Par Lab Parallel Boot Camp

Performance Tools

Karl Fuerlinger

fuerling@eecs.berkeley.edu
Outline

• Motivation

• Concepts and Definitions
  – Instrumentation, monitoring, analysis

• Some tools and their functionality
  – PAPI – access to hardware performance counters
  – ompP – profiling OpenMP code
  – IPM – monitoring message passing applications
Motivation

• Performance analysis is important
  – For HPC: computer systems are large investments
    » Procurement: O($40 Mio)
    » Operational costs: ~$5 Mio per year
    » Power: 1 MWyear ~$1 Mio
  
  – Goals:
    » Solve larger problems (new science)
    » Solve problems faster (turn-around time)
    » Improve error bounds on solutions (confidence)
• The typical performance optimization cycle

- Code Development
- Measure
- Analyze
- Modify / Tune
- Usage / Production

Instrumentation

Functionally complete and correct program

Complete, correct and well-performing program
Instrumentation

- **Instrumentation**: adding measurement probes to the code in order to observe its execution.
  - Can be done on several levels and different techniques for different levels.
  - Different overheads and levels of accuracy with each technique.
  - No application instrumentation needed: run in a simulator. E.g., Valgrind, SIMICS, etc. but speed is an issue.
• **Library Instrumentation:**

![Diagram of library instrumentation]

- **MPI library interposition**
  - All functions are available under two names: `MPI_Xxx` and `PMPI_Xxx`,
  - `MPI_Xxx` symbols are **weak**, can be over-written by interposition library
  - Measurement code in the interposition library measures begin, end, transmitted data, etc... and calls corresponding PMPI routine.
  - Not all MPI functions need to be instrumented
• Preprocessor Instrumentation
  – Example: Instrumenting OpenMP constructs with Opari
  – Preprocessor operation

```
POMP_Parallel_fork [master]
#pragma omp parallel {
/* user code in parallel region */
/* user code in parallel region */
POMP_Barrier_enter [team]
#pragma omp barrier
POMP_Barrier_exit [team]
POMP_Parallel_end [team]
}
POMP_Parallel_join [master]
```

This approach is used for OpenMP instrumentation by most vendor-independent tools. Examples: TAU/Kojak/Scalasca/ompP
• Source code instrumentation
  – **User-added** time measurement, etc. (e.g., `printf()`, `gettimeofday()`)
  – **Think twice** before you roll your own solution, many **tools** expose mechanisms for source code instrumentation in addition to automatic instrumentation facilities they offer
  – Instrument program phases:
    » Initialization
    » main loop iteration 1,2,3,4,...
    » data post-processing

  – Pragma and pre-processor based

    ```c
    #pragma pomp inst begin(foo)
    // application code
    #pragma pomp inst end(foo)
    ```

  – Macro / function call based

    ```c
    ELG_USER_START("name");
    // application code
    ELG_USER_END("name");
    ```
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
• 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
Profiling: Inclusive vs. Exclusive Time

```c
int main()
{
    /* takes 100 secs */
    f1(); /* takes 20 secs */
    /* other work */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */
    /* other work */
}
```

- **Inclusive time for** `main`
  - 100 secs

- **Exclusive time for** `main`
  - 100-20-50-20=10 secs

- **Exclusive time** sometimes called “self” time

- Similar definitions for inclusive/exclusive time for `f1()` and `f2()`

- Similar for other metrics, such as hardware performance counters, etc
Tracing Example: Instrumentation, Monitor, Trace

Process A:
```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

Process B:
```c
void slave {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definitions:
1. master
2. slave
3. ...

Event table:
- Time: 58 A ENTER 1
- Time: 60 B ENTER 2
- Time: 62 A SEND B
- Time: 64 A EXIT 1
- Time: 68 B RECV A
- Time: 69 B EXIT 2
- ...

Timestamp, time, location, event, context
### Tracing: Timeline Visualization

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER</td>
<td>2</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND</td>
<td>B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV</td>
<td>A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- **Main** (blue) for master processes
- **Red** for A
- **Blue** for B

Timeline:
- **58** to **70**

Timeline Table:

```
master
slave
...
58 A ENTER 1
60 B ENTER 2
62 A SEND B
64 A EXIT 1
68 B RECV A
69 B EXIT 2
...
```
• 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
• 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
• 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**
• „Late sender“ pattern

• This pattern can be detected automatically by analyzing the trace
• Motivation

• Concepts and Definitions
  – Instrumentation, monitoring, analysis

• Some tools and their functionality
  – PAPI – access to hardware performance counters
  – ompP – profiling OpenMP code
  – IPM – monitoring message passing applications
• **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 (several hundred)
  – 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 provides 3 interfaces to the underlying counter hardware:

  – A low level API manages hardware events (preset and native) 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 (preset only). Meant for programmers wanting simple event measurements.

  – 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
#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 predefined 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)
• Motivation

• Concepts and Definitions
  – Instrumentation, monitoring, analysis

• Some tools and their functionality
  – PAPI – access to hardware performance counters
  – ompP – profiling OpenMP code
  – IPM – monitoring message passing applications
ompP: Profiling tool for OpenMP
- Based on source code instrumentation
- Independent of the compiler and runtime used
- Tested and supported: Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers
- Supports HW counters through PAPI
- Uses source code instrumenter Opapi from the KOJAK/SCALASCA toolset
- Available for download (GPL): http://www.ompp-tool.com
ompP’s Profiling Report

• Header
  – Date, time, duration of the run, number of threads, used hardware counters,…

• Region Overview
  – Number of OpenMP regions (constructs) and their source-code locations

• Flat Region Profile
  – Inclusive times, counts, hardware counter data

• Callgraph

• Callgraph Profiles
  – With Inclusive and exclusive times

• Overhead Analysis Report
  – Four overhead categories
  – Per-parallel region breakdown
  – Absolute times and percentages
Example profiling data

Code:

```c
#pragma omp parallel
{
    #pragma omp critical
    {  
sleep(1.0);
    }
}
```

Profile:

<table>
<thead>
<tr>
<th>TID</th>
<th>execT</th>
<th>execC</th>
<th>bodyT</th>
<th>enterT</th>
<th>exitT</th>
<th>PAPI_TOT_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.00</td>
<td>1</td>
<td>1.00</td>
<td>2.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>6347</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>3</td>
<td>4.00</td>
<td>1</td>
<td>1.00</td>
<td>3.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>SUM</td>
<td>10.01</td>
<td>4</td>
<td>4.00</td>
<td>6.00</td>
<td>0.00</td>
<td>11132</td>
</tr>
</tbody>
</table>

Components:

- Source code location and type of region
- Timing data and execution counts, depending on the particular construct
- One line per thread, last line sums over all threads
- Hardware counter data (if PAPI is available and HW counters are selected)
- Data is “exact” (measured, not based on sampling)
### Flat Region Profile (2)

- Times and counts reported by ompP for various OpenMP constructs

<table>
<thead>
<tr>
<th>construct</th>
<th>exec</th>
<th>execC</th>
<th>enterT</th>
<th>startupT</th>
<th>bodyT</th>
<th>sectionT</th>
<th>sectionC</th>
<th>singleT</th>
<th>singleC</th>
<th>exitBarT</th>
<th>exitT</th>
<th>shutdownT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BARRIER</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER REGION</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRITICAL</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECTIONS</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ends with **T**: time

Ends with **C**: count

Main = enter + body + barr + exit
Callgraph

- **Callgraph View**
  - ‘Callgraph’ or ‘region stack’ of OpenMP constructs
  - Functions can be included by using Opari’s mechanism to instrument user defined regions: #pragma pomp inst begin(...), #pragma pomp inst end(...)

- **Callgraph profile**
  - Similar to flat profile, but with inclusive/exclusive times

- **Example:**

```c
main()
{
    #pragma omp parallel
    {
        foo1();
        foo2();
    }
}

void foo1()
{
    bar();
}

void foo2()
{
    bar();
}

void bar()
{
    #pragma omp critical
    {
        sleep(1.0);
    }
}
```
Callgraph (2)

- Callgraph display

<table>
<thead>
<tr>
<th>Incl. CPU time</th>
<th>[APP 4 threads]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.22 (100.0%)</td>
<td></td>
</tr>
<tr>
<td>32.06 (99.50%)</td>
<td>PARALLEL +R00004 main.c (42-46)</td>
</tr>
<tr>
<td>10.02 (31.10%)</td>
<td>USERREG</td>
</tr>
<tr>
<td>10.02 (31.10%)</td>
<td>CRITICAL</td>
</tr>
<tr>
<td>16.03 (49.74%)</td>
<td>USERREG</td>
</tr>
<tr>
<td>16.03 (49.74%)</td>
<td>CRITICAL</td>
</tr>
</tbody>
</table>

- Callgraph profiles

```plaintext
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
TID  execT/I  execT/E  execC
0    1.00     0.00     1
1    3.00     0.00     1
2    2.00     0.00     1
3    4.00     0.00     1
SUM  10.01    0.00     4

[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
[=03] R00003 main.c (33-36) (unnamed) CRITICAL
TID  execT  execC  bodyT/I  bodyT/E  enterT  exitT
0    1.00    1      1.00     1.00     0.00    0.00
1    3.00    1      1.00     1.00     2.00    0.00
2    2.00    1      1.00     1.00     1.00    0.00
3    4.00    1      1.00     1.00     3.00    0.00
SUM  10.01   4      4.00     4.00     6.00    0.00
```
• Certain timing categories reported by ompP can be classified as overheads:
  – Example: `exitBarT`: time wasted by threads idling at the exit barrier of work-sharing constructs. Reason is most likely an imbalanced amount of work

• Four overhead categories are defined in ompP:
  – **Imbalance**: waiting time incurred due to an imbalanced amount of work in a worksharing or parallel region
  
  – **Synchronization**: overhead that arises due to threads having to synchronize their activity, e.g. `barrier` call
  
  – **Limited Parallelism**: idle threads due not enough parallelism being exposed by the program
  
  – **Thread management**: overhead for the creation and destruction of threads, and for signaling critical sections, locks as available
### Overhead Analysis (2)

<table>
<thead>
<tr>
<th>construct</th>
<th>main</th>
<th>enter</th>
<th>startup</th>
<th>body</th>
<th>section</th>
<th>sectionC</th>
<th>singleT</th>
<th>singleC</th>
<th>exitBar</th>
<th>exitT</th>
<th>shutdownT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BARRIER</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER REGION</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRITICAL</td>
<td>●</td>
<td>●</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(M)</td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>●</td>
<td>●</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(I)</td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(I)</td>
</tr>
<tr>
<td>SECTIONS</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●(I/L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●(L)</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>●</td>
<td>●</td>
<td>●(M)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(I)</td>
<td>●</td>
<td>●(M)</td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>●</td>
<td>●</td>
<td>●(M)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(I)</td>
<td>●</td>
<td>●(M)</td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●(M)</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●(I/L)</td>
<td>●</td>
<td>●(M)</td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●(M)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●(I)</td>
<td>●</td>
<td>●(M)</td>
</tr>
</tbody>
</table>

**S:** Synchronization overhead  
**M:** Thread management overhead  
**I:** Imbalance overhead  
**L:** Limited Parallelism overhead
ompP’s Overhead Analysis Report

--- ompP Overhead Analysis Report ---

<table>
<thead>
<tr>
<th>Total runtime (wallclock)</th>
<th>172.64 sec [32 threads]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parallel regions</td>
<td>12</td>
</tr>
<tr>
<td>Parallel coverage</td>
<td>134.83 sec (78.10%)</td>
</tr>
</tbody>
</table>

Parallel regions sorted by wallclock time:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011 PARALL</td>
<td>mgrid.F (360-384)</td>
<td>55.75 (32.29)</td>
</tr>
<tr>
<td>R00019 PARALL</td>
<td>mgrid.F (403-427)</td>
<td>23.02 (13.34)</td>
</tr>
<tr>
<td>R00009 PARALL</td>
<td>mgrid.F (204-217)</td>
<td>11.94 ( 6.92)</td>
</tr>
</tbody>
</table>

SUM 134.83 (78.10)

Overheads wrt. each individual parallel region:

<table>
<thead>
<tr>
<th>Total</th>
<th>Ovhd (%)</th>
<th>Synch (%)</th>
<th>Imbal (%)</th>
<th>Limpar (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011</td>
<td>1783.95</td>
<td>337.26 (18.91)</td>
<td>0.00 (0.00)</td>
<td>305.75 (17.14)</td>
<td>31.51 (1.77)</td>
</tr>
<tr>
<td>R00019</td>
<td>736.80</td>
<td>129.95 (17.64)</td>
<td>0.00 (0.00)</td>
<td>104.28 (14.15)</td>
<td>25.66 (3.48)</td>
</tr>
<tr>
<td>R00009</td>
<td>382.15</td>
<td>183.14 (47.92)</td>
<td>0.00 (0.00)</td>
<td>96.47 (25.24)</td>
<td>86.67 (22.68)</td>
</tr>
<tr>
<td>R00015</td>
<td>276.11</td>
<td>68.85 (24.94)</td>
<td>0.00 (0.00)</td>
<td>51.15 (18.52)</td>
<td>17.70 (6.41)</td>
</tr>
</tbody>
</table>

SUM 4314.62 1277.89 (23.13)

Overheads wrt. whole program:

<table>
<thead>
<tr>
<th>Total</th>
<th>Ovhd (%)</th>
<th>Synch (%)</th>
<th>Imbal (%)</th>
<th>Limpar (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011</td>
<td>1783.95</td>
<td>337.26 (6.10)</td>
<td>0.00 (0.00)</td>
<td>305.75 (5.53)</td>
<td>31.51 (0.57)</td>
</tr>
<tr>
<td>R00009</td>
<td>382.15</td>
<td>183.14 (3.32)</td>
<td>0.00 (0.00)</td>
<td>96.47 (1.75)</td>
<td>86.67 (1.57)</td>
</tr>
<tr>
<td>R00005</td>
<td>264.16</td>
<td>164.90 (2.98)</td>
<td>0.00 (0.00)</td>
<td>63.92 (1.16)</td>
<td>100.98 (1.83)</td>
</tr>
<tr>
<td>R00007</td>
<td>230.63</td>
<td>151.91 (2.75)</td>
<td>0.00 (0.00)</td>
<td>68.58 (1.24)</td>
<td>83.33 (1.51)</td>
</tr>
</tbody>
</table>

SUM 4314.62 1277.89 (23.13)
OpenMP Scalability Analysis

• Methodology
  – Analyze how overheads behave for increasing thread counts
  – Graphs show accumulated runtime over all threads for fixed workload (strong scaling)
  – Horizontal line = perfect (linear) scalability
Example

- NAS Parallel Benchmarks
- Class C, SGI Altix machine (Itanium 2, 1.6 GHz, 6MB L3 Cache)
Application 314.mgrid_m

- Scales relatively poorly, application has 12 parallel loops, all contribute with increasingly severe load imbalance
- Smaller load imbalance for thread counts of 32 and 16. Only three loops show this behavior
- In all three cases, the iteration count is always a power of two (2 to 256), hence thread counts which are not a power of two exhibit more load imbalance
SPEC OpenMP Benchmarks (2)

- Application 316.applu
  - Super-linear speedup
  - Only one parallel region (ssor.f 138-209) shows super-linear speedup, contributes 80% of accumulated total execution time
  - Most likely reason for super-linear speedup: increased overall cache size

![Graph showing 316.ovhds.dat and L3_Misses]
- Application 313.swim
  - Dominating source of inefficiency is thread management overhead
  - Main source: reduction of three scalar variables in a small parallel loop in swim.f 116-126.
  - At 128 threads more than 6 percent of the total accumulated runtime is spent in the reduction operation
  - Time for the reduction operation is larger than time spent in the body of the parallel region
• Application 318.galgel
  – Scales very poorly, large fraction of overhead not accounted for by ompP (most likely memory access latency, cache conflicts, false sharing)
  – lapack.f90 5081-5092 contributes significantly to the bad scaling
    » accumulated CPU time increases from 107.9 (2 threads) to 1349.1 seconds (32 threads)
    » 32 thread version is only 22% faster than 2 thread version (wall-clock time)
    » 32 thread version parallel efficiency is only approx. 8%

8/21/2009 Karl Fuerlinger Performance Tools: 44
Outline

• Motivation

• Concepts and Definitions
  – Instrumentation, monitoring, analysis

• Some tools and their functionality
  – PAPI – access to hardware performance counters
  – ompP – profiling OpenMP code
  – IPM – monitoring message passing applications
• IPM provides a performance profile of a job
  – „Flip of a switch“ operation
IPM: Design Goals

• Provide high-level performance profile
  – *Event inventory*: which events happened and how much time did they take
  – How much time in communication operations
  – Less focus on drill-down into application than other tools

• Efficiency
  – Fixed memory footprint (approx. 1-2 MB per MPI rank)
  – Monitoring data is kept in a hash-table, avoid dynamic memory allocation
  – Low CPU overhead: 1-2%

• Ease of use
  – HTML, or ASCII-based output format
  – Flip of a switch, no recompilation, no user instrumentation
  – Portability
IPM: Methodology

• MPI_Init()
  – Initialize monitoring environment, allocate memory

• For each MPI call
  – Compute hash key from
    » Type of call (send/recv/bcast/...)
    » Buffer size (in bytes)
    » Communication partner rank
    » Call-site, region or phase identifier, ...
  – Store / update value in hash table with timing data
    » Number of invocations
    » Minimum duration, maximum duration, summed time

• MPI_Finalize()
  – Aggregate, report to stdout, write XML log
Using IPM: Basics

- Do “module load ipm”, then run normally
- Upon completion you get

```
###IPMv0.85############################################################################
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                         mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55              wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17              %comm     : 32.43
# gbytes  : 2.57604e+00 total             gflop/sec : 2.04615e+00 total
###############################################################################
```

Maybe that’s enough. If so you’re done.

Have a nice day.
### IPMv0.85

**command**: `../exe/pmemd -O -c inpcrd -o res` (completed)

**host**: s05405  
**mpi_tasks**: 64 on 4 nodes

**start**: 02/22/05/10:03:55  
**wallclock**: 24.278400 sec

**stop**: 02/22/05/10:04:17  
**%comm**: 32.43

**gbytes**: 2.57604e+00 total  
**gflop/sec**: 2.04615e+00 total

#### [total] <avg> min max

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>wallclock</td>
<td>1373.67</td>
<td>21.4636</td>
<td>21.1087</td>
<td>24.2784</td>
</tr>
<tr>
<td>user</td>
<td>936.95</td>
<td>14.6398</td>
<td>12.68</td>
<td>20.3</td>
</tr>
<tr>
<td>system</td>
<td>227.7</td>
<td>3.55781</td>
<td>1.51</td>
<td>5</td>
</tr>
<tr>
<td>mpi</td>
<td>503.853</td>
<td>7.8727</td>
<td>4.2293</td>
<td>9.13725</td>
</tr>
<tr>
<td>%comm</td>
<td>32.4268</td>
<td>17.42</td>
<td>41.407</td>
<td></td>
</tr>
<tr>
<td>gflop/sec</td>
<td>2.04614</td>
<td>0.0319709</td>
<td>0.02724</td>
<td>0.04041</td>
</tr>
<tr>
<td>gbytes</td>
<td>2.57604</td>
<td>0.0402507</td>
<td>0.0399284</td>
<td>0.0408173</td>
</tr>
<tr>
<td>gbytes_tx</td>
<td>0.665125</td>
<td>0.0103926</td>
<td>1.09673e-05</td>
<td>0.0368981</td>
</tr>
<tr>
<td>gbyte_rx</td>
<td>0.659763</td>
<td>0.0103088</td>
<td>9.83477e-07</td>
<td>0.0417372</td>
</tr>
</tbody>
</table>
Want more detail? IPM\_REPORT=full

<table>
<thead>
<tr>
<th>#</th>
<th>[time]</th>
<th>[calls]</th>
<th>&lt;%mpi&gt;</th>
<th>&lt;%wall&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_Cyc</td>
<td>3.00519e+11</td>
<td>4.69561e+09</td>
<td>4.50223e+09</td>
<td>5.83342e+09</td>
</tr>
<tr>
<td>PM_FPU0_CMPL</td>
<td>2.45263e+10</td>
<td>3.83223e+08</td>
<td>3.3396e+08</td>
<td>5.12702e+08</td>
</tr>
<tr>
<td>PM_FPU1_CMPL</td>
<td>1.48426e+10</td>
<td>2.31916e+08</td>
<td>1.90704e+08</td>
<td>2.8053e+08</td>
</tr>
<tr>
<td>PM_FPU_FMA</td>
<td>1.03083e+10</td>
<td>1.61067e+08</td>
<td>1.36815e+08</td>
<td>1.96841e+08</td>
</tr>
<tr>
<td>PM_INST_CMPL</td>
<td>3.33597e+11</td>
<td>5.21245e+09</td>
<td>4.33725e+09</td>
<td>6.44214e+09</td>
</tr>
<tr>
<td>PM_LD_CMPL</td>
<td>1.03239e+11</td>
<td>1.61311e+08</td>
<td>1.36815e+08</td>
<td>1.96841e+08</td>
</tr>
<tr>
<td>PM_ST_CMPL</td>
<td>7.19365e+10</td>
<td>1.12401e+09</td>
<td>8.77684e+08</td>
<td>1.29017e+09</td>
</tr>
<tr>
<td>PM_TLB_MISS</td>
<td>1.67892e+08</td>
<td>2.62332e+06</td>
<td>1.16104e+06</td>
<td>2.36664e+07</td>
</tr>
</tbody>
</table>

8/21/2009 Karl Fuerlinger Performance Tools: 51
There’s a lot more information in the logfile than you get to stdout. A logfile is written that has the hash table, switch traffic, memory usage, executable information, ...

Parallelism in writing of the log (when possible)

The IPM logs are durable performance profiles serving

- HPC center production needs:
  https://www.nersc.gov/nusers/status/llsum/
  http://www.sdsc.edu/user_services/top/ipm/

- HPC research: ipm_parse renders txt and html
  http://www.nersc.gov/projects/ipm/ex3/

- your own XML consuming entity, feed, or process
Message Sizes: CAM 336 way

per MPI call

per MPI call & buffer size
Scalability: Required

32K tasks AMR code
More than a pretty picture
Which problems should be tackled with IPM?

• Performance Bottleneck Identification
  – Does the profile show what I expect it to?
  – Why is my code not scaling?
  – Why is my code running 20% slower than I expected?

• Understanding Scaling
  – Why does my code scale as it does?

• Optimizing MPI Performance
  – Combining Messages
Application Assessment with IPM

- Provide high level performance numbers with small overhead
  - To get an initial read on application runtimes
  - For allocation/reporting
  - To check the performance weather on systems with high variability

- What’s going on overall in my code?
  - How much comp, comm, I/O?
  - Where to start with optimization?

- How is my load balance?
  - Domain decomposition vs. concurrency (M work on N tasks)
When to reach for another tool

- Full application tracing
- Looking for hotspots on the statement level in code
- Want to step through the code
- Data structure level detail
- Automated performance feedback
What’s wrong here?

Communication

% of MPI Time

Communication Event Statistics (100.00% detail)

<table>
<thead>
<tr>
<th>Event</th>
<th>Buffer Size</th>
<th>Ncalls</th>
<th>Total Time</th>
<th>Min Time</th>
<th>Max Time</th>
<th>%MPI</th>
<th>%Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td>8</td>
<td>3278848</td>
<td>124132.547</td>
<td>0.00</td>
<td>114.920</td>
<td>59.35</td>
<td>16.88</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0</td>
<td>35173439489</td>
<td>43439.102</td>
<td>0.00</td>
<td>41.961</td>
<td>20.77</td>
<td>5.91</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>98304</td>
<td>13221888</td>
<td>15710.953</td>
<td>0.00</td>
<td>3.586</td>
<td>7.51</td>
<td>2.14</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>196608</td>
<td>13221888</td>
<td>5331.236</td>
<td>0.00</td>
<td>5.716</td>
<td>2.55</td>
<td>0.72</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>589824</td>
<td>206848</td>
<td>5166.272</td>
<td>0.00</td>
<td>7.265</td>
<td>2.47</td>
<td>0.70</td>
</tr>
</tbody>
</table>

5/21/2009 Karl Fuerlinger
Performance Tools: 59
Is MPI_BARRIER time bad? Probably. Is it avoidable?

- The stray / unknown / debug barrier
- Barriers used for I/O ordering
Summary

• Performance monitoring concepts
  – Instrument, measure, analyze
  – Profiling/tracing, sampling, direct measurement

• Tools
  – PAPI, ompP, IPM as examples

• Lots of other tools
  – Vendor tools: Cray PAT, Sun Studio, Intel Thread Profiler, Vtune, PTU,…
  – Portable tools: TAU, Perfsuite, Paradyn, HPCToolkit, Kojak, Scalasca, Vampir, oprofile, gprof, …

Thank you for your attention!