Performance Engineering of Parallel Applications

Philip Blood
Pittsburgh Supercomputing Center
blood@psc.edu

European-US Summer School on HPC Challenges in Computational Sciences
Dublin, Ireland
Fitting algorithms to hardware…and vice versa

Molecular dynamics simulations on Application Specific Integrated Circuit (ASIC)

DE Shaw Research

Ivaylo Ivanov, Andrew McCammon, UCSD
Code Development and Optimization Process

- Choice of algorithm most important consideration (serial and parallel)
- Highly scalable codes must be designed to be scalable from the beginning!
- Measurement may reveal need for new algorithm or completely different implementation rather than optimization
- Focus of this lecture: using tools to assess parallel performance
A little background...
Hardware Counters

- Counters: set of registers that count processor events, like floating point operations, or cycles (Opteron has 4 registers, so 4 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, Perfsuite, IPM
• Generally requires linux kernel patch
  – Direct support in linux kernels $\geq 2.6.31$ (use latest PAPI)
Example: High-level interface

```c
#include <papi.h>
#define NUM_EVENTS 2
main()
{
    int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long_long values[NUM_EVENTS];
    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
    /* Do some computation here*/
    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
    /* Do some computation here */
    /* Stop counting events */
    if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
}```
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
Is There a Performance Problem?

• What does it mean for a code to perform “poorly”?
  – HPL on 4K cores can take a couple of hrs
  – Quantum calculations involving a few atoms may take a week
  – Depends on the work being done

• Where does performance need to be improved?
  – Serial performance problem?
  – Parallel performance problem?
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
IPM

- Very good tool to get an overall picture
  - Overall MFLOP
  - Communication/Computation ratio

- Pros
  - Quick and easy!
  - Minimal overhead (uses sampling rather than source code instrumentation)

- Cons
  - Harder to get at “nitty gritty” details
  - No OpenMP support

http://ipm-hpc.sourceforge.net/
IPM Mechanics

On Ranger:

1) module load ipm

2) just before the ibrun command in the batch script add:
   setenv LD_PRELOAD $TACC_IPM_LIB/libipm.so

3) run as normal

4) to generate webpage
   
   module load ipm (if not already)
   ipm_parse -html <xml_file>

You should be left with a directory with the html in. Tar it up, move to your local computer and open index.html with your browser.
IPM Overhead

• **Was run with 500 MD steps (time in sec)**
  – base: MD steps: 5.14637E+01
  – base-ipm: MD steps: 5.13576E+01

• **Overhead is negligible**
IPM Results: Overall Picture
IPM – Communication (overall)

Communication balance by task (sorted by MPI time)

Sorted index

Message Buffer Size Distributions: time

Done
PerfSuite

• Similar to IPM: great for getting overall picture of application performance

• Pros
  – Easy: no need to recompile
  – Minimal overhead
  – Provides function-level information

• Cons
  – Not available on all architectures: (Intel/AMD x86 and x86-64, Intel ia64, IBM POWER)

http://perfsuite.sourceforge.net/
PerfSuite Mechanics: Overall performance

% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data, # then use psprocess to produce an output file (default is plain text)

# First run: this will give you a summary of performance information over total program execution (e.g. MFLOPS)
% psrun myprog

% psprocess myprog.12345.xml > myprog.txt
First case provides hardware counter stats

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
<th>Counter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conditional branch instructions mispredicted.....</td>
<td>4831072449</td>
</tr>
<tr>
<td>4</td>
<td>Floating point instructions</td>
<td>86124489172</td>
</tr>
<tr>
<td>5</td>
<td>Total cycles</td>
<td>594547754568</td>
</tr>
<tr>
<td>6</td>
<td>Instructions completed</td>
<td>1049339828741</td>
</tr>
</tbody>
</table>

Statistics

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graduated instructions per cycle</td>
<td>1.765</td>
</tr>
<tr>
<td>Graduated floating point instructions per cycle</td>
<td>0.145</td>
</tr>
<tr>
<td>Level 3 cache miss ratio (data)</td>
<td>0.957</td>
</tr>
<tr>
<td>Bandwidth used to level 3 cache (MB/s)</td>
<td>385.087</td>
</tr>
<tr>
<td>% cycles with no instruction issue</td>
<td>10.410</td>
</tr>
<tr>
<td>% cycles stalled on memory access</td>
<td>43.139</td>
</tr>
<tr>
<td>MFLOPS (cycles)</td>
<td>115.905</td>
</tr>
<tr>
<td>MFLOPS (wallclock)</td>
<td>114.441</td>
</tr>
</tbody>
</table>
UNRES: Serial Performance

Processor and System Information (abbreviated output from PerfSuite)
=================================================================
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 Bigben (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

Cores

<table>
<thead>
<tr>
<th>Cores</th>
<th>timesteps/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

- **Bigben**
- **Ideal**
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!
PerfSuite Mechanics: Function breakdown

% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an output file (default is plain text)

# This will break down cycles spent in each function

% psrun -C -c papi_profile_cycles.xml myprog

% psprocess -e myprog myprog.67890.xml > myprog_functions.txt
# 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 with PerfSuite

### 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 PerfSuite
- Can use PerfSuite and IPM during production runs due to low overhead—minimal impact on application performance
HPC Toolkit

- Powerful sampling based tool
- No recompilation necessary
- Function level information available
- Worth checking out:

http://hpctoolkit.org/
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
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
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)
    http://www.psc.edu/index.php/tau/430
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:**
```fortran
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
end
```

**After:**
```fortran
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)
end
```
Reducing Overhead

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”
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:
  
  ```
  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
Getting Call Path Information

Right click name of node and select “Show Thread Call Graph”
Phase Profiling: 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)

Metric: GET_TIME_OF_DAY  
Value: Exclusive  
Units: seconds

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP_MD_MATRICES</td>
<td>364.929</td>
</tr>
<tr>
<td>BANAI</td>
<td>21.167</td>
</tr>
<tr>
<td>EINVIT</td>
<td>11.686</td>
</tr>
<tr>
<td>BANACH</td>
<td>10.284</td>
</tr>
<tr>
<td>TEREK-3</td>
<td>8.853</td>
</tr>
<tr>
<td>FREDA</td>
<td>5.97</td>
</tr>
<tr>
<td>ELECC</td>
<td>2.154</td>
</tr>
<tr>
<td>EGB</td>
<td>1.017</td>
</tr>
<tr>
<td>ELAU</td>
<td>1.193</td>
</tr>
<tr>
<td>OINV_MULT</td>
<td>0.953</td>
</tr>
<tr>
<td>ESCP</td>
<td>0.742</td>
</tr>
<tr>
<td>MPI_Barrier()</td>
<td>0.659</td>
</tr>
<tr>
<td>MPI_Waits()</td>
<td>0.359</td>
</tr>
<tr>
<td>SUM_GRADIENT</td>
<td>0.344</td>
</tr>
<tr>
<td>MPI_Reduce()</td>
<td>0.305</td>
</tr>
<tr>
<td>INT_FROM_CART1</td>
<td>0.223</td>
</tr>
<tr>
<td>MULTIBODY_HB</td>
<td>0.208</td>
</tr>
<tr>
<td>MPI_Allreduce()</td>
<td>0.148</td>
</tr>
<tr>
<td>ZEROGRAD</td>
<td>0.142</td>
</tr>
<tr>
<td>SET_MATRICES</td>
<td>0.134</td>
</tr>
<tr>
<td>INTCARTDERIV</td>
<td>0.127</td>
</tr>
<tr>
<td>ADD_INT_FROM</td>
<td>0.117</td>
</tr>
<tr>
<td>VEC_AND_DERIV</td>
<td>0.113</td>
</tr>
<tr>
<td>MPI_Bcast()</td>
<td>0.109</td>
</tr>
<tr>
<td>STATOUT</td>
<td>0.108</td>
</tr>
<tr>
<td>MPI_Scatterv()</td>
<td>0.095</td>
</tr>
<tr>
<td>READFDE</td>
<td>0.087</td>
</tr>
<tr>
<td>OPENUNITS</td>
<td>0.057</td>
</tr>
<tr>
<td>INIT_INT_TABLE</td>
<td>0.055</td>
</tr>
<tr>
<td>ADD_HE_CONTACT</td>
<td>0.055</td>
</tr>
<tr>
<td>ETURN4</td>
<td>0.062</td>
</tr>
<tr>
<td>ETOR_D</td>
<td>0.046</td>
</tr>
<tr>
<td>EBEND</td>
<td>0.048</td>
</tr>
<tr>
<td>EQLPAT</td>
<td>0.044</td>
</tr>
<tr>
<td>INT_TO_CART</td>
<td>0.04</td>
</tr>
</tbody>
</table>
### Key UNRES Functions (MD Time Only)

#### Phase: PHASE_MD
**Metric:** TIME
**Value:** Exclusive
**Units:** seconds

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEEC</td>
<td>0.109</td>
</tr>
<tr>
<td>EGE</td>
<td>1.899</td>
</tr>
<tr>
<td>GNY_MULT</td>
<td>1.062</td>
</tr>
<tr>
<td>ESCP</td>
<td>0.739</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>0.519</td>
</tr>
<tr>
<td>SUM_GRADIENT</td>
<td>0.36</td>
</tr>
<tr>
<td>MULTIBODY_HB</td>
<td>0.261</td>
</tr>
<tr>
<td>INT_FROM_CART</td>
<td>0.225</td>
</tr>
<tr>
<td>ZEROGRAD</td>
<td>0.169</td>
</tr>
<tr>
<td>SET_MATRICES</td>
<td>0.152</td>
</tr>
<tr>
<td>MPI_Union Allreduce()</td>
<td>0.148</td>
</tr>
<tr>
<td>MPI_Waitall()</td>
<td>0.135</td>
</tr>
<tr>
<td>VEC_AND_CERIV</td>
<td>0.133</td>
</tr>
<tr>
<td>MPI_Bcast()</td>
<td>0.084</td>
</tr>
<tr>
<td>ETOR_D</td>
<td>0.054</td>
</tr>
<tr>
<td>EBEND</td>
<td>0.05</td>
</tr>
<tr>
<td>ETURN4</td>
<td>0.044</td>
</tr>
<tr>
<td>MPI_ScatterV()</td>
<td>0.038</td>
</tr>
<tr>
<td>ADD_HB_CONTACT</td>
<td>0.035</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>0.027</td>
</tr>
<tr>
<td>ETURN3</td>
<td>0.026</td>
</tr>
<tr>
<td>ESC</td>
<td>0.022</td>
</tr>
<tr>
<td>CHAINBUILD_CART</td>
<td>0.018</td>
</tr>
<tr>
<td>MPI_Irecv()</td>
<td>0.017</td>
</tr>
<tr>
<td>PHASE_MD</td>
<td>0.014</td>
</tr>
<tr>
<td>ETOR</td>
<td>0.001</td>
</tr>
<tr>
<td>ETOTAL</td>
<td>0.008</td>
</tr>
<tr>
<td>INTCARTDERIV</td>
<td>0.008</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! Login nodes will give you an error.

• Run TAU (perhaps with phases defined to isolate regions of interest)

• Specify PAPI hardware counters at run time:
  setenv TAU_METRICS GET_TIME_OF_DAY:PAPI_FP_OPS:PAPI_TOT_CYC
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
Detecting Parallel Performance Issues: 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)
Detecting Parallel Performance Issues: 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

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
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 is undergoing another round of profiling and optimization

Phase: PHASE_MD
Metric: TIME
Value: Exclusive
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!
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
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
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)
Acknowledgment

• POINT team: http://www.nic.uoregon.edu/point
• Raghu Reddy
Questions?

blood@psc.edu