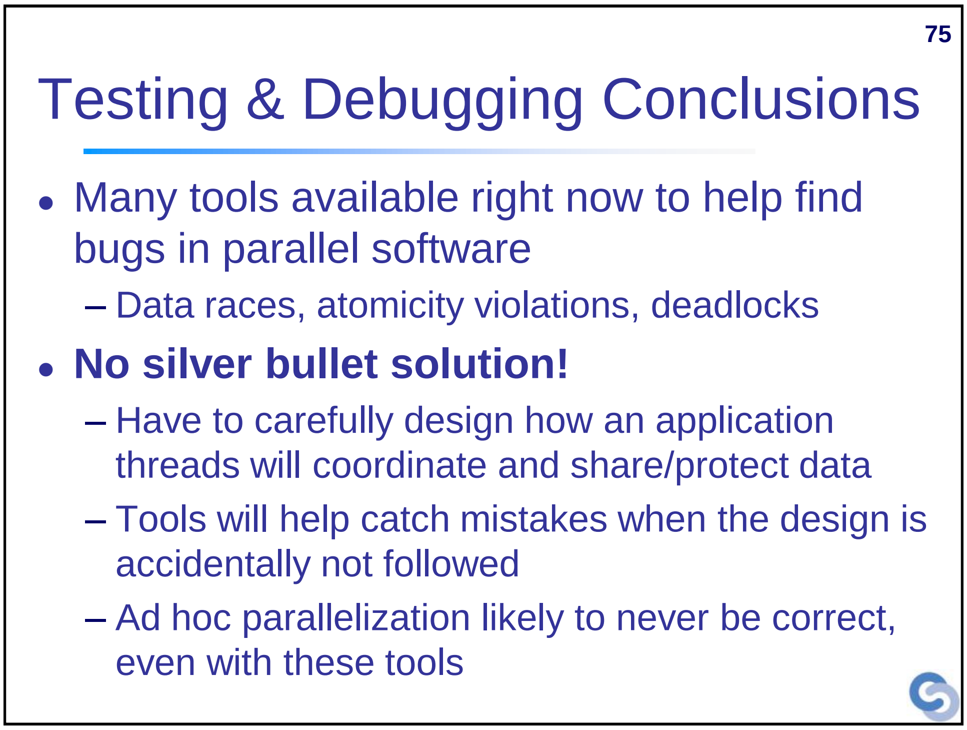


Synchronization Fixes

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

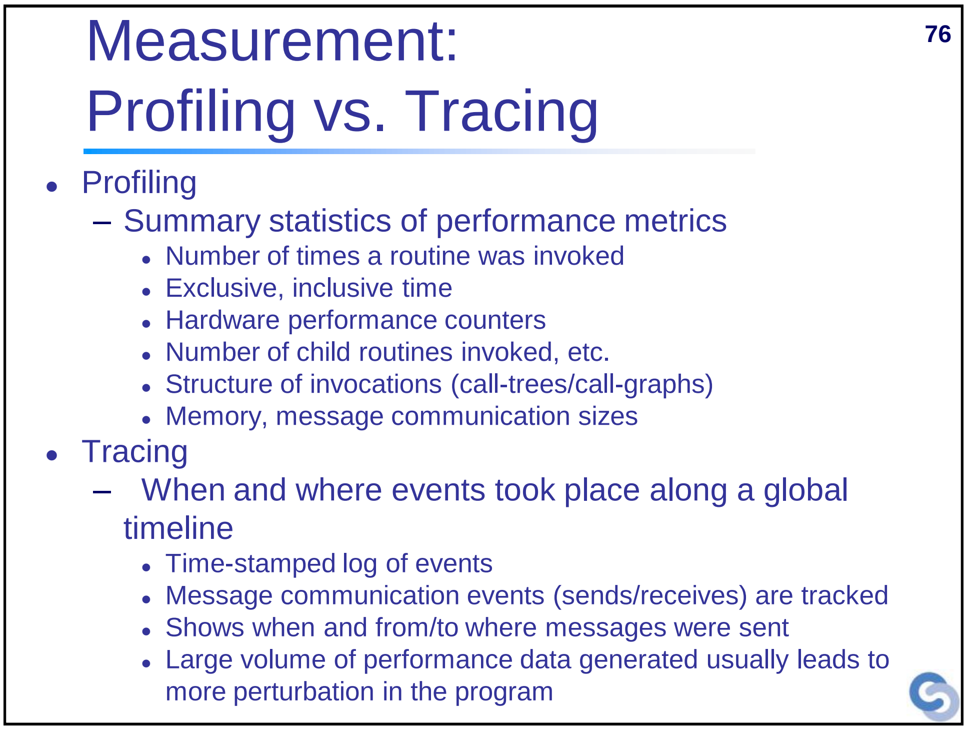
General Optimizations

- Serial Optimizations
  - Serial optimizations along the critical path should affect execution time
- Parallel Optimizations
  - Reduce synchronization object contention
  - Balance workload
  - Functional parallelism
- Analyze benefit of increasing number of processors
- Analyze the effect of increasing the number of threads on scaling performance



Testing & Debugging Conclusions

- Many tools available right now to help find bugs in parallel software
  - Data races, atomicity violations, deadlocks
- **No silver bullet solution!**
  - Have to carefully design how an application threads will coordinate and share/protect data
  - Tools will help catch mistakes when the design is accidentally not followed
  - Ad hoc parallelization likely to never be correct, even with these tools



Measurement: Profiling vs. Tracing

- Profiling
  - Summary statistics of performance metrics
    - Number of times a routine was invoked
    - Exclusive, inclusive time
    - Hardware performance counters
    - Number of child routines invoked, etc.
    - Structure of invocations (call-trees/call-graphs)
    - Memory, message communication sizes
- Tracing
  - When and where events took place along a global timeline
    - Time-stamped log of events
    - Message communication events (sends/receives) are tracked
    - Shows when and from/to where messages were sent
    - Large volume of performance data generated usually leads to more perturbation in the program

Measurement: Profiling

- Profiling
  - Helps to expose performance bottlenecks and hotspots
  - 80/20 –rule or *Pareto principle:* often 80% of the execution time in 20% of your application
  - Optimize what matters, don’t waste time optimizing things that have negligible overall influence on performance
- Implementation
  - **Sampling**: periodic OS interrupts or hardware counter traps
    - Build a histogram of sampled program counter (PC) values
    - Hotspots will show up as regions with many hits
  - **Measurement**: direct insertion of measurement code
    - Measure at start and end of regions of interests, compute difference

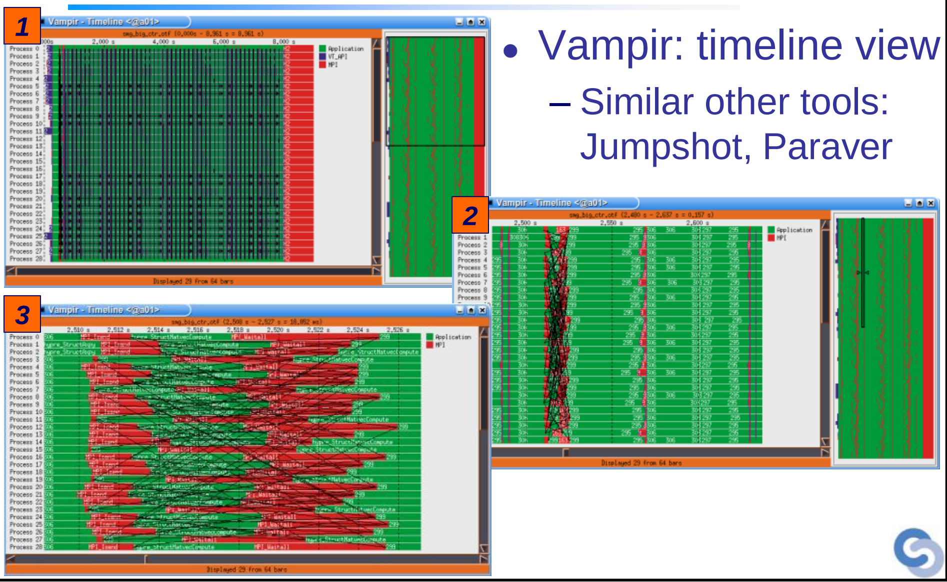
Measurement: Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
    - entering/exiting code region (function, loop, block, …)
    - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

Performance Data Analysis

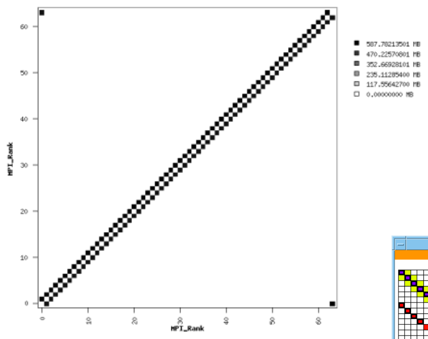
- Draw conclusions from measured performance data
- Manual analysis
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling
- Automated analysis
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca, Periscope

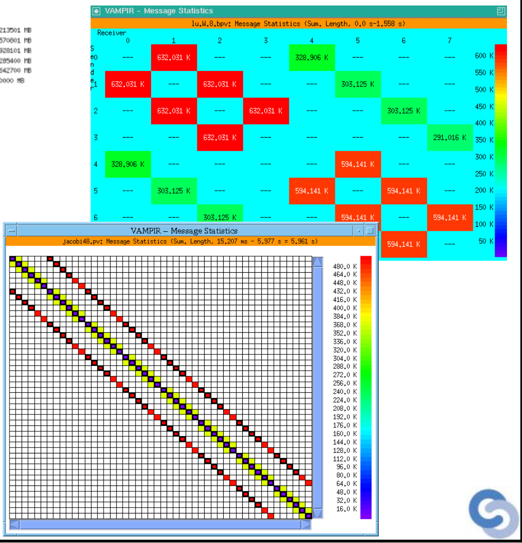
Trace File Visualization



- Vampir: timeline view
  - Similar other tools: Jumpshot, Paraver

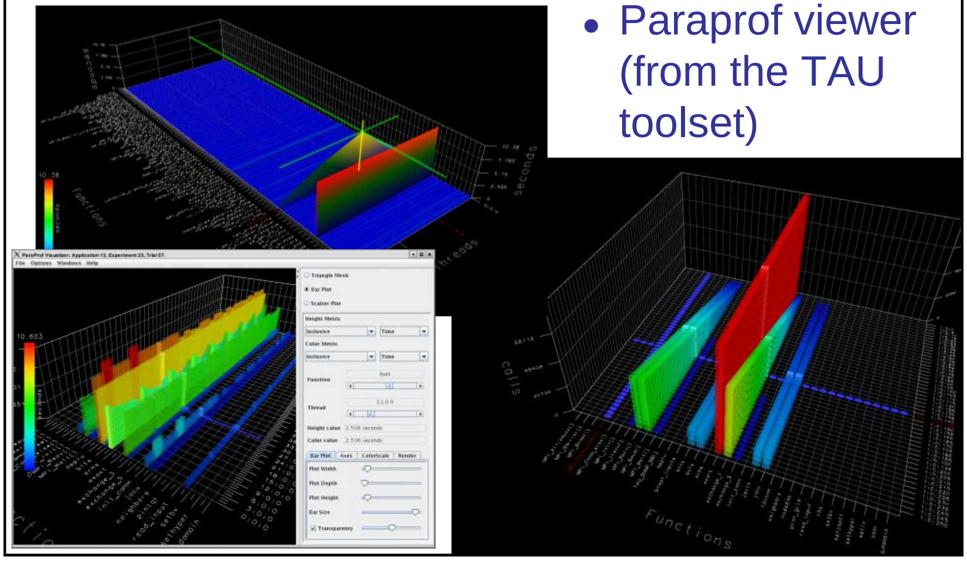
Trace File Visualization





- Vampir/IPM: message communication statistics

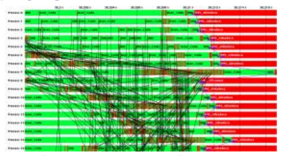
3D Performance Data Exploration



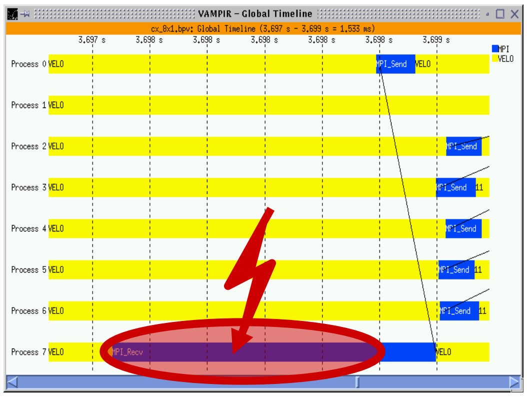
- Paraprof viewer (from the TAU toolset)

Automated Performance Analysis

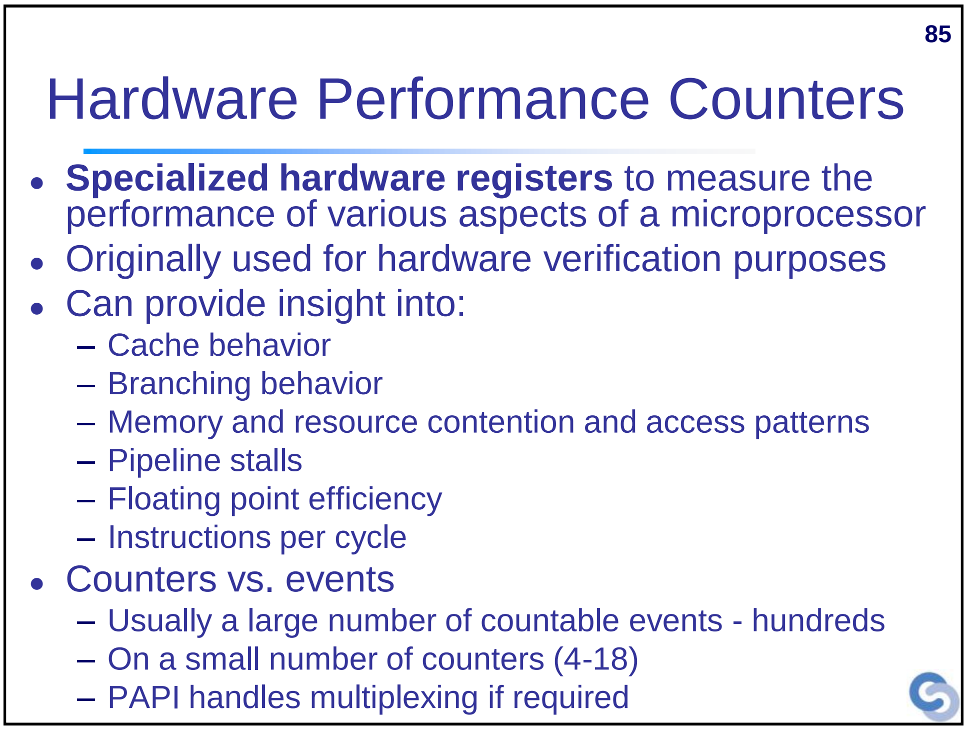
- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: 1.6 million cores
  - Trend to multi-core
- Large amounts of performance data when tracing
  - Several gigabytes or even terabytes
- Not all programmers are performance experts
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- Automation can solve some of these issues



Automation Example

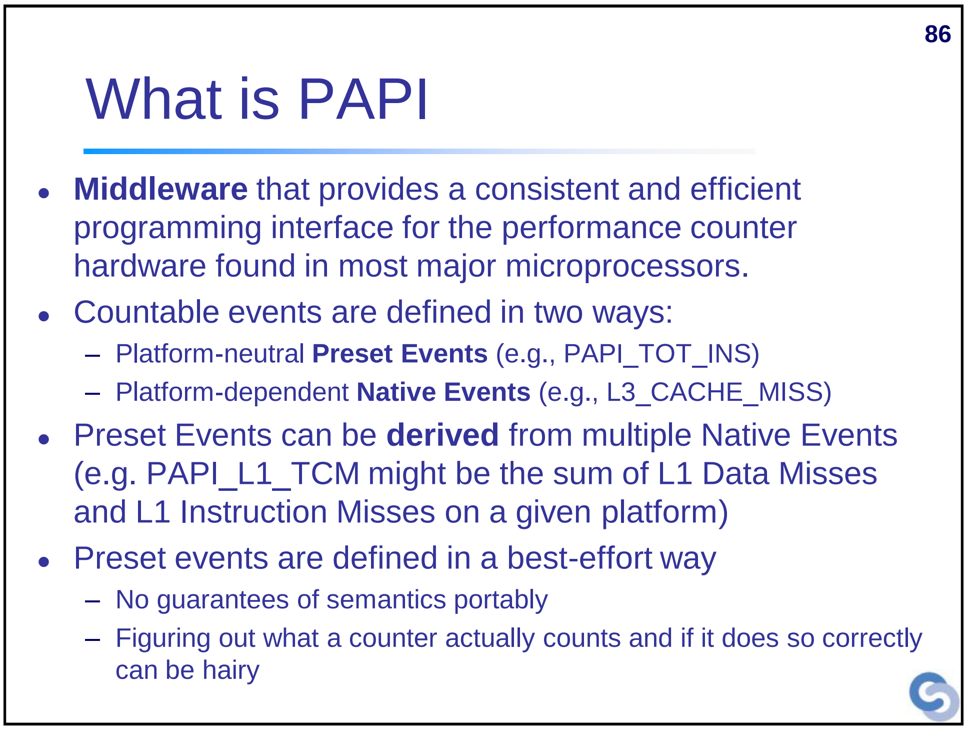


- „Late sender“ pattern
- This pattern can be detected automatically by analyzing the trace



Hardware Performance Counters

- **Specialized hardware registers** to measure the performance of various aspects of a microprocessor
- Originally used for hardware verification purposes
- Can provide insight into:
  - Cache behavior
  - Branching behavior
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
- Counters vs. events
  - Usually a large number of countable events - hundreds
  - On a small number of counters (4-18)
  - PAPI handles multiplexing if required



What is PAPI

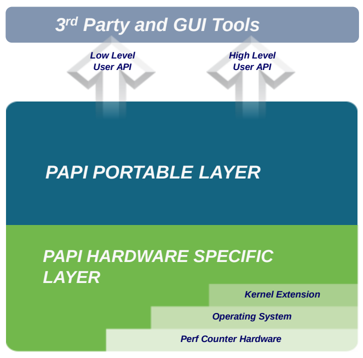
- **Middleware** that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.
- Countable events are defined in two ways:
  - Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
  - Platform-dependent **Native Events** (e.g., L3_CACHE_MISS)
- Preset Events can be **derived** from multiple Native Events (e.g. PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform)
- Preset events are defined in a best-effort way
  - No guarantees of semantics portably
  - Figuring out what a counter actually counts and if it does so correctly can be hairy

PAPI Hardware Events

- Preset Events
  - Standard set of over 100 events for application performance tuning
  - No standardization of the exact definitions
  - Mapped to either single or linear combinations of native events on each platform
  - Use **papi_avail** to see what preset events are available on a given platform
- Native Events
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use **papi_native_avail** utility to see all available native events
- Use **papi_event_chooser** utility to select a compatible set of events

PAPI Counter Interfaces

- PAPI provides 3 interfaces to the underlying counter hardware:
  - A **low level API** manages hardware events in user defined groups called EventSets. Meant for experienced application programmers wanting fine-grained measurements.
  - A **high level API** provides the ability to start, stop and read the counters for a specified list of events.
  - **Graphical** and end-user tools provide facile data collection and visualization.



PAPI High Level Calls

- PAPI_num_counters()
  - get the number of hardware counters available on the system
- PAPI_flips (**float** *rtime**, float** *ptime, **long long** *flpins, **float** *mflips)
  - simplified call to get Mflips/s (floating point instruction rate), real and processor time
- PAPI_flops (**float** *rtime, **float** *ptime, **long long** *flpops, **float** *mflops)
  - simplified call to get Mflops/s (floating point operation rate), real and processor time
- PAPI_ipc (**float** *rtime, **float** *ptime, **long long** *ins, **float** *ipc)
  - gets instructions per cycle, real and processor time
- PAPI_accum_counters (**long long** *values, **int** array_len)
  - add current counts to array and reset counters
- PAPI_read_counters (**long long** *values, **int** array_len)
  - copy current counts to array and reset counters
- PAPI_start_counters (**int** *events, **int** array_len)
  - start counting hardware events
- PAPI_stop_counters (**long long** *values, **int** array_len)
  - stop counters and return current counts

PAPI Example Low Level API Usage

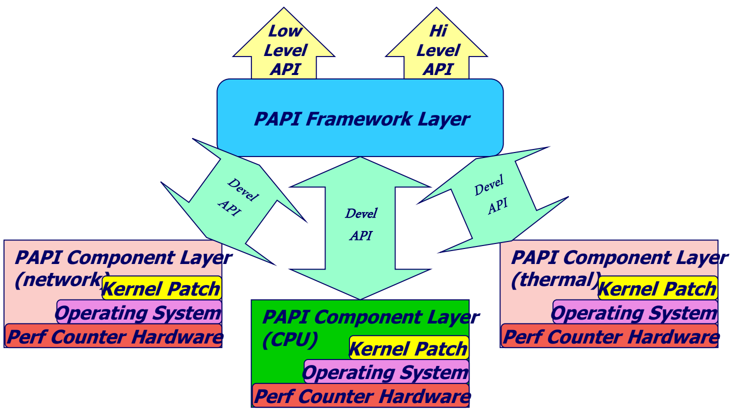
```
#include "papi.h”
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start (EventSet);

do_work();  /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);
```

Using PAPI through tools

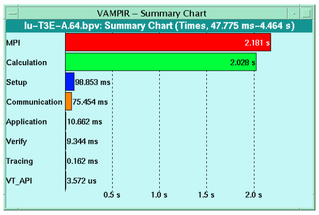
- You can use PAPI directly in your application, but most people use it through tools
- Tool might have a predfined set of counters or lets you select counters through a configuration file/environment variable, etc.
- Tools using PAPI
  - TAU (UO)
  - PerfSuite (NCSA)
  - HPCToolkit (Rice)
  - KOJAK, Scalasca (FZ Juelich, UTK)
  - Open|Speedshop (SGI)
  - ompP (UCB)
  - IPM (LBNL)

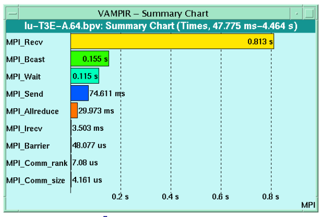
Component PAPI Design

Re-Implementation of PAPI w/ support for multiple monitoring domains



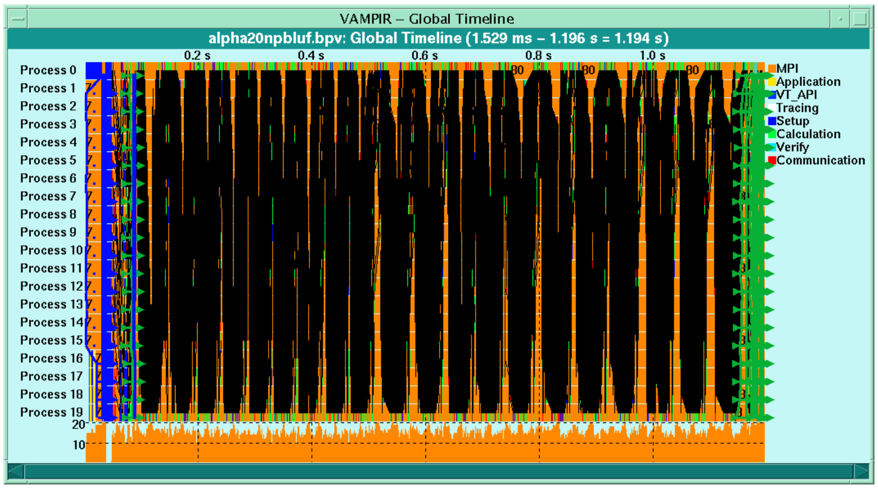
Vampir overview statistics





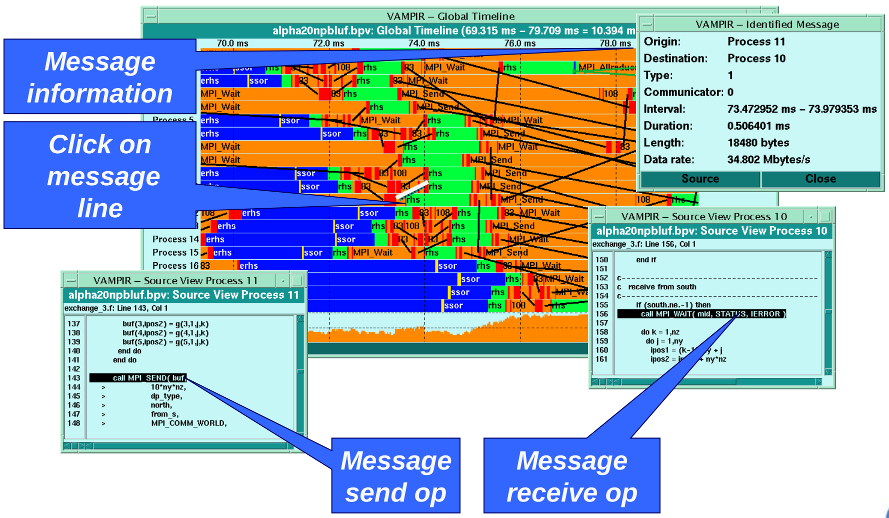
- Aggregated profiling information
  - Execution time
  - Number of calls
- This profiling information is computed from the trace
  - Change the selection in main timeline window
- Inclusive or exclusive of called routines

Timeline display



- To zoom, mark region with the mouse

Timeline display – message details



Origin: Process 11
Destination: Process 10
Type: 1
Communicator: 0
Interval: 73.472952 ms – 73.979353 ms
Duration: 0.506401 ms
Length: 18480 bytes
Data rate: 34.802 Mbytes/s

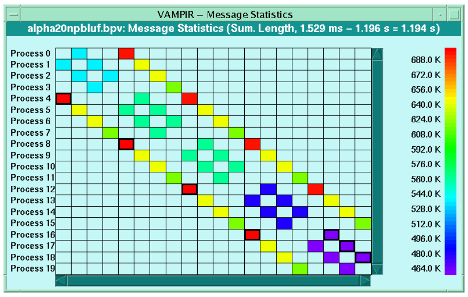
VAMPIR – Source View Process 11
alpha20npbluf.bpv: Source View Process 11
exchange_3.f: Line 143, Col 1

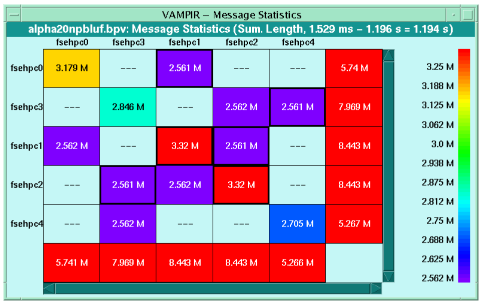
```
            buf(3,ipos2) = g(3,1,j,k)
            buf(4,ipos2) = g(4,1,j,k)
            buf(5,ipos2) = g(5,1,j,k)
          end do
        end do

        call MPI_SEND( buf,
     >                 10*ny*nz,
     >                 dp_type,
     >                 north,
     >                 from_s,
     >                 MPI_COMM_WORLD,
```

VAMPIR – Source View Process 10
alpha20npbluf.bpv: Source View Process 10
exchange_3.f: Line 156, Col 1

```
        end if

c-----------------------------------------
c   receive from south
c-----------------------------------------
        if (south.ne.-1) then
          call MPI_WAIT( mid, STATUS, IERROR )

          do k = 1,nz
            do j = 1,ny
              ipos1 = (k-1[illegible]y + j
              ipos2 = i[illegible] + ny*nz
```

Communication statistics

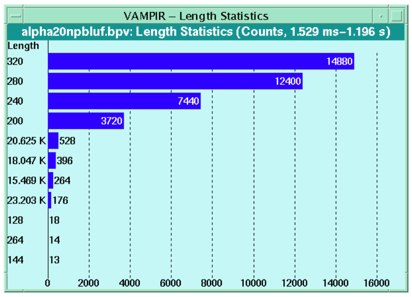


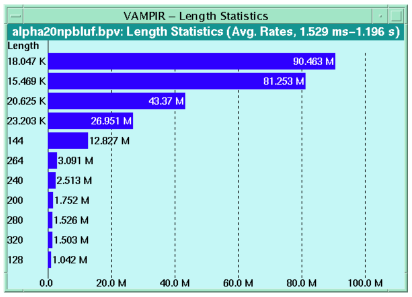

VAMPIR – Message Statistics
alpha20npbluf.bpv: Message Statistics (Sum. Length, 1.529 ms – 1.196 s = 1.194 s)

	fsehpc0	fsehpc3	fsehpc1	fsehpc2	fsehpc4	
fsehpc0	3.179 M	---	2.561 M	---	---	5.74 M
fsehpc3	---	2.846 M	---	2.562 M	2.561 M	7.969 M
fsehpc1	2.562 M	---	3.32 M	2.561 M	---	8.443 M
fsehpc2	---	2.561 M	2.562 M	3.32 M	---	8.443 M
fsehpc4	---	2.562 M	---	---	2.705 M	5.267 M
	5.741 M	7.969 M	8.443 M	8.443 M	5.266 M	

- Message statistics for each process/node pair:
  - Byte and message count
  - min/max/avg message length, bandwidth

Message histograms

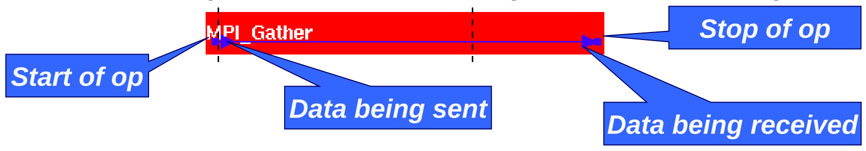


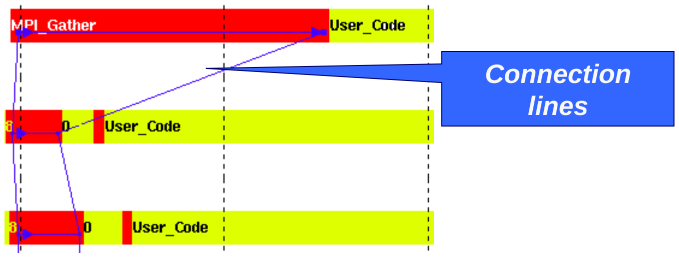


- Message statistics by length, tag or communicator
  - Byte and message count
  - Min/max/avg bandwidth

Collective operations

- For each process: mark operation locally

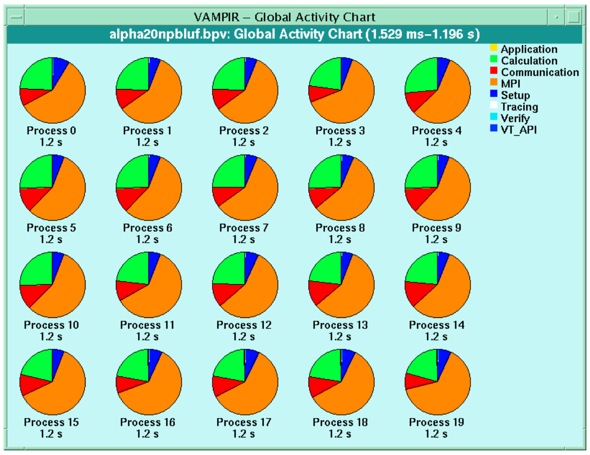




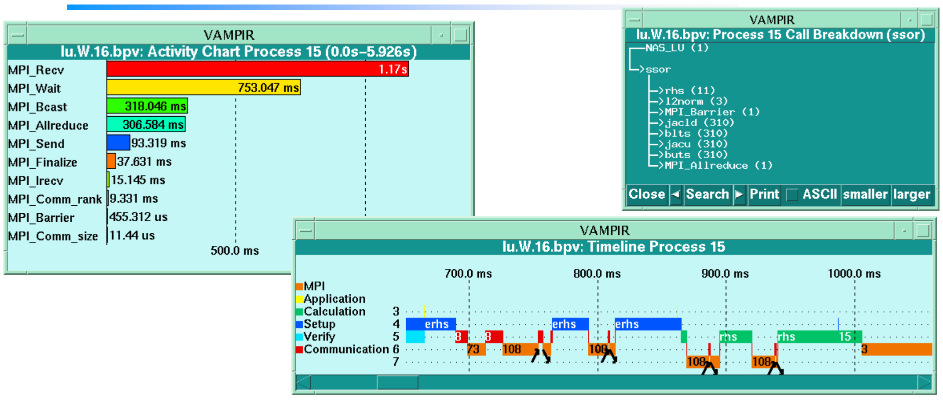
- Connect start/stop points by lines

Activity chart

- Profiling information for all processes



Process–local displays

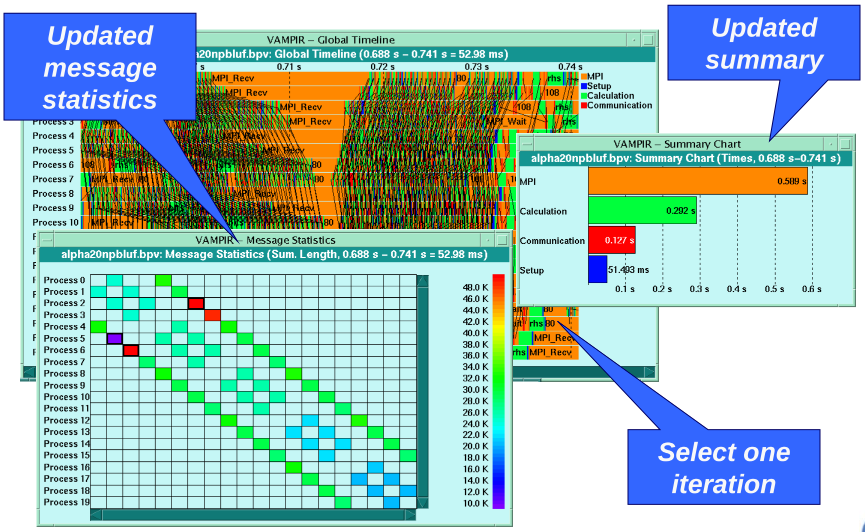


MPI_Recv	1.17s
MPI_Wait	753.047 ms
MPI_Bcast	318.046 ms
MPI_Allreduce	306.584 ms
MPI_Send	93.319 ms
MPI_Finalize	37.631 ms
MPI_Irecv	15.145 ms
MPI_Comm_rank	9.331 ms
MPI_Barrier	455.312 us
MPI_Comm_size	11.44 us

```
┌─NAS_LU (1)
│
└─>ssor
    │
    ├─>rhs (11)
    ├─>l2norm (3)
    ├─>MPI_Barrier (1)
    ├─>jacld (310)
    ├─>blts (310)
    ├─>jacu (310)
    ├─>buts (310)
    └─>MPI_Allreduce (1)
```

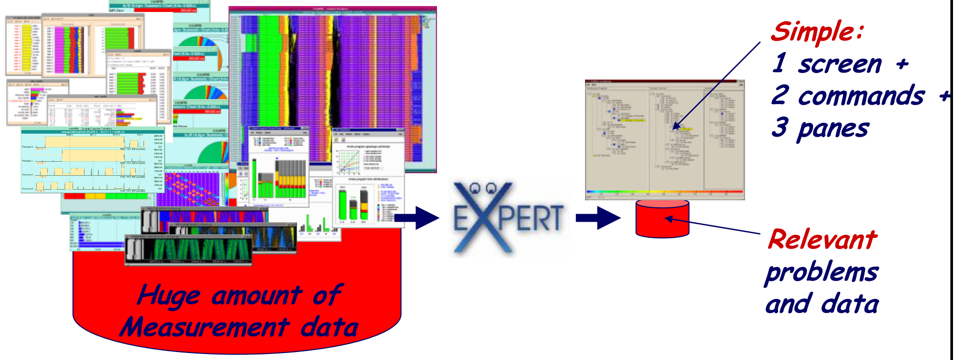
- Timeline (showing calling levels)
- Activity chart
- Calling tree (showing number of calls)

Effects of zooming


Updated message statistics
Updated summary
Select one iteration
VAMPIR – Global Timeline
VAMPIR – Summary Chart
alpha20npbluf.bpv: Summary Chart (Times, 0.688 s–0.741 s)
MPI 0.589 s
Calculation 0.292 s
Communication 0.127 s
Setup 51.493 ms
VAMPIR – Message Statistics
alpha20npbluf.bpv: Message Statistics (Sum. Length, 0.688 s – 0.741 s = 52.98 ms)

KOJAK / Scalasca – Basic Idea

- "Traditional" Tool
- Automatic Tool

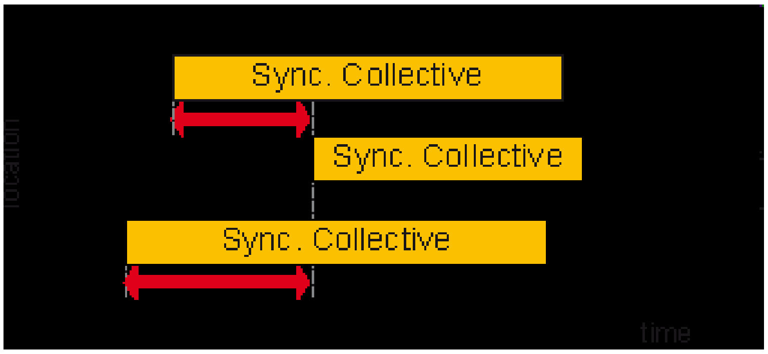

Huge amount of Measurement data
EXPERT
Simple: 1 screen + 2 commands + 3 panes
Relevant problems and data

- For non-standard / tricky cases (10%)
- For expert users

⇨ More productivity for performance analysis process!

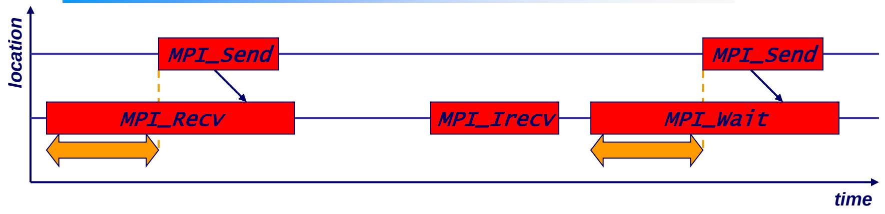
- For standard cases (90% ?!)
- For "normal" users
- Starting point for experts

MPI-1 Pattern: Wait at Barrier

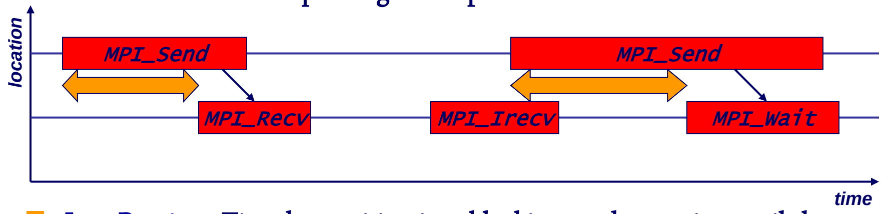


- Time spent in front of MPI synchronizing operation such as barriers

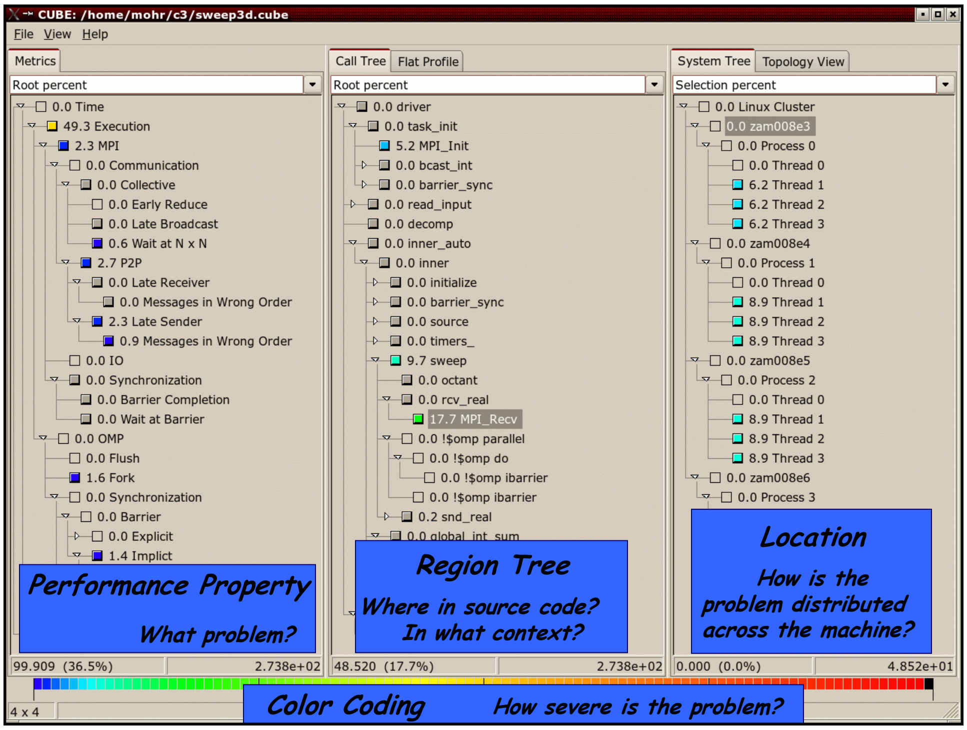
MPI-1 Pattern: Late Sender / Receiver



- Late Sender: Time lost waiting caused by a blocking receive operation posted earlier than the corresponding send operation



- Late Receiver: Time lost waiting in a blocking send operation until the corresponding receive operation is called



```
CUBE: /home/mohr/c3/sweep3d.cube
File  View  Help

Metrics                              Call Tree | Flat Profile           System Tree | Topology View
Root percent                         Root percent                       Selection percent
0.0 Time                             0.0 driver                         0.0 Linux Cluster
  49.3 Execution                       0.0 task_init                      0.0 zam008e3
    2.3 MPI                              5.2 MPI_Init                       0.0 Process 0
      0.0 Communication                  0.0 bcast_int                        0.0 Thread 0
        0.0 Collective                   0.0 barrier_sync                     6.2 Thread 1
          0.0 Early Reduce             0.0 read_input                         6.2 Thread 2
          0.0 Late Broadcast           0.0 decomp                             6.2 Thread 3
          0.6 Wait at N x N            0.0 inner_auto                     0.0 zam008e4
        2.7 P2P                          0.0 inner                          0.0 Process 1
          0.0 Late Receiver                0.0 initialize                     0.0 Thread 0
            0.0 Messages in Wrong Order    0.0 barrier_sync                   8.9 Thread 1
          2.3 Late Sender                  0.0 source                         8.9 Thread 2
            0.9 Messages in Wrong Order    0.0 timers_                        8.9 Thread 3
      0.0 IO                               9.7 sweep                      0.0 zam008e5
      0.0 Synchronization                    0.0 octant                     0.0 Process 2
        0.0 Barrier Completion               0.0 rcv_real                     0.0 Thread 0
        0.0 Wait at Barrier                    17.7 MPI_Recv                  8.9 Thread 1
    0.0 OMP                                  0.0 !$omp parallel               8.9 Thread 2
      0.0 Flush                                0.0 !$omp do                   8.9 Thread 3
      1.6 Fork                                   0.0 !$omp ibarrier       0.0 zam008e6
      0.0 Synchronization                      0.0 !$omp ibarrier           0.0 Process 3
        0.0 Barrier                          0.2 snd_real
          0.0 Explicit                       0.0 global_int_sum
          1.4 Implicit

99.909 (36.5%)      2.738e+02        48.520 (17.7%)     2.738e+02       0.000 (0.0%)      4.852e+01
4 x 4
```

Performance Property

What problem?

Region Tree

Where in source code? In what context?

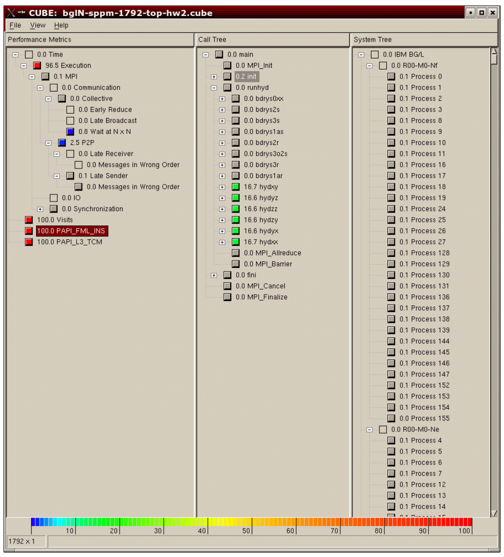
Location

How is the problem distributed across the machine?

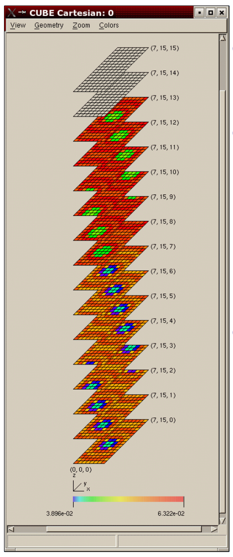
Color Coding *How severe is the problem?*

KOJAK: sPPM run on (8x16x14) 1792 PEs

108



```
CUBE: bglN-sppm-1792-top-hw2.cube
File  View  Help

Performance Metrics                  Call Tree                 System Tree
0.0 Time                             0.0 main                  0.0 IBM BG/L
  96.5 Execution                       0.0 MPI_Init              0.0 R00-M0-Nf
    0.1 MPI                            0.2 init                    0.1 Process 0
      0.0 Communication                0.0 runhyd                  0.1 Process 1
        0.0 Collective                   0.0 bdrys0xx              0.1 Process 2
          0.0 Early Reduce               0.0 bdrys2s               0.1 Process 3
          0.0 Late Broadcast             0.0 bdrys3s               0.1 Process 8
          0.8 Wait at N x N              0.0 bdrys1as              0.1 Process 9
        2.5 P2P                          0.0 bdrys2r               0.1 Process 10
          0.0 Late Receiver              0.0 bdrys3o2s             0.1 Process 11
            0.0 Messages in Wrong Order  0.0 bdrys3r               0.1 Process 16
          0.1 Late Sender                0.0 bdrys1ar              0.1 Process 17
            0.0 Messages in Wrong Order  16.7 hydxy                0.1 Process 18
      0.0 IO                             16.6 hydyz                0.1 Process 19
      0.0 Synchronization                16.6 hydzz                0.1 Process 24
  100.0 Visits                           16.6 hydzy                0.1 Process 25
  100.0 PAPI_FML_INS                     16.6 hydyx                0.1 Process 26
  100.0 PAPI_L3_TCM                      16.7 hydxx                0.1 Process 27
                                         0.0 MPI_Allreduce         0.1 Process 128
                                         0.0 MPI_Barrier           0.1 Process 129
                                       0.0 fini                    0.1 Process 130
                                       0.0 MPI_Cancel              0.1 Process 131
                                       0.0 MPI_Finalize            0.1 Process 136
                                                                   0.1 Process 137
                                                                   0.1 Process 138
                                                                   0.1 Process 139
                                                                   0.1 Process 144
                                                                   0.1 Process 145
                                                                   0.1 Process 146
                                                                   0.1 Process 147
                                                                   0.1 Process 152
                                                                   0.1 Process 153
                                                                   0.1 Process 154
                                                                   0.0 Process 155
                                                                 0.0 R00-M0-Ne
                                                                   0.1 Process 4
                                                                   0.1 Process 5
                                                                   0.1 Process 6
                                                                   0.1 Process 7
                                                                   0.1 Process 12
                                                                   0.1 Process 13
                                                                   0.1 Process 14

10  20  30  40  50  60  70  80  90  100
1792 x 1
```



- Topology display
- Shows distribution of pattern over HW topology
- Easily scales to even larger systems

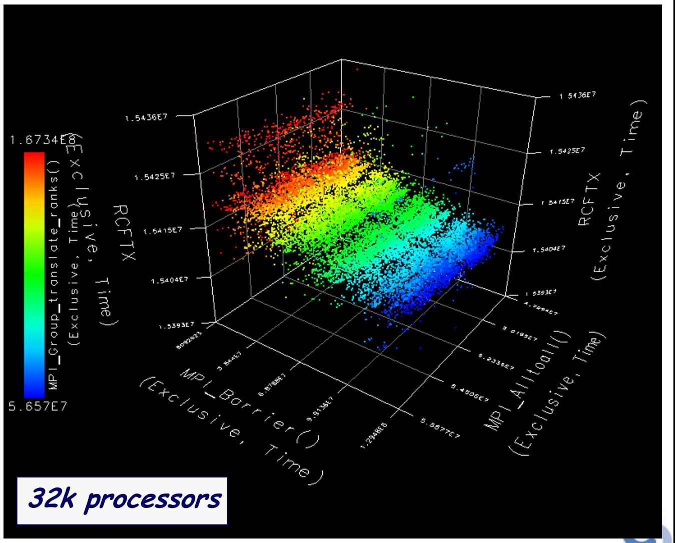
TAU Parallel Performance System

- http://www.cs.uoregon.edu/research/tau/
- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Integration in complex software, systems, applications

Tuning and Analysis Utilities

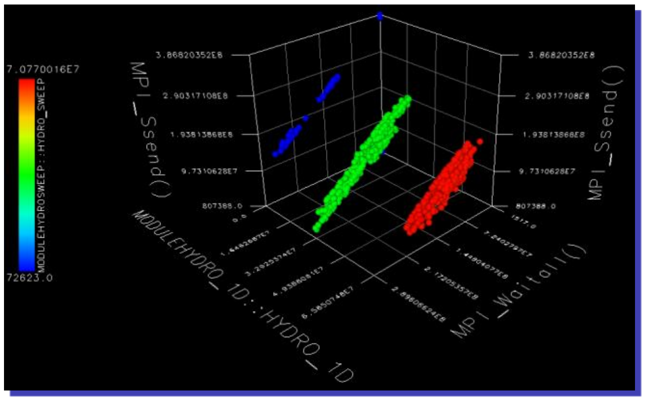
ParaProf – 3D Scatterplot (Miranda)

- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library



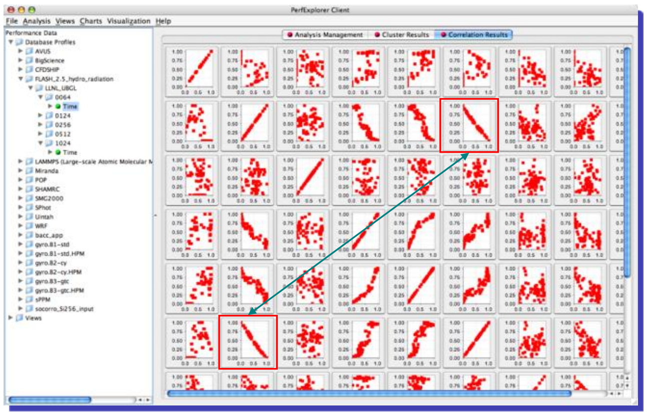
PerfExplorer - Cluster Analysis

- Four significant events automatically selected (from 16K processors)
- Clusters and correlations are visible



PerfExplorer – Correlation Analysis (Flash)

- Describes strength and direction of a linear relationship between two variables (events) in the data



Tools References

- IBM ConTest (Noise-Making for Java):
  - https://www.research.ibm.com/haifa/projects/verification/contest/index.html
- Cuzz (Random scheduling for C++/.NET):
  - http://research.microsoft.com/en-us/projects/cuzz/
- Intel Thread Checker and Parallel Inspector (C/C++):
  - http://software.intel.com/en-us/intel-thread-checker/
  - http://software.intel.com/en-us/intel-parallelinspector/
  - http://software.intel.com/en-us/articles/intel-parallel-studio-xe/
- Helgrind, DRD, ThreadSanitizer (Dynamic Data Race Detection/Prediction for C/C++):
  - http://valgrind.org/docs/manual/hg-manual.html
  - http://code.google.com/p/data-race-test/
- CHORD (Static Race/Deadlock Detection for Java):
  - http://code.google.com/p/jchord/

Tools References

- CalFuzzer (Java):
  - http://srl.cs.berkeley.edu/~ksen/calfuzzer/
- Thrille (C):
  - http://github.com/nicholasjalbert/Thrille
- CHESS (C++/.NET Model Checking, Race Detection):
  - http://research.microsoft.com/en-us/projects/chess/default.aspx
- Java Path Finder (Model Checking for Java):
  - http://babelfish.arc.nasa.gov/trac/jpf
- Tau Performance System (Fortran, C, C++, Java, Python):
  - http://www.cs.uoregon.edu/research/tau/home.php
- Vampir/GuideView (C/C++ and Fortran):
  - https://computing.llnl.gov/code/vgv.html
- Performance Application Programming Interface (PAPI):
  - http://icl.cs.utk.edu/papi/