Performance Engineering of Parallel Applications

Philip Blood
Pittsburgh Supercomputing Center
blood@psc.edu

International Summer School on HPC Challenges in Computational Sciences
Ljubljana, Slovenia
Acknowledgment

• Christian Feld, Jülich Supercomputing Centre
• Virtual Institute - High Productivity Supercomputing (VI-HPS)
• Raghu Reddy
Outline for Performance Sessions

• Thursday:
  – Introduction to performance engineering (Phil Blood)
  – Performance profiling of scientific application with Score-P (Christian Feld)

• Friday:
  – Analysis of performance profiles with TAU Paraprof (Phil Blood)
  – Trace measurement using Score-P (Christian Feld)
  – Trace analysis with Scalasca (Christian Feld)
Fitting algorithms to hardware…and vice versa

Molecular dynamics simulations on Application Specific Integrated Circuit (ASIC)

DE Shaw Research

Ivaylo Ivanov, Andrew McCammon, UCSD
Choice of **algorithm** most important consideration (serial and parallel)

Highly scalable codes must be designed to be scalable from the beginning!

Analysis may reveal need for new algorithm or completely different implementation rather than optimization

Focus of this lecture: using tools to assess parallel performance
Performance engineering workflow
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results

- Optimization
  - Modifications intended to eliminate/reduce performance problem

Slide courtesy VI-HPS
Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)

- Collection of performance data
- Aggregation of performance data

- Calculation of metrics
- Identification of performance problems
- Presentation of results

- Modifications intended to eliminate/reduce performance problem

Preparation

Measurement

Optimization

Analysis

Slide courtesy VI-HPS
Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)
- Collection of performance data
- Aggregation of performance data
- Modifications intended to eliminate/reduce performance problem
- Calculation of metrics
- Identification of performance problems
- Presentation of results

Preparation

Measurement

Optimization

Analysis

Slide courtesy VI-HPS
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Optimization
  - Modifications intended to eliminate/reduce performance problem

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results

- Optimization
  - Modifications intended to eliminate/reduce performance problem

Slide courtesy VI-HPS
Performance engineering workflow

Preparation:
- Prepare application with symbols
- Insert extra code (probes/hooks)

Measurement:
- Collection of performance data
- Aggregation of performance data

Optimization:
- Modifications intended to eliminate/reduce performance problem

Analysis:
- Calculation of metrics
- Identification of performance problems
- Presentation of results

Slide courtesy VI-HPS
Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)
- Collection of performance data
- Aggregation of performance data
- Modifying codes to eliminate/reduce performance problems
- Calculation of metrics
- Identification of performance problems
- Presentation of results
Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)
- Collection of performance data
- Aggregation of performance data
- Modifications intended to eliminate/reduce performance problem
- Calculation of metrics
- Identification of performance problems
- Presentation of results
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Optimization
  - Modifications intended to eliminate/reduce performance problem

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results

Slide courtesy VI-HPS
A little background...
Hardware Counters

- Counters: set of registers that count processor events, like floating point operations, or cycles
- Opteron “Istanbul” has 6 counter registers, so 6 types of events can be monitored simultaneously
- **PAPI: Performance API**
- Standard API for accessing hardware performance counters
- Enable mapping of code to underlying architecture
- Facilitates compiler optimizations and hand tuning
- Seeks to guide compiler improvements and architecture development to relieve common bottlenecks
Features of PAPI

• Portable: uses same routines to access counters across all architectures
• High-level interface
  – Using predefined standard events the same source code can access similar counters across various architectures without modification.
  – `papi_avail`
• Low-level interface
  – Provides access to all machine specific counters (requires source code modification)
  – Increased efficiency and flexibility
  – `papi_native_avail`
• Third-party tools
  – TAU, HPC Toolkit
• Might require linux kernel patch
  – **Direct support in linux kernels ≥ 2.6.31 (use latest PAPI)**
Measurement Techniques

• **When is measurement triggered?**
  – Sampling (indirect, external, low overhead)
    • interrupts, hardware counter overflow, …
  – Instrumentation (direct, internal, high overhead)
    • through code modification

• **How are data recorded?**
  – Profiling
    • summarizes performance data during execution
    • per process / thread and organized with respect to context
  – Tracing
    • trace record with performance data and timestamp
    • per process / thread
Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions

```c
int foo()
{
    int a;
    a = a + 1;
    bar();
    a = a + 1;
    return a;
}
```
Applying Performance Tools to Improve Parallel Performance of the UNRES MD code

The UNRES molecular dynamics (MD) code utilizes a carefully-derived mesoscopic protein force field to study and predict protein folding pathways by means of molecular dynamics simulations.

Structure of UNRES

- Two issues
  - Master/Worker code
    ```
    if (myrank==0)
       MD=>...=>EELEC
    else
       ERGASTULUM=>...=>EELEC
    endif
    ```
  - Significant startup time: must remove from profiling
    - Setup time: 300 sec
    - MD Time: 1 sec/step
    - Only MD time important for production runs of millions of steps
    - Could run for 30,000 steps to amortize startup!
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Is There a Performance Problem?

• What does it mean for a code to perform “poorly”?
Is There a Performance Problem?

- What does it mean for a code to perform “poorly”? Depends on the work being done.
Is There a Performance Problem?

• What does it mean for a code to perform “poorly”?
  – Depends on the work being done
    – Traditional measure: Percentage of peak performance
Is There a Performance Problem?

• What does it mean for a code to perform “poorly”?
  – Depends on the work being done
    – Traditional measure: Percentage of peak performance
  – What performance should I expect with my algorithm?
Is There a Performance Problem?

- What does it mean for a code to perform “poorly”?  
  - Depends on the work being done  
    - Traditional measure: Percentage of peak performance  
  - What performance should I expect with my algorithm?  
    - Roofline models: establish performance bounds for various numerical methods
Is There a Performance Problem?

• What does it mean for a code to perform “poorly”?
  – Depends on the work being done
    – Traditional measure: Percentage of peak performance
  – What performance should I expect with my algorithm?
    – Roofline models: establish performance bounds for various numerical methods
    – Arithmetic Intensity: Ratio of total floating-point operations (FLOPs) to total data movement (bytes)
Is There a Performance Problem?

- What does it mean for a code to perform “poorly”?
  - Depends on the work being done
    - Traditional measure: Percentage of peak performance
  - What performance should I expect with my algorithm?
    - Roofline models: establish performance bounds for various numerical methods
    - Arithmetic Intensity: Ratio of total floating-point operations (FLOPs) to total data movement (bytes)

Detecting Performance Problems

• Serial Performance: Fraction of Peak
  – 20% peak (overall) is usually decent; After that you decide how much effort it is worth
  – Theoretical FLOP/sec peak = FLOP/cycle * cycles/sec
  – 80:20 rule

• Parallel Performance: Scalability
  – Does run time decrease by 2x when I use 2x cores? (total work remains constant)
    • Strong scalability
  – Does run time remain the same when I keep the amount of work per core the same?
    • Weak scalability
Use a Sampling Tool for Initial Performance Check

- **HPC Toolkit**
  - Powerful sampling based tool
  - No recompilation necessary
  - Function level information available

- **PerfExpert**: TACC-developed automated performance analysis built on HPC Toolkit

- Worth checking out:
  - [http://hpctoolkit.org/](http://hpctoolkit.org/)
  - [http://www.tacc.utexas.edu/perfexpert](http://www.tacc.utexas.edu/perfexpert)
UNRES: Serial Performance

Processor and System Information
================================================================================================
Node CPUs : 768
Vendor : Intel
Family : Itanium 2
Clock (MHz) : 1669.001

Statistics
================================================================================================
Floating point operations per cycle.......................... 0.597
MFLOPS (cycles).................................................. 995.801
CPU time (seconds)................................................ 1404.675

• Theoretical peak on Itanium2: 4 FLOP/cycle *1669 MHz = 6676 MFLOPS
• UNRES getting 15% of peak--needs serial optimization on Itanium
• Much better on x86_64: 1720 MFLOPS, 33% peak
• Make sure compiler is inlining (-ipo needed for ifort, –Minline=reshape needed for pgf90)
UNRES: Parallel Performance

UNRES Performance: Cray XT3

- **Bigben**
- **Ideal**

- **X-axis**: Cores
- **Y-axis**: Timesteps/sec
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Which Functions are Important?

- Usually a handful of functions account for 90% of the execution time
- Make sure you are measuring the production part of your code
- For parallel apps, measure at high core counts – insignificant functions become significant!
### Contributions of Functions

#### Function Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>154346</td>
<td>76.99%</td>
<td>76.99%</td>
<td>pc_jac2d_blk3</td>
</tr>
<tr>
<td>14506</td>
<td>7.24%</td>
<td>84.23%</td>
<td>cg3_blk</td>
</tr>
<tr>
<td>10185</td>
<td>5.08%</td>
<td>89.31%</td>
<td>matxvec2d_blk3</td>
</tr>
<tr>
<td>6937</td>
<td>3.46%</td>
<td>92.77%</td>
<td>__kmp_x86_pause</td>
</tr>
<tr>
<td>4711</td>
<td>2.35%</td>
<td>95.12%</td>
<td>__kmp_wait_sleep</td>
</tr>
<tr>
<td>3042</td>
<td>1.52%</td>
<td>96.64%</td>
<td>dot_prod2d_blk3</td>
</tr>
<tr>
<td>2366</td>
<td>1.18%</td>
<td>97.82%</td>
<td>add_exchange2d_blk3</td>
</tr>
</tbody>
</table>

#### Function:File:Line Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function:File:Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>39063</td>
<td>19.49%</td>
<td>19.49%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:20</td>
</tr>
<tr>
<td>24134</td>
<td>12.04%</td>
<td>31.52%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:19</td>
</tr>
<tr>
<td>15626</td>
<td>7.79%</td>
<td>39.32%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:21</td>
</tr>
<tr>
<td>15028</td>
<td>7.50%</td>
<td>46.82%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:33</td>
</tr>
<tr>
<td>13878</td>
<td>6.92%</td>
<td>53.74%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:24</td>
</tr>
<tr>
<td>11880</td>
<td>5.93%</td>
<td>59.66%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:31</td>
</tr>
<tr>
<td>8896</td>
<td>4.44%</td>
<td>64.10%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:22</td>
</tr>
<tr>
<td>7863</td>
<td>3.92%</td>
<td>68.02%</td>
<td>matxvec2d_blk3:/home/rkufrin/apps/aspcg/matxvec2d_blk3.f:19</td>
</tr>
<tr>
<td>7145</td>
<td>3.56%</td>
<td>71.59%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:32</td>
</tr>
</tbody>
</table>
UNRES Function Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>2905589</td>
<td>51.98%</td>
<td>51.98%</td>
<td>eelecij</td>
</tr>
<tr>
<td>827023</td>
<td>14.79%</td>
<td>66.77%</td>
<td>egb</td>
</tr>
<tr>
<td>634107</td>
<td>11.34%</td>
<td>78.11%</td>
<td>setup_md_matrices</td>
</tr>
<tr>
<td>247353</td>
<td>4.42%</td>
<td>82.54%</td>
<td>escp</td>
</tr>
<tr>
<td>220089</td>
<td>3.94%</td>
<td>86.48%</td>
<td>etrbk3</td>
</tr>
<tr>
<td>183492</td>
<td>3.28%</td>
<td>89.76%</td>
<td>einvit</td>
</tr>
<tr>
<td>144851</td>
<td>2.59%</td>
<td>92.35%</td>
<td>banach</td>
</tr>
<tr>
<td>132058</td>
<td>2.36%</td>
<td>94.71%</td>
<td>ginv_mult</td>
</tr>
<tr>
<td>66182</td>
<td>1.18%</td>
<td>95.89%</td>
<td>multibody_hb</td>
</tr>
<tr>
<td>39495</td>
<td>0.71%</td>
<td>96.60%</td>
<td>etred3</td>
</tr>
<tr>
<td>38111</td>
<td>0.68%</td>
<td>97.28%</td>
<td>eelec</td>
</tr>
</tbody>
</table>

- Short runs include some startup functions amongst top functions
- To eliminate this perform a full production run with sampling tool
- Can use sampling tools during production runs due to low overhead—minimal impact on application performance
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks—use tracing to help identify cause and effect
Digging Deeper: Instrument Key Functions

• Instrumentation: Insert functions into source code to measure performance
• Pro: Gives precise information about where things happen
• Con: High overhead and perturbation of application performance
• Thus essential to only instrument important functions
Choose a tool: there are many!

- VI-HPS maintains a list and tool guide
- Will use TAU as an example in this presentation
- Focus on the general principles rather than specific details
- Christian Feld will take you through specific details using Score-P and Scalasca tools during hands-on session
TAU: Tuning and Analysis Utilities

• Useful for a more detailed analysis
  – Routine level
  – Loop level
  – Performance counters
  – Communication performance

• A more sophisticated tool
  – Performance analysis of Fortran, C, C++, Java, and Python
  – Portable: Tested on all major platforms
  – Steeper learning curve

http://www.cs.uoregon.edu/research/tau/home.php
General Instructions for TAU

- Use a TAU Makefile stub (even if you don’t use makefiles for your compilation)
- Use TAU scripts for compiling (tau_cc.sh tau_f90.sh)
- Example (most basic usage):

  ```
  module load tau
  
  setenv TAU_MAKEFILE <path>/Makefile.tau-papi-pdt-pgi
  setenv TAU_OPTIONS "-optVerbose -optKeepFiles"
  
  tau_f90.sh -o hello hello_mpi.f90
  ```

- Excellent “Cheat Sheet”!
  - Everything you need to know?! (Almost)
Using TAU with Makefiles

• Fairly simple to use with well written makefiles:

  setenv TAU_MAKEFILE <path>/Makefile.tau-papi-mpi-pdt-pgi
  setenv TAU_OPTIONS "-optVerbose –optKeepFiles –optPreProcess"
  make FC=tau_f90.sh

  – run code as normal
  – run pprof (text) or paraprof (GUI) to get results
  – paraprof --pack file.ppk (packs all of the profile files into
    one file, easy to copy back to local workstation)

• Example scenarios
  – Typically you can do cut and paste from here:
    http://www.cs.uoregon.edu/research/tau/docs/scenario/index.html
Tiny Routines: High Overhead

**Before:**

double precision function scalar(u,v)
double precision u(3),v(3)
   scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
return
eend

**After:**

double precision function scalar(u,v)
double precision u(3),v(3)
call TAU_PROFILE_TIMER(profiler, 'SCALAR [...]')
call TAU_PROFILE_START(profiler)
scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
call TAU_PROFILE_STOP(profiler)
return
call TAU_PROFILE_STOP(profiler)
eend
Reducing Overhead

ParaProf Profile Visualization Tool

Overhead (time in sec):
MD steps base: 51.4 seconds
MD steps with TAU: 315 seconds

Must reduce overhead to get meaningful results:

• In paraprof go to “File” and select “Create Selective Instrumentation File”

Click on one of these labels to reveal detailed function info
Selective Instrumentation File

TAU automatically generates a list of routines that you can save to a selective instrumentation file.
Selective Instrumentation File

- Automatically generated file essentially eliminates overhead in instrumented UNRES
- In addition to eliminating overhead, use this to specify:
  - Files to include/exclude
  - Routines to include/exclude
  - Directives for loop instrumentation
  - Phase definitions
- Specify the file in `TAU_OPTIONS` and recompile:
  ```bash
  setenv TAU_OPTIONS "-optVerbose -optKeepFiles
  -optPreProcess -optTauSelectFile=select .tau"
  ```
- [http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01.html](http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01.html)
Getting a Call Path with TAU

• Why do I need this?
  – To optimize a routine, you often need to know what is above and below it
  – e.g. Determine which routines make significant MPI calls
  – Helps with defining phases: stages of execution within the code that you are interested in

• To get callpath info, do the following at runtime:
  setenv TAU_CALLPATH 1 (this enables callpath)
  setenv TAU_CALLPATH_DEPTH 5 (defines depth)

• Higher depth introduces more overhead in TAU
Getting Call Path Information

Right click name of node and select “Show Thread Call Graph”
Isolate regions of code execution

- Eliminated overhead, now we need to deal with startup time:
  - Choose a region of the code of interest: e.g. the main computational kernel
  - Determine where in the code that region begins and ends (call path can be helpful)
  - Then put something like this in selective instrumentation file:
    ```
    static phase name="foo1_bar" file="foo.c" line=26 to line=27
    ```
  - Recompile and rerun
Key UNRES Functions in TAU (with Startup Time)

To get this view, left click on Mean, Max, Min, or Node labels on left hand side of main Paraprof window

<table>
<thead>
<tr>
<th>Metric: GET_TIME_OF_DAY</th>
<th>Value: Exclusive</th>
<th>Units: seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>364.929</td>
<td>21.167</td>
<td>SETUP_MD_MATRICES</td>
</tr>
<tr>
<td></td>
<td>11.638</td>
<td>BANAI</td>
</tr>
<tr>
<td></td>
<td>10.204</td>
<td>EINVIT</td>
</tr>
<tr>
<td></td>
<td>8.803</td>
<td>BANACH</td>
</tr>
<tr>
<td></td>
<td>5.97</td>
<td>ETEBIK3</td>
</tr>
<tr>
<td></td>
<td>2.154</td>
<td>ELEEC</td>
</tr>
<tr>
<td></td>
<td>1.017</td>
<td>FREDAG</td>
</tr>
<tr>
<td></td>
<td>1.193</td>
<td>EGB</td>
</tr>
<tr>
<td></td>
<td>0.953</td>
<td>ELAU</td>
</tr>
<tr>
<td></td>
<td>0.742</td>
<td>OINV_MULTI</td>
</tr>
<tr>
<td></td>
<td>0.659</td>
<td>ESCP</td>
</tr>
<tr>
<td></td>
<td>0.359</td>
<td>MPI_Barrier()</td>
</tr>
<tr>
<td></td>
<td>0.344</td>
<td>MPI_Waitall()</td>
</tr>
<tr>
<td></td>
<td>0.305</td>
<td>SUM_GRADIENT</td>
</tr>
<tr>
<td></td>
<td>0.223</td>
<td>MPI_Reduce()</td>
</tr>
<tr>
<td></td>
<td>0.208</td>
<td>INT_FROM_CART1</td>
</tr>
<tr>
<td></td>
<td>0.148</td>
<td>MULTIBODY_HB</td>
</tr>
<tr>
<td></td>
<td>0.142</td>
<td>MPI_Alreduce()</td>
</tr>
<tr>
<td></td>
<td>0.134</td>
<td>ZEROGRAD</td>
</tr>
<tr>
<td></td>
<td>0.127</td>
<td>SET_MATRICES</td>
</tr>
<tr>
<td></td>
<td>0.117</td>
<td>INTCARTDERIV</td>
</tr>
<tr>
<td></td>
<td>0.113</td>
<td>VEC_AND_DERIV</td>
</tr>
<tr>
<td></td>
<td>0.109</td>
<td>MPI_Bcast()</td>
</tr>
<tr>
<td></td>
<td>0.108</td>
<td>STATOUT</td>
</tr>
<tr>
<td></td>
<td>0.081</td>
<td>MPI_Sendrecv()</td>
</tr>
<tr>
<td></td>
<td>0.07</td>
<td>READFDB</td>
</tr>
<tr>
<td></td>
<td>0.057</td>
<td>OPENUNITS</td>
</tr>
<tr>
<td></td>
<td>0.055</td>
<td>INIT_INT_TABLE</td>
</tr>
<tr>
<td></td>
<td>0.055</td>
<td>ADD_HB_CONTACT</td>
</tr>
<tr>
<td></td>
<td>0.052</td>
<td>ETURN4</td>
</tr>
<tr>
<td></td>
<td>0.046</td>
<td>ETOR_D</td>
</tr>
<tr>
<td></td>
<td>0.046</td>
<td>EBEND</td>
</tr>
<tr>
<td></td>
<td>0.044</td>
<td>EOLRAT</td>
</tr>
<tr>
<td></td>
<td>0.044</td>
<td>INIT_TO_CART</td>
</tr>
</tbody>
</table>
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – **Measure code performance using hardware counters**
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Detecting Serial Performance Issues

- Identify hardware performance counters of interest
  - papi_avail
  - papi_native_avail
  - Run these commands on compute nodes!
- Run TAU (perhaps isolating regions of interest)
- Specify PAPI hardware counters at run time

```
setenv TAU_METRICS GET_TIME_OF_DAY:PAPI_FP_OPS:PAPI_TOT_CYC
```

- Be careful! Definition (and accuracy) of PAPI hardware counter presets can vary between architectures
Create a Derived Metric in Paraprof Manager
Perf of EELEC (peak is 2)

Go to: Paraprof manager
Options->”Show derived metrics panel”
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Do compiler optimization first!

EELEC – After forcing inlining with compiler
Further Info on Serial Optimization

• Tools help you find issues, areas of code to focus on – solving issues is application and hardware specific

• Good resource on techniques for serial optimization:
  – CI-Tutor course: “Performance Tuning for Clusters” http://ci-tutor.ncsa.illinois.edu/
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
TAU Recipe #1: Detecting Serial Bottlenecks

- To identify scaling bottlenecks, do the following for each run in a scaling study (e.g. 2-64 cores):
  1) In Paraprof manager right-click “Default Exp” and select “Add Trial”. Find packed profile file and add it.
  2) If you defined a phase, from main paraprof window select: Windows -> Function Legend-> Filter->Advanced Filtering
  3) Type in the name of the phase you defined, and click ‘Apply’
  4) Return to Paraprof manager, right-click the name of the trial, and select “Add to Mean Comparison Window”
- Compare functions across increasing core counts
Serial Bottleneck Detection in UNRES: Function Scaling (2-32 cores)

- Examine timings of functions in your region of interest as you scale up
- Identify functions that do not scale well or that need to be parallelized
- Find communication routines that are starting to dominate runtime
- **Caution:** Looking at mean execution time may not reveal some scaling problems (load imbalance)
Serial Bottleneck Detection in UNRES: Function Scaling (2-32 cores)

- Examine timings of functions in your region of interest as you scale up
- Identify functions that do not scale well or that need to be parallelized
- Find communication routines that are starting to dominate runtime
- **Caution:** Looking at mean execution time may not reveal some scaling problems (load imbalance)

Serial function begins to dominate runtime
TAU Recipe #2: Detecting Parallel Load Imbalance

- Examine timings of functions in your region of interest
  - If you defined a phase, from paraprof window, right-click on phase name and select: ‘Show profile for this phase’

- To look at load imbalance in a particular function:
  - Left-click on function name to look at timings across all processors

- To look at load imbalance across all functions:
  - In Paraprof window go to ‘Options’
  - Uncheck ‘Normalize’ and ‘Stack Bars Together’
Load Imbalance Detection in UNRES
Load Imbalance Detection in UNRES

Only looking at time spent in the important MD phase

Phase: PHASE_MD
Metric: TIME
Value: Exclusive
Load Imbalance Detection in UNRES

Only looking at time spent in the important MD phase

Observe multiple causes of load imbalance, as well as the serial bottleneck
Load Imbalance Detection in UNRES

Only looking at time spent in the important MD phase

Observe multiple causes of load imbalance, as well as the serial bottleneck
Load Imbalance Detection in UNRES

- In this case: Developers unaware that chosen algorithm would create load imbalance
- Reexamined available algorithms and found one with much better load balance – also fewer floating point operations!
- Also parallelized serial function causing bottleneck

Only looking at time spent in the important MD phase

Observe multiple causes of load imbalance, as well as the serial bottleneck
Major Serial Bottleneck and Load Imbalance in UNRES Eliminated
Major Serial Bottleneck and Load Imbalance in UNRES Eliminated

- Due to 4x faster serial algorithm the balance between computation and communication has shifted – communication must be more efficient to scale well
- Code then undergoes another round of profiling and optimization
Next Iteration of Performance Engineering with Optimized Code

Load imbalance on one processor causing other processors to idle in MPI_Barrier

May need to change how data is distributed, or even change underlying algorithm.

But beware investing too much effort for minimal gain!
Use Call Path Information: MPI Calls

Use call path information to find routines from which key MPI calls are made. Include these routines in tracing experiment.

To show source locations select: File -> Preferences
Use Call Path Information: MPI Calls

Use call path information to find routines from which key MPI calls are made. Include these routines in tracing experiment.

To show source locations select:
File -> Preferences
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Some Take-Home Points

• Good choice of (serial and parallel) algorithm is most important
• Performance measurement can help you determine if algorithm and implementation is good
• Do compiler and MPI parameter optimizations first
• Check/optimize serial performance before investing a lot of time in improving scaling
• Choose the right tool for the job
• Know when to stop: 80:20 rule
• XSEDE (and PRACE) staff collaborate with code developers to help with performance engineering of parallel codes (Extended Collaborative Support)
Questions?

blood@psc.edu
Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team
Christian Feld – Jülich Supercomputing Centre
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Optimization
  - Modifications intended to eliminate/reduce performance problem

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results
Fragmentation of tools landscape

- Several performance tools co-exist
  - Separate measurement systems and output formats
- Complementary features and overlapping functionality
- Redundant effort for development and maintenance
  - Limited or expensive interoperability
- Complications for user experience, support, training

Vampir
VampirTrace
OTF

Scalasca
EPILOG / CUBE

TAU
TAU native formats

Periscope
Online measurement
Scalasca ↔ TAU ↔ VAMPIR ↔ Paraver
Score-P project idea

- Start a community effort for a common infrastructure
  - Score-P instrumentation and measurement system
  - Common data formats OTF2 and CUBE4

- Developer perspective:
  - Save manpower by sharing development resources
  - Invest in new analysis functionality and scalability
  - Save efforts for maintenance, testing, porting, support, training

- User perspective:
  - Single learning curve
  - Single installation, fewer version updates
  - Interoperability and data exchange

- Project funded by BMBF
- Close collaboration PRIMA project funded by DOE
Partners

- Forschungszentrum Jülich, Germany
- German Research School for Simulation Sciences, Aachen, Germany
- Gesellschaft für numerische Simulation mbH Braunschweig, Germany
- RWTH Aachen, Germany
- Technische Universität Darmstadt, Germany
- Technische Universität Dresden, Germany
- Technische Universität München, Germany
- University of Oregon, Eugene, USA
Score-P functionality

- Provide typical functionality for HPC performance tools
- Support all fundamental concepts of partner’s tools

- Instrumentation (various methods)
- Sampling (experimental)
- Flexible measurement without re-compilation:
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data

- MPI/SHMEM, OpenMP/Pthreads, and hybrid parallelism (and serial)
- Enhanced functionality (CUDA, OpenCL, OpenACC, highly scalable I/O)
Design goals

▪ Functional requirements
  ▪ Generation of call-path profiles and event traces
  ▪ Using direct instrumentation and sampling
  ▪ Recording time, visits, communication data, hardware counters
  ▪ Access and reconfiguration also at runtime
  ▪ Support for MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC and their valid combinations

▪ Non-functional requirements
  ▪ Portability: all major HPC platforms
  ▪ Scalability: petascale
  ▪ Low measurement overhead
  ▪ Robustness
  ▪ Open Source: 3-clause BSD license
Score-P overview

Event traces (OTF2)

Call-path profiles (CUBE4, TAU)

Online interface

Hardware counter (PAPI, rusage, PERF, plugins)

Instrumentation wrapper

Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenCL, OpenACC)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)

Vampir
Scalasca
CUBE
TAU
TAUdb
Periscope
Future features and management

- Scalability to maximum available CPU core count
- Support for binary instrumentation
- Support for new programming models, e.g., PGAS
- Support for new architectures

- Ensure a single official release version at all times which will always work with the tools
- Allow experimental versions for new features or research

- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
Hands-on:
NPB-MZ-MPI / BT