Studying the behavior of parallel applications and identifying bottlenecks by using performance analysis tools

G. Markomanolis

INRIA, LIP, ENS de Lyon

Ecole IN2P3 d’informatique "Méthodologie et outils d’optimisation en développement logiciel" 9 February 2012
Outline

1. Context and motivation
2. Introduction to Performance Engineering
3. Performance Application Programming Interface
4. Scalasca
   - Scalasca hands-on
5. TAU
   - TAU hands-on
6. PerfExpert
7. Score-P
8. Talk about accuracy
Goals

- Overview of the programming tools suite
- Explain the functionality of the tools
- Teach how to use the tools
- Hands-on
Introduction to Performance Engineering

Performance factors of parallel applications
  • “Sequential” factors
    ▶ Computation
    ▶ Cache and memory
    ▶ Input/output
  • “Parallel” factors
    ▶ Partitioning/decomposition
    ▶ Communication
    ▶ Multithreading
    ▶ Synchronization
    ▶ Parallel I/O
    ▶ Mapping
Performance engineering workflow

- Prepare application
- Collect the relevant data to the execution of the instrumented application
- Identification of performance metrics
- Presentation of results
- Modifications in order to reduce performance problems
Metrics of performance

- How often an event occurs
- The duration of some intervals, e.g. the time spent some communication calls
- The size of the messages during the communication
- Derived metrics
Example

- Execution time
- How many times a function is called
- IPC (Instructions per cycle)
- FLOPS
Execution time

- **Wall-clock time**
  - Includes waiting time
  - Includes the time consumed by other applications in time-sharing environments

- **CPU time**
  - Time spent by the CPU for the application
  - No measurement of the context-switched out time

- Use mean or minimum of several runs
Inclusive vs. Exclusive values

```plaintext
int foo()
{
    int a;
    a = 1 + 1;
    bar();
    a = a + 1;
    return a;
}
```
Measurement techniques

Methods for the measurement
  ▶ Sampling
  ▶ Code instrumentation

Record the data
  ▶ Profiling
  ▶ Tracing
Sampling

```c
int main()
{
    int i;

    for (i=0; i < 3; i++)
        foo(i);

    return 0;
}

void foo(int i)
{
    if (i > 0)
        foo(i - 1);
}
```

- Statistical inference of program behaviour
- Not very detailed information
- Only for long-running applications
- Unmodified executables
Instrumentation

- Every event is captured
- Detailed information
- Processing of source-code or executable
- Overhead

```c
int main()
{
    int i;
    Enter("main");
    for (i=0; i < 3; i++)
        foo(i);
    Leave("main");
    return 0;
}

void foo(int i)
{
    Enter("foo");
    if (i > 0)
        foo(i - 1);
    Leave("foo");
}
```
Instrumentation techniques

- Static instrumentation
- Dynamic instrumentation
- Code Modification
  - Manually
  - Automatically
    - Preprocessor
    - Compiler
    - Linking against a pre-instrumented library
    - Binary-rewrite
Critical issues

- **Accuracy**
  - Intrusion overhead
  - Perturbation
  - Accuracy of times & counters

- **Granularity**
  - Number of measurements?
  - How much information?
Profiling

- Record of aggregated information
  - Total, maximum ...

- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
  - Functions, call sites
  - Processes, threads
Types of profiles

- Flat profile
  - Metrics per routine for the instrumented region
  - Calling context is not taken into account

- Call-path profile
  - Metrics per executed call path
  - Distinguished by partial calling context

- Special profiles
  - Profile specific events, e.g. MPI calls
  - Comparing processes/threads
Tracing I

- Recording all the events for the demanded code
  - Enter/leave of a region
  - Send/receive a message
- Extra information in event record
  - Timestamp, location, event type
  - Event-related info (e.g., communicator, sender/receiver)
- Chronologically ordered sequence of event records
Tracing II

Process A

```c
void foo() {
    trc_enter("foo");
    ...
    trc_send(B);
    send(B, tag, buf);
    ...
    trc_exit("foo");
}
```

Local trace A

- 58 ENTER 1
- 62 SEND B
- 64 EXIT 1

Local trace B

- 60 ENTER 1
- 68 RECV A
- 69 EXIT 1

Global trace

- 58 A ENTER 1
- 60 B ENTER 2
- 62 A SEND B
- 64 A EXIT 1
- 68 B RECV A
- 69 B EXIT 2

Process B

```c
void bar() {
    trc_enter("bar");
    ...
    recv(A, tag, buf);
    trc_recv(A);
    ...
    trc_exit("bar");
}
```
Tracing vs. Profiling

- **Tracing advantages**
  - It is possible to reconstruct the dynamic application behaviour on any required level of abstraction
  - From the tracing it is possible to extract the profiling.

- **Disadvantages**
  - The traces can get really large especially when using a lot of processes or the applications is constituted by many events
  - Writing events to a file at runtime causes perturbation
Performance analysis procedure

- Performance problem?
  - Time / speedup / scalability measurements

- Key bottleneck?
  - MPI/ OpenMP / Flat profiling

- Where is the key bottleneck?
  - Call-path profiling

- Why?
  - Hardware counter analysis, selective instrumentation for better analysis

- Scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function by function
Performance Application Programming (PAPI)

Middleware that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors. Hardware performance counters can provide insight into:

- Whole program timing
- Cache behaviors
- Branch behaviors
- Memory and resource contention and access patterns
- Pipeline stalls
- Floating point efficiency
- Instructions per cycle
- Subroutine resolution
- Process or thread attribution
PAPI

- Events
  - Platform-neutral Present Events (e.g., PAPI_TOT_INS)
  - Platform-dependent Native Events (e.g., L3_CACHE_MISS)

- Present Events
  - Standard set of over 100 events for application performance tuning (not all of them available on every processor)
  - No standardization of the exact definition
  - Mapped to either single or linear combinations of native events on each platform
  - The `papi_avail` provides the available preset events on a given platform

- Native events
  - All the countable events by the CPU
  - Same interface as for preset events
  - The `papi_native_avail` provides the available native events on a given platform

- It is needed to use the tool `papi_event_chooser` in order to find out the compatible set of events that can be measured at the same moment
Intel XEON X5675

% papi_avail
Available events and hardware information.

PAPI Version : 4.2.0.0
Vendor string and code : GenuineIntel (1)
Model string and code : Intel(R) Xeon(R) CPU X5675 @ 3.07GHz (44)
CPU Revision : 2.000000
CPUID Info : Family: 6 Model: 44 Stepping: 2
CPU Megahertz : 3066.216064
CPU Clock Megahertz : 3066
Hdw Threads per core : 2
Cores per Socket : 6
NUMA Nodes : 2
CPU’s per Node : 12
Total CPU’s : 24
Number Hardware Counters : 7
Max Multiplex Counters : 64

The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>API_VEC_SP</td>
<td>0x80000069</td>
<td>Yes</td>
<td>No</td>
<td>Single precision vector/SIMD instr.</td>
</tr>
<tr>
<td>PAPI_VEC_DP</td>
<td>0x8000006a</td>
<td>Yes</td>
<td>No</td>
<td>Double precision vector/SIMD instr.</td>
</tr>
</tbody>
</table>

Of 107 possible events, 57 are available, of which 14 are derived.
PAPI Counter Interfaces

PAPI provides 3 interfaces:

- Low Level API manages hardware events in user defined groups called EventSets
- High Level API provides the ability to start, stop and read the counters for a specific list of events
- Graphical and and-user tools provide facile data collection and visualization
Component PAPI (PAPI-C)

- **Motivation:**
  - Hardware counters for network counters, thermal & power measurement
  - Measure multiple counter domains at once

- **Goals:**
  - Isolate hardware dependent code in a separable component module
  - Add or modify API calls to support access to various components
Component PAPI

PAPI FRAMEWORK

PAPI COMPONENT (NETWORK)
  Operating System
  Counter Hardware

PAPI COMPONENT (CPU)
  Operating System
  Counter Hardware

PAPI COMPONENT (THERMAL)
  Operating System
  Counter Hardware
Scalable performance analysis of large-scale parallel applications

Scalasca
Techniques

- **Profile analysis:**
  - Summary of aggregated metrics
    - per function/call-path and/or per process/thread
  - mpiP, TAU, PerfSuite, Vampir

- **Time-line analysis**
  - Visual representation of the space/time sequence of events
  - An execution is demanded

- **Pattern analysis**
  - Search for characteristic event sequences in event traces
  - Manually: Visual time-line analysis
  - Automatically: Scalasca
Automatic trace analysis

- Trace an application
- Automatic search for patterns on inefficient behaviour
- Classification of behaviour
- Much faster than manual trace analysis
- Scalability
Overview

- Supports parallel programming paradigms & languages
  - MPI, OpenMP, OpenMP/MPI
  - Fortran, C, C++
- Profiling, Tracing, Event trace analysis
Instrumentation

- Code instrumentation
- Add instrumentation and measurement library into application executable
- MPI standard profiling interface (PMPI) to acquire MPI events
Measurement runtime summarization

- Measurements summarized by thread & call-path during execution
- Analysis report unified
- Presentation of summary analysis
Measurement event tracing & analysis

- Time-stamped events buffered for each thread
- Flushed to files
- Trace analysis
- Presentation of analysis report
Measurement event tracing & analysis

- Scalasca instrumenter (SKIN)
- Scalasca measurement collector & analyzer (SCAN)
- Scalasca analysis report examiner (SQUARE)
EPIK

- Measurement & analysis runtime system
  - Manages runtime configuration and parallel execution
  - Configuration specified by EPIK.CONF (epik_conf)
  - An experiment archive is created (epik_<title>)
  - Optional:
    - Runtime summarization report
    - Tracing
    - Filtering of events
    - Hardware counter measurements

- Experiment archive directory
  - Contains measurement and related files
  - Contains analysis reports
### Scalasca actions

<table>
<thead>
<tr>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalasca -instrument</td>
</tr>
<tr>
<td>scalasca -analyze</td>
</tr>
<tr>
<td>scalasca -examine</td>
</tr>
</tbody>
</table>
Parallel program analysis report exploration tools
  - Libraries for XML report
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration

Used by Scalasca, Marmot, ompP, PerfSuite, etc.
CUBE3 Analysis

- Three coupled tree browsers
  - Performance property
  - Call-tree path
  - System location

- CUBE3 displays severities
  - Value for precise comparison
  - Colour for easy identification of hotspots
  - Inclusive value when closed and exclusive value when expanded
  - Customizable through display mode
CUBE3 - summary

Figure out the kind of the performance problem

Relation between the problem and the source code

Distribution across the processes
CUBE3 - trace
Scalasca summary: LU benchmark, class B, (NPB) on AMD1 node

- 11.69% of time spent in MPI point-to-point communication
- 97.35% of which is on program callpath MAIN/SSOR
- With 17.0% std dev over 32 processes
Scalasca trace: LU benchmark, class B, (NPB) on AMD1 node

- We can observe that the MPI point-to-point is separated to Late Sender and Late receiver.
- Late Sender is the 53.57% of the total execution.
- 99.88% of this time is in SSOR.
Scalasca 1.4

- Automatic function instrumentation and filtering
  - GCC, IBM, Intel, Pathscale & PGI compilers
  - Optional PDToolkit selective instrumentation
  - Declare which functions to exclude or include for the instrumentation

- MPI measurements & analysis
  - Scalable runtime summarization & event tracing
  - Just re-link the application executable

- OpenMP measurement & analysis
  - Demanded application source instrumentation
  - Thread management

- Hybrid OpenMP/MPI measurement & analysis
  - Combined the previous
Scalasca 1.4

- Measurement configuration of MPI events wrappers
  - P2P, COLL, ENV, CG, TOPO, ...
- MPI RMA communication analysis
- Reduced runtime overhead & lowered distortion at scale
Hands-on: NPB-MPI / LU

Scalasca
Performance analysis steps

1. Program instrumentation: skin
2. Summary measurement collection & analysis: scan [-s]
3. Summary analysis report examination: square
4. Summary experiment scoring: square -s
5. Event trace collection & analysis: scan -t
6. Event trace analysis report examination: square

- Configuration & customization
  - Instrumentation, Measurement, Analysis, Presentation
Connect to the nodes with enabled graphics connection

% ssh -X username@intelnode
% ssh -X username@amd1node
% ssh -X username@amd2node
NPB-MPI suite

- The NAS Parallel Benchmark suite
  - Download from
    http://www.nas.nasa.gov/publications/npb.html
  - 9 benchmarks
    - Configurable for various sizes & size of problems

- Copy the NAS to your home folder

  ```bash
  % cp -r /srv/app/data/tutorial .
  ```
NAS Parallel Benchmarks (NPB):

- **Mixed case**: LU factorization (LU)
  - Instances
    - From 2 to 32 processes
    - Classes A and B

- **Compile**

  % make LU NPROCS=<number> CLASS=<class>

  - Where `<number>` is the number of the processes power of two and `<class>` is the letter of the class, S,W,A,B,C,D or E
NPB - MPI / LU

- Studying the MPI version of the LU benchmark from the NAS Parallel Benchmarks (NPB) suite
- Summary measurement & analysis
  - Automatic instrumentation
  - Summary analysis report examination
  - PAPI hardware counter metrics
- Trace measurement collection & analysis
  - Filter determination, specification & configuration
  - Automatic trace analysis report patterns
- Manual and PDT instrumentation
- Measurement configuration
- Analysis report algebra
Scalasca usage

- Execute **scalasca**

```
% scalasca
Scalasca 1.4
Toolset for scalable performance analysis of large-scale parallel applications usage: scalasca [-v][-n] {action}
  1. prepare application objects and executable for measurement:
     scalasca -instrument <compile-or-link-command> # skin
  2. run application under control of measurement system:
     scalasca -analyze <application-launch-command> # scan
  3. interactively explore measurement analysis report:
     scalasca -examine <experiment-archive|report> # square

-v: enable verbose commentary
-n: show actions without taking them
-h: show quick reference guide (only)
```
NPB instrumentation

- Go to the NAS MPI root path
  ```
  % cd ~/tutorial/NPB3.3-MPI
  ```

- Add compile/link commands in Makefile (config/make.def)
  ```
  MPIF77=scalasca -instrument mpif77
  ```
  or
  ```
  MPIF77=$(PREP) mpif77
  ```

- Clean up any previous file
  ```
  % make clean
  ```

- Compile the LU benchmark for class A and 8 processors
  ```
  % make LU CLASS=A NPROCS=8
  ```
  or
  ```
  % make LU CLASS=A NPROCS=8 PREP=``scalasca -instrument''
  ```
LU summary measurement

- Enter the folder with the executables that are instrumented by Scalasca
  
  ```
  % cd bin.scalasca
  ```

- Execute the benchmark for 4 processes
  
  ```
  % scalasca -analyze mpirun -np 8 lu.A.8
  S=C=A=N: Scalasca 1.4 runtime summarization
  S=C=A=N: ./epik_lu_8_sum experiment archive
  /usr/bin/mpirun -np 8 lu.A.8
  [00000]EPIK: Created new measurement archive ./epik_lu_8_sum
  [00000]EPIK: Activated ./epik_lu_8_sum [NO TRACE] (0.011s)
  [... output ...]
  [00000]EPIK: 69 unique paths (64 max paths, 5 max frames, 0 unknowns)
  [00000]EPIK: Unifying... done (0.002s)
  [00000]EPIK: Collating... done (0.002s)
  [00000]EPIK: Closed experiment ./epik_lu_8_sum (0.004s)
  maxHeap(*)=20.695/81.918MB
  S=C=A=N: Wed Jan 25 15:17:34 2012: Collect done (status=0) 17s
  S=C=A=N: ./epik_lu_8_sum complete.
  ```
LU summary measurement

- Execute the Scalasca GUI

```bash
% scalasca -examine epik_lu_8_sum
```

- The measurement archive directory contains
  - a file that contains the execution output (epik.log)
  - the current configuration (epik.conf)
  - the analysis report that was collated after measurement (epitome.cube)
  - the complete analysis report produced during post-processing (summary.cube.gz)
LU summary measurement view

4.79e6 Visits
16 Synchronizations
6.33e5 Communications
1.95e9 Bytes transferred
2.88 Computational imbalance
LU summary measurement, system tree

Distribution of the time across the processes
LU summary measurement, box plot

- 4.79e6 Visits
- 16 Synchronizations
- 6.33e5 Communications
- 1.95e9 Bytes transferred
- 2.88 Computational imbalance
LU summary measurement, topology

- 100.28 Time
- 4.79e6 Visits
- 6.33e5 Communications
- 1.95e9 Bytes transferred
- 2.88 Computational Imbalance
LU summary measurement, call tree

- 100.28 Time
- 4.79e6 Visits
- 15 Synchronizations
- 6.33e5 Communications
- 1.95e9 Bytes transferred
- 2.88 Computational imbalance

Call tree:
- 0.00 main
  - 0.87 init_comm_
  - 0.00 read_input_
  - 0.00 timer_clear_
  - 0.00 proc_grid_
  - 0.00 neighbors_
  - 0.00 subdomain_
  - 0.00 setcoeff_
  - 0.85 setbx_
  - 2.23 setlv_
  - 0.42 erhs_
  - 1.45 ssor_
    - 24.73 rhs_
    - 0.04 l2norm_
      - 0.00 timer_clear_
      - 0.00 MPI_Barrier
      - 0.00 timer_start_
      - 13.34 jactd_
      - 16.33 bits_
      - 12.83 jacu_
    - 19.96 buts
      - 0.00 timer_stop_
      - 0.00 timer_read_
      - 0.12 MPI_Allreduce
    - 0.10 error_
    - 0.00 pintor

System tree:
- - Linux Cluster
  - - amdinode
    - 3.25 Process 0
    - 2.52 Process 1
    - 2.39 Process 2
    - 2.14 Process 3
    - 2.91 Process 4
    - 2.38 Process 5
    - 2.30 Process 6
    - 2.07 Process 7
LU summary measurement, metric tree
LU summary measurement, source browser
What we did till now

- Instrument an application
- Analyze its execution with a summary measurement
- Examine it with the interactive analysis report explorer GUI
- Time metrics
- Visit counts
- MPI message statistics
- Computational imbalance
LU summary analysis result scoring

% scalasca -examine -s epik.lu.8_sum/
/srv/app/scalasca/bin/cube3_score -r ./epik.lu.8_sum/summary.cube.gz > ./epik.lu.8_sum/epik.score

Reading ./epik.lu.8_sum/summary.cube.gz... done.
Estimated aggregate size of event trace (total_tbc): 130098176 bytes
Estimated size of largest process trace (max_tbc): 17275190 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes or reduce requirements using file listing names of USR regions to be filtered.)

INFO: Score report written to ./epik.lu.8_sum/epik.score

- The estimated size of the traces will be 130MB
- The maximum trace buffer is around to 17.3MB per process
  - If the available buffer is smaller than 17.3MB, then there will be perturbation because of flushes to the hard disk during the measurement

- Region classification
  - MPI (pure MPI library functions)
  - OMP (pure OMP functions)
  - USR (user-level source local computation)
  - COM(combined USR with OpenMP/MPI)
  - ANY/ALL (aggregate of all region types)
## LU summary analysis report

```plaintext
% less epik_lu_8_sum/epik.score

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>17275190</td>
<td>100.28</td>
<td>100.00</td>
<td>(summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>4574534</td>
<td>19.04</td>
<td>18.99</td>
<td>(summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>2259600</td>
<td>52.82</td>
<td>52.67</td>
<td>(summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>10441032</td>
<td>28.30</td>
<td>28.22</td>
<td>(summary) USR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>9692928</td>
<td>1.37</td>
<td>1.37</td>
<td>exact_</td>
</tr>
<tr>
<td>MPI</td>
<td>2372550</td>
<td>1.84</td>
<td>1.84</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>MPI</td>
<td>2147556</td>
<td>5.85</td>
<td>5.83</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>COM</td>
<td>1493952</td>
<td>0.46</td>
<td>0.46</td>
<td>exchange_1_</td>
</tr>
<tr>
<td>USR</td>
<td>373488</td>
<td>12.63</td>
<td>12.59</td>
<td>jacu_</td>
</tr>
<tr>
<td>COM</td>
<td>373488</td>
<td>13.06</td>
<td>13.02</td>
<td>blts_</td>
</tr>
<tr>
<td>USR</td>
<td>373488</td>
<td>13.34</td>
<td>13.30</td>
<td>jacld_</td>
</tr>
<tr>
<td>COM</td>
<td>373488</td>
<td>16.29</td>
<td>16.25</td>
<td>buts_</td>
</tr>
<tr>
<td>MPI</td>
<td>35190</td>
<td>0.02</td>
<td>0.01</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td>MPI</td>
<td>18360</td>
<td>2.43</td>
<td>2.42</td>
<td>MPI_Wait</td>
</tr>
<tr>
<td>COM</td>
<td>12192</td>
<td>0.70</td>
<td>0.70</td>
<td>exchange_3_</td>
</tr>
<tr>
<td>COM</td>
<td>6072</td>
<td>20.48</td>
<td>20.42</td>
<td>rhs_</td>
</tr>
<tr>
<td>USR</td>
<td>768</td>
<td>0.00</td>
<td>0.00</td>
<td>timer_clear_</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>48</td>
<td>0.94</td>
<td>0.94</td>
<td>setiv_</td>
</tr>
<tr>
<td>USR</td>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
<td>timer_start_</td>
</tr>
<tr>
<td>USR</td>
<td>48</td>
<td>0.02</td>
<td>0.02</td>
<td>setbv_</td>
</tr>
<tr>
<td>USR</td>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
<td>timer_stop_</td>
</tr>
<tr>
<td>USR</td>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
<td>timer_read_</td>
</tr>
<tr>
<td>EPK</td>
<td>48</td>
<td>0.11</td>
<td>0.11</td>
<td>TRACING</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>MPI_Finalize</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>0.04</td>
<td>0.04</td>
<td>error_</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>verify_</td>
</tr>
<tr>
<td>USR</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>print_results_</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>main</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>0.32</td>
<td>0.32</td>
<td>erhs_</td>
</tr>
</tbody>
</table>
```
LU summary analysis report

- The estimated size of the traces will be 130MB
- The maximum trace buffer is around to 17.3MB per process
  - If the available buffer is smaller than 17.3MB, then there will be perturbation because of flushes to the hard disk during the measurement
  ```
  export ELG_BUFFER=17300000
  ```
- 28.22% of the total execution is caused by USR regions
  - We should check if there is overhead because of frequently executed small routines
- Solutions:
  - Declare the appropriate buffer
  - Declare a filter file listing (USR) regions in order not to be measured
LU summary analysis report filtering

- We choose the USR regions with small percentage of the execution time and big trace buffer in comparison with the other regions

```
% cat lu.filtering
# filtering for the LU benchmark
exact_
```

- Report scoring with the corresponding filter file

```
% scalasca -examine -s -f lu.filtering ./epik_lu_8_sum/
/srv/app/scalasca/bin/cube3_score -f lu.filtering -r
./epik_lu_8_sum/summary.cube.gz >
./epik_lu_8_sum/epik.score_lu.filtering
Reading ./epik_lu_8_sum/summary.cube.gz... done.
Applying filter "lu.filtering":
Estimated aggregate size of event trace (total_tbc): 54560192 bytes
Estimated size of largest process trace (max_tbc): 7582262 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes.)
INFO: Score report written to
./epik_lu_8_sum/epik.score_lu.filtering
```

- Now the estimated size of the traces is 54.6MB, decreased by 58% in comparison with the non filtering approach
- The maximum trace buffer is 7.6MB
## LU summary analysis report with filtering

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>17275190</td>
<td>100.28</td>
<td>100.00 (summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>4574534</td>
<td>19.04</td>
<td>18.99 (summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>2259600</td>
<td>52.82</td>
<td>52.67 (summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>10441032</td>
<td>28.30</td>
<td>28.22 (summary) USR</td>
</tr>
<tr>
<td>FLT</td>
<td>9692928</td>
<td>1.37</td>
<td>1.37 (summary) FLT</td>
</tr>
<tr>
<td>ANY</td>
<td>7582262</td>
<td>98.91</td>
<td>98.63 (summary) ALL-FLT</td>
</tr>
<tr>
<td>MPI</td>
<td>4574534</td>
<td>19.04</td>
<td>18.99 (summary) MPI-FLT</td>
</tr>
<tr>
<td>COM</td>
<td>2259600</td>
<td>52.82</td>
<td>52.67 (summary) COM-FLT</td>
</tr>
<tr>
<td>USR</td>
<td>748152</td>
<td>26.93</td>
<td>26.85 (summary) USR-FLT</td>
</tr>
<tr>
<td>USR</td>
<td>9692928</td>
<td>1.37</td>
<td>1.37 exact_</td>
</tr>
<tr>
<td>MPI</td>
<td>2372550</td>
<td>1.84</td>
<td>1.84 MPI_Send</td>
</tr>
<tr>
<td>MPI</td>
<td>2147556</td>
<td>5.85</td>
<td>5.83 MPI_Recv</td>
</tr>
<tr>
<td>COM</td>
<td>1493952</td>
<td>0.46</td>
<td>0.46 exchange_1_</td>
</tr>
<tr>
<td>USR</td>
<td>373488</td>
<td>12.63</td>
<td>12.59 jacu_</td>
</tr>
<tr>
<td>COM</td>
<td>373488</td>
<td>13.06</td>
<td>13.02 blts_</td>
</tr>
</tbody>
</table>

- The mark + indicates the filtered routines
LU filtered summary measurement

- Save the previous measurement

  ```
  % mv epik_lu_8_sum epik_lu_8_sum_no_filter
  ```

- Enable the filtering and the corresponding file

  ```
  % export EPK_FILTER=lu.filtering
  ```

- Execute the LU benchmark for class A and 8 processes

  ```
  % scalasca -analyze mpirun -np 8 lu.A.8
  S=C=A=N: Scalasca 1.4 runtime summarization
  S=C=A=N: ./epik_lu_8_sum experiment archive
  S=C=A=N: Tue Apr 12 00:06:01 2011: Collect start
  /srv/app/openmpi/bin//mpirun -np 8 lu.A.8
  [00000]EPIK: Created new measurement archive ./epik_lu_8_sum
  [00000]EPIK: EPK_FILTER "lu.filtering" filtered 1 of 222 functions
  [00000]EPIK: Activated ./epik_lu_8_sum [NO TRACE] (0.009s)
  [... output ...]
  [00000]EPIK: 66 unique paths (61 max paths, 5 max frames, 0 unknowns)
  [00000]EPIK: Unifying... done (0.005s)
  [00000]EPIK: Collating... done (0.005s)
  [00000]EPIK: Closed experiment ./epik_lu_8_sum (0.011s)
  maxHeap(*)=15.469/121.660MB
  S=C=A=N: Tue Apr 12 00:06:14 2011: Collect done (status=0) 13s
  S=C=A=N: ./epik_lu_8_sum complete.
  ```
Examine the scoring of the new measurement

```bash
% scalasca -examine -s epik_lu_8_sum
INFO: Post-processing runtime summarization report...
/srv/app/scalasca/bin/cube3_score -r ./epik_lu_8_sum/summary.cube.gz > ./epik_lu_8_sum/epik.score
Reading ./epik_lu_8_sum/summary.cube.gz... done.
Estimated aggregate size of event trace (total_tbc): 54560192 bytes
Estimated size of largest process trace (max_tbc): 7582262 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes or reduce requirements using file listing names of USR regions to be filtered.)

INFO: Score report written to ./epik_lu_8_sum/epik.score
LU filtered summary report

- View the score file

```
% less epik_lu_8_sum/epik.score
flt type  max_tbc  time  % region
  ANY  7582262  98.10  100.00 (summary) ALL
  MPI  4574534  18.62  18.98 (summary) MPI
  COM  2259600  52.43  53.45 (summary) COM
  USR  748152  26.93  27.45 (summary) USR

  MPI  2372550  1.68  1.71 MPI_Send
  MPI  2147556  6.22  6.34 MPI_Recv
  COM  1493952  0.45  0.46 exchange_1_
  COM  373488  16.20  16.52 buts_
  USR  373488  12.61  12.85 jacu_
  COM  373488  12.98  13.23 blts_
...
```

- Reduction on the execution time (for bigger sizes of problems the difference is more obvious)
- Small decrease of the MPI and COM timings.
LU trace measurement collection

Execute the application with the “-t” flag

```bash
% scalasca -analyze -t mpirun -np 8 lu.A.8
S=C=A=N: Scalasca 1.4 trace collection and analysis
S=C=A=N: ./epik_lu_8_trace experiment archive
S=C=A=N: Tue Apr 12 00:45:59 2011: Collect start
/srv/app/openmpi/bin//mpirun -np 8 lu.A.8
[00000]EPIK: Created new measurement archive ./epik_lu_8_trace
[00000]EPIK: EPK_FILTER "lu.filtering" filtered 1 of 222 functions
[00000]EPIK: Activated ./epik_lu_8_trace [10000000 bytes] (0.206s)
[ ... output ... ]
[00000]EPIK: Flushed 6057882 bytes to file ./epik_lu_8_trace/ELG/00000
[00000]EPIK: Unifying... done (0.012s)
[00000]EPIK: Collating... done (0.012s)
[00001]EPIK: Flushed 7582272 bytes to file ./epik_lu_8_trace/ELG/00001
[00002]EPIK: Flushed 7582272 bytes to file ./epik_lu_8_trace/ELG/00002
[ ... ]
[00000]EPIK: 1flush=0.006GB@11.549MB/s, Pflush=0.045GB@77.119MB/s
[00000]EPIK: Closed experiment ./epik_lu_8_trace (1.125s)
maxHeap(*)=16.211/125.527MB
S=C=A=N: Tue Apr 12 00:46:18 2011: Collect done (status=0) 19s
S=C=A=N: Tue Apr 12 00:46:18 2011: Analyze start
/srv/app/openmpi/bin//mpirun -np 8 /srv/app/scalasca/bin/scout.mpi . ./epik_lu_8_trace
```

- One file per MPI rank is created in the experiment directory epik_lu_8_trace
LU trace measurement analysis

- Scalasca provides the SCOUT tool, a parallel trace analyzer which analyzes the trace files and produces an analysis report.

```
S=C=A=N: Tue Apr 12 00:46:18 2011: Collect done (status=0) 19s
S=C=A=N: Tue Apr 12 00:46:18 2011: Analyze start
/srv/app/openmpi/bin/mpirun -np 8 /srv/app/scalasca/bin/scout.mpi ./epik_lu_8_trace
SCOUT Copyright (c) 1998-2011 Forschungszentrum Juelich GmbH

Analyzing experiment archive ./epik_lu_8_trace

Reading definitions file ... done (0.003s).
Reading event trace files ... done (0.734s).
Preprocessing ... done (0.058s).
Analyzing trace data ... done (0.753s).
Writing report files ... done (0.026s).

Max. memory usage : 26.234MB
Total processing time : 1.593s
S=C=A=N: Tue Apr 12 00:46:21 2011: Analyze done (status=0) 3s
Warning: 19.605MB of analyzed trace data retained in ./epik_lu_8_trace/ELG!
S=C=A=N: ./epik_lu_8_trace complete.
```

- The maximum amount of memory used by any process is 26.234MB.
LU trace measurement, metric tree, communication

% square epi_lu_8_trace
EPIK user instrumentation API, Fortran

```fortran
#include ``epik_user.inc''

subroutine foo(...)
    declarations
    EPIK_FUNC_REG("foo")
    EPIK_USER_REG(r_name,"iteration loop")
    EPIK_FUNC_START()
    ...
    EPIK_USER_START(r_name)
    do i= 1, 100
        ...
    end do
    EPIK_USER_END(r_name)
    ...
    EPIK_FUNC_END()
end subroutine foo
```
EPIK user instrumentation API, C/C++

```c
#include "epik_user.h"

void foo(...)
{
    /* declarations */
    EPIK_USER_REG(r_name,"iteration loop");
    EPIK_FUNC_START();
    ...
    EPIK_USER_START(r_name);
    for (i = 0; i < 10; ++i)
    {
        ...
    }
    EPIK_USER_END(r_name);
    ...
    EPIK_FUNC_END();
}
```

- In order to compile the source code with EPIK commands we have to use the “-user” flag
  
  `scalasca -instrument -user mpif77`

- We can mark a specific area and observe its performance during the analysis
Automatic instrumentation using PDT

- In order to enable PDT-based source-code instrumentation, the option "-pdt" is required and disable the compiler instrumentation by "-comp=none"

```bash
% scalasca -instrument -pdt -comp=none mpif77 ...
```

- Option for selective instrumentation file

```bash
% scalasca -instrument -pdt -comp=none -optTauSelectFile=lu.pdt mpif77 ...
```

- Note: For Fortran 77 most times is needed to give an extra option

```bash
% scalasca -instrument -pdt -comp=none -optTauSelectFile=lu.pdt mpif77 \ 
-ffixed-line-length-0
```

- Format of the selective instrumentation file
  - Exclude files
    ```
    BEGIN_FILE_EXCLUDE_LIST
    test.c # Excludes file test.c
    foo*.c # Excludes all C files with prefix 'foo'
    END_FILE_EXCLUDE_LIST
    ```
Automatic instrumentation using PDT II

- **Exclude functions**
  
  ```
  BEGIN_EXCLUDE_LIST
  # Exclude C function matmult
  void matmult(Matrix*, Matrix*, Matrix*) C

  # Exclude C++ functions with prefix 'sort_' and a
  # single int pointer argument
  void sort_#(int *)

  # Exclude all void functions in namespace 'foo'
  void foo::#
  END_EXCLUDE_LIST
  ```

- **The mark # is wildcard for a routine name and the mark * is a wildcard character**

- **Include functions for instrumentation**

  ```
  BEGIN_INCLUDE_LIST/END_INCLUDE_LIST
  ```

- **Exclude the function EXACT from the LU benchmark**
  
  ```
  % cat lu.pdt
  BEGIN_EXCLUDE_LIST
  EXACT
  END_EXCLUDE_LIST
  ```
Automatic instrumentation using PDT III

- Declare the appropriate compile command in the config/make.def file
  
  ```
  MPIF77 = scalasca -instrument -pdt -comp=none -optTauSelectFile=/path/lu.pdt \
  mpif77 -ffixed-line-length-0
  ```

- Compile the LU benchmark, for class A and 8 processes from the NPB root path
  
  ```
  % make LU NPROCS=8 CLASS=A
  ```

- Enter the Scalasca folder and execute the benchmark
  
  ```
  % cd bin.scalasca
  % scalasca -analyze mpirun -np 8 lu.A.8
  ```

- Now if you apply the scoring and see the output file
  
  ```
  % scalasca -examine -s epik_lu_8_sum
  % cat epik_lu_8_sum/epik.score | grep EXACT
  ```

Then there is no function EXACT that is traced
LU summary measurement, hardware counters

- Measure the PAPI hardware counters:
  - PAPI_TOT_INS (total instructions completed)
  - PAPI_FP_OPS (floating point operations)
  - PAPI_L2_TCM (L2 cache misses)
  - PAPI_RES_STL (stalled cycles on any resource)

% export EPK_METRICS=PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:PAPI_RES_STL
CUBE3 algebra utilities

- Extract only the SSOR region with its sub-regions

```
% cube3_cut -r 'SSOR' epik_lu_8_sum_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:PAPI_RES_STL/epitome.cube
Reading epik_lu_8_sum_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:PAPI_RES_STL/epitome.cube ... done.
++++++++++++ Cut operation begins +++++++++++++++++++++++++
++++++++++++ Cut operation ends successfully ++++++++++++++++
Writing cut.cube.gz ... done.
```

- View the new CUBE3 file

```
% square cut.cube.gz
```
LU summary measurement, cut for SSOR
LU summary measurement, compare two executions

- Change the name of the experiment archive directory in order to execute again the experiment with different compiler options

```bash
% mv epik_lu_8_sum_PAPI_TOT_INS\:PAPI_FP_OPS\:PAPI_L2_TCM\:PAPI_RES_STL/ \ epik_lu_b_8_o3_sum_PAPI_TOT_INS\:PAPI_FP_OPS\:PAPI_L2_TCM\:PAPI_RES_STL/
```

- Declare the option “-O2” in the file config/make.def

```bash
FFLAGS = -O2
```

- Compile the LU benchmark, class B, 8 processes

```bash
% make clean
% make LU NPROCS=8 CLASS=B
```

- Enter the directory with the executables

```bash
% cd bin.scalasca
```

- Execute the benchmark

```bash
% scalasca -analyze mpirun --bind-to-core -np 8 lu.B.8
```
Compare the two executions

```plaintext
% cube3_diff epik_lu_b_8_o2_sum_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM: \ 
PAPI_RES_STL/epitome.cube epik_lu_8_sum_PAPI_TOT_INS:PAPI_FP_OPS: \ 
PAPI_L2_TCM:PAPI_RES_STL/epitome.cube
Reading epik_lu_b_8_o2_sum_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM: \ 
PAPI_RES_STL/epitome.cube ... done.
Reading epik_lu_8_sum_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM: \ 
PAPI_RES_STL/epitome.cube ... done.
++++++++++++ Diff operation begins ++++++++++++++++++++++++++++++++ INFO::Merging metric dimension... done.
INFO::Merging program dimension... done.
INFO::Merging system dimension... done.
INFO::Mapping severities... done.
INFO::Merging topologies... done.
INFO::Diff operation... done.
++++++++++++ Diff operation ends successfully ++++++++++++++++++++ Writing diff.cube.gz ... done.
```
LU summary measurement, cut for SSOR

- View the new CUBE3 file
  
  ```
  % square diff.cube.gz
  ```

- Not all the parts of the code were improved by the change of the optimization option
CUBE3 utilities

There are more CUBE3 utilities:

- **Difference**
  
  \[ \%\text{cube3\_diff}\ \text{first.cube}\ \text{second.cube}\ -o\ \text{new.cube} \]

- **Merge two different measurements with different metrics**
  
  \[ \%\text{cube3\_merge}\ \text{first.cube}\ \text{second.cube}\ -o\ \text{new.cube} \]

- **Calculate the mean of many measurements**
  
  \[ \%\text{cube3\_mean}\ \text{first.cube}\ \text{second.cube}\ \text{third.cube}\ \text{fourth.cube}\ -o\ \text{new.cube} \]

- **Compare two measurements if they are exactly the same**
  
  \[ \%\text{cube3\_cmp}\ \text{first.cube}\ \text{second.cube}\ \text{third.cube}\ -o\ \text{new.cube} \]

- **Cut, re-root selected sub-trees**
  
  \[ \%\text{cube3\_cut}\ -r\ \text{name\_of\_sub\_tree}\ \text{first.cube}\ -o\ \text{new.cube} \]

- There are more utilities, like \text{cube3\_clean}
LU benchmark, class B

We are going to execute the LU benchmark for class B and various number of processors and observe performance issues.

- Go to the root folder of the serial version of NPB and compile the LU benchmark for class B

```bash
% cd ~/tutorial/NPB3.3-SER
% make clean
% make LU CLASS=B
```

- Go to the executable directory and execute the benchmark

```bash
% cd bin.scalasca
% scalasca -analyze ./lu.B.x
```

- Explore the measurement analysis report

```bash
% square epik_lu_O_trace_PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:\PAPI_RES_STL
```
LU measurement summary for the serial version and class B

The computation execution time is 514.61 seconds and there are 8.15e11 stalled cycles on any resource.
Similar for the MPI version of the LU benchmark, compile it for 2 processors and class B

```
% cd ~/tutorial/NPB3.3-MPI
% make clean
% make LU NPROCS=2 CLASS=B
```

Compile the benchmark also for 4, 8, 16 and 32 processors

Go to the executable directory

```
% cd bin.scalasca
```

Declare the appropriate ELG buffer

```
% echo `export ELG_BUFFER_SIZE=60000000` >> ~/.bashrc
```
Execute the LU benchmark

- Execute the benchmarks
  - `% scalasca -analyze mpirun -np 2 --bind-to-core lu.B.2`
  - `% scalasca -analyze mpirun -np 4 --bind-to-core lu.B.4`
  - `% scalasca -analyze mpirun -np 8 --bind-to-core lu.B.8`
  - `% scalasca -analyze mpirun -np 16 --bind-to-core lu.B.16`
  - `% scalasca -analyze mpirun -np 32 --bind-to-core lu.B.32`

- Let's examine the measurement analysis report for the 2 processors
  - `% square epik_lu_2_sum_PAPI_TOT_INS:PAPI_FP_OPS:\PAPI_L2_TCM:PAPI_RES_STL`
The total computation execution time is 365.14 seconds for all the processors (sum) and the exclusive time for the SSOR function is 9.45 seconds.

The communication time is less than 7 seconds.
LU measurement summary for 2 processors, class B, stalled cycles on any resource

There are 5.3e11 stalled cycles on any resource for all the processors (around to 8.6e9 per processor)
The total computation execution time is 352.89 seconds for all the processors (sum) and the exclusive time for the SSOR function is 9.91 seconds.

The communication time is less than 16 seconds.
LU measurement summary for 4 processors, class B, stalled cycles on any resource

There are 4.95e11 stalled cycles on any resource for all the processors
The total computation execution time is 396.73 seconds for all the processors (sum) and the exclusive time for the SSOR function is 17.62 seconds.

The communication time is less than 40 seconds.
LU measurement summary for 8 processors, class B, stalled cycles on any resource

- There are $5.92 \times 10^{11}$ stalled cycles on any resource for all the processors
The total computation execution time is 486.97 seconds for all the processors (sum) and the exclusive time for the SSOR function is 25.38 seconds.

The communication time is less than 89 seconds.

The value of the total instructions is increased by 28%.
Comparison of the metrics for the Intel processor

- There is no so big difference on the Intel processor and on older AMD Opteron processors (2xx and 2xxx)
- On Intel processor the difference is 11.5%
LU measurement summary for 16 processors, class B, stalled cycles on any resource

- There are 7.85e11 stalled cycles on any resource for all the processors.
  Check the variation per processor on the system tree.
LU measurement summary for 16 processors, class B, stalled cycles on any resource for the region RHS
LU measurement summary for 32 processors, class B, execution time

- The total computation execution time is 528.46 seconds for all the processors (sum) and the exclusive time for the SSOR function is 26.08 seconds.
- The communication time is less than 177 seconds.
LU measurement summary for 32 processors, class B, stalled cycles on any resource

There are $8.10 \times 10^{11}$ stalled cycles on any resource for all the processors. Check the variation per processor on the system tree.
LU measurement summary for 32 processors, class B, stalled cycles on any resource, box plot

- The minimum value is $7.22 \times 10^8$ and the maximum $1.94 \times 10^9$ where the mean is $1.24 \times 10^9$
The minimum value is $9.41 \times 10^9$ and the maximum $1.39 \times 10^{10}$ where the mean is $1.19 \times 10^{10}$
LU measurement summary for 32 processors, class B, computational imbalance, overload, box plot

- Overload means that the execution time is bigger than the average value of all the processes. There is no single participant, so this overload is caused by more than one process.
Underload means that the execution time is less than the average value of all the processes. There is no non-participant, so all the processes execute the underloaded call-path.
We can observe that the underload value for the last process is big enough in comparison with the rest ones.
Overall conclusions from the summary measurements

- The floating operations remain stable but not the total completed instructions. The variation on Intel processor or older AMD Opterons is not so big as on this AMD Opteron.
- The role of the communication is important while we increase the number of the processes.
- The computation time is increasing while we increase the number of the processes.
LU measurement trace for 2 processors, class B, identify communication issues

- The late sender measures the lost time which is caused by a blocking receive operation which is posted earlier than the corresponding send operation.
- The process 0 delays the execution for 0.31 seconds because of the MPI_Wait and the process 1 for 0.99 seconds.
We can observe the number of the visits for each function, for example there are 506 calls to MPI_Wait for the call-path RHS - EXCHANGE_3
LU measurement trace for 4 processors, class B, identify communication issues

- The lost time because of the Late Sender is almost the same as in the previous case
- However we have a case of different sources which its duration is 2.02 seconds
LU measurement trace for 4 processors, class B, identify communication issues, late sender from different sources

In this case or we should reverse the sequence of the MPI_Recv calls in order to avoid this phenomenon or to use the MPI_ANY_SOURCE tag.
The maximum duration of the Late Sender starts to increase (1.57 seconds for process 7)
LU measurement trace for 16 processors, class B, identify communication issues

Similar, the maximum duration of the Late Sender increases (2.29 seconds for process 15)
LU measurement trace for 16 processors, class B, identify communication issues, late sender from different sources

- The maximum delay because of the wrong sequence of the MPI_RECV calls is 2.5 seconds for process 0
LU measurement trace for 32 processors, class B, identify communication issues

While in the previous cases the delay of the MPI_Wait was bigger than the other MPI calls now it is not. It is crucial to study the other call paths as the delay is 29.93 seconds for the MPI_Recv of the EXCHANGE_1 region.
LU measurement trace for 32 processors, class B, identify communication issues, late sender from different sources

For this case the duration of the Late Sender is increased and its proportional to the total communication time is increased
The total delay time caused by the MPI_Allreduce is 1.42 seconds where the minimum delay is 0.02 seconds, the maximum 0.06 seconds and the mean time is 0.04 seconds.

The boxplot provides useful information for a lot of processors with an easy way.
Conclusions

- As we increase the number of the processors that participate to the execution, the Late Sender delay is becoming bigger and should be fixed by applying a better load balancing on the computation part as some processors finish faster than the others.

- Moreover, the delay because of the difference of sources is increasing and the proposed ways to be fixed are by changing the sequence of the MPI_Recv calls or use the MPI_ANY_SOURCE.
TAU Performance System

TAU
TAU Performance System

- Tuning and Analysis Utilities
- Performance profiling and tracing
- Instrumentation, measurement, analysis, visualization
- Performance data management and data mining
- Easy to integrate in application frameworks
TAU Performance System

- TAU is a performance evaluation tool
- Parallel profiling and tracing
- TAU can automatically instrument your source code through PDT for routines, loops, I/O, memory, phases, etc.
- TAU provides various analysis tools
Simplest Case

- **Uninstrumented code:**
  
  % mpirun -np 4 lu.B.4

- **With TAU:**
  
  % mpirun -np 4 tau_exec ./lu.B.4
  % paraprof
How does TAU work?

- **Instrumentation:**
  - Adds probes to perform measurements
  - Source code instrumentation
  - Wrapping external libraries (I/O, CUDA, OpenCL)
  - Rewriting the binary executable

- **Measurement:**
  - Profiling or Tracing
  - Direct instrumentation
  - Indirect instrumentation (sampling)
  - Throttling
  - Per-thread storage of performance data
  - Interface with external packages (PAPI, Scalasca, Score-P, VampirTrace)

- **Analysis:**
  - Visualization of profiles and traces
  - 3D visualization with paraprof, perfexplorer tools
  - Trace conversion tools
Using TAU: Introduction

- TAU supports several measurement and thread option
- Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated during the configuration of the tool
- Instrumenting source code automatically using PDT
  - Choose the appropriate TAU stub makefile
    ```
    % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
    ```
  - Use tau_f90.sh, tau_cxx.sh, tau_cc.sh as F90, C++ and C compilers
    ```
    mpif90 test.f90 -> tau_f90.sh test.f90
    ```
- Set runtime environment variables, execute application and analyze the data
  ```
  % pprof (text based profile display)
  % paraprof (GUI)
  ```
- **Important**: For calling pprof just execute pprof_tau for avoiding conflict with the pprof tool
TAU Instrumentation Approach

- Supports both direct and indirect performance observation
  - Direct instrumentation of program code
  - Instrumentation invokes performance measurement
  - Event measurement
  - Indirect mode: sampling, hardware performance counter overflow

- User-defined events
  - Interval (Start/stop)
  - Atomic, trigger at a single point with data
  - Context events, atomic events with executing context
Direct Observation: Events

- **Event types**
  - **Interval events**
    - Measures exclusive & inclusive duration between events
    - Metrics monotonically increase
  - **Atomic events**
    - Capture performance data state
    - Shows extent variation of triggered values

- **Code events**
  - Routines, classes, templates
  - Statement-level blocks, loops
Interval and Atomic events

**Interval events**

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.658</td>
<td>22.695</td>
<td>1</td>
<td>28</td>
<td>22695820 APPLU</td>
</tr>
<tr>
<td>93.4</td>
<td>660</td>
<td>21.202</td>
<td>2</td>
<td>100689</td>
<td>10601171 APPLU =&gt; SSOR</td>
</tr>
<tr>
<td>93.4</td>
<td>660</td>
<td>21.202</td>
<td>2</td>
<td>100689</td>
<td>10601171 SSOR</td>
</tr>
<tr>
<td>32.0</td>
<td>6.376</td>
<td>7.257</td>
<td>253</td>
<td>506</td>
<td>286866 APPLU =&gt; SSOR =&gt; RHS</td>
</tr>
<tr>
<td>32.0</td>
<td>6.376</td>
<td>7.257</td>
<td>253</td>
<td>506</td>
<td>286866 RHS</td>
</tr>
<tr>
<td>29.6</td>
<td>2.436</td>
<td>6.728</td>
<td>25100</td>
<td>50200</td>
<td>268 APPLU =&gt; SSOR =&gt; BUTS</td>
</tr>
<tr>
<td>29.6</td>
<td>2.436</td>
<td>6.728</td>
<td>25100</td>
<td>50200</td>
<td>268 BUTS</td>
</tr>
<tr>
<td>21.6</td>
<td>693</td>
<td>4.910</td>
<td>100400</td>
<td>100400</td>
<td>49 EXCHANGE_1</td>
</tr>
<tr>
<td>17.3</td>
<td>3.932</td>
<td>3.932</td>
<td>50200</td>
<td>50200</td>
<td>85 APPLU =&gt; SSOR =&gt; BUTS =&gt; EXCHANGE_1 =&gt; MPI_Recv()</td>
</tr>
<tr>
<td>17.3</td>
<td>3.932</td>
<td>3.932</td>
<td>50200</td>
<td>50200</td>
<td>78 APPLU =&gt; SSOR =&gt; BUTS =&gt; EXCHANGE_1 =&gt; MPI_Recv()</td>
</tr>
<tr>
<td>11.5</td>
<td>1.991</td>
<td>2.610</td>
<td>25100</td>
<td>50200</td>
<td>104 APPLU =&gt; SSOR =&gt; BLTS</td>
</tr>
<tr>
<td>8.7</td>
<td>1.978</td>
<td>1.978</td>
<td>25100</td>
<td>50200</td>
<td>79 APPLU =&gt; SSOR =&gt; JACLD</td>
</tr>
<tr>
<td>8.7</td>
<td>1.978</td>
<td>1.978</td>
<td>25100</td>
<td>50200</td>
<td>79 JACLD</td>
</tr>
<tr>
<td>8.3</td>
<td>1.892</td>
<td>1.892</td>
<td>25100</td>
<td>50200</td>
<td>75 APPLU =&gt; SSOR =&gt; JACU</td>
</tr>
</tbody>
</table>

**Atomic events**

**USER EVENTS PROFILE : NODE 0, CONTEXT 0, THREAD 0**

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std Dev</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.03E+05</td>
<td>6.826E-04</td>
<td>6.578E-04</td>
<td>6.826E-04</td>
<td>14.08</td>
<td>Heap Memory Used (KB) at Entry: APPLU</td>
</tr>
<tr>
<td>1</td>
<td>6.578E-04</td>
<td>6.578E-04</td>
<td>6.578E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; ERHS</td>
</tr>
<tr>
<td>2</td>
<td>6.803E-04</td>
<td>6.803E-04</td>
<td>6.803E-04</td>
<td>8.398</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; ERHS =&gt; EXCHANGE_3 =&gt; MPI_Recv()</td>
</tr>
<tr>
<td>2</td>
<td>6.803E-04</td>
<td>6.803E-04</td>
<td>6.803E-04</td>
<td>2.494</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; ERHS =&gt; EXCHANGE_3 =&gt; MPI_Send()</td>
</tr>
<tr>
<td>2</td>
<td>6.805E-04</td>
<td>6.805E-04</td>
<td>6.805E-04</td>
<td>7.338</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; ERHS =&gt; EXCHANGE_3 =&gt; MPI_Wait()</td>
</tr>
<tr>
<td>1</td>
<td>6.823E-04</td>
<td>6.823E-04</td>
<td>6.823E-04</td>
<td>1.242</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; ERROR =&gt; MPI_Allreduce()</td>
</tr>
<tr>
<td>1</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; INIT_COMM =&gt; MPI_Comm_rank()</td>
</tr>
<tr>
<td>1</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; INIT_COMM =&gt; MPI_Comm_size()</td>
</tr>
<tr>
<td>1</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; INIT_COMM =&gt; MPI_Init()</td>
</tr>
<tr>
<td>1</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>6.758E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; INIT_COMM =&gt; NOCEDIM</td>
</tr>
<tr>
<td>1</td>
<td>6.826E-04</td>
<td>6.826E-04</td>
<td>6.826E-04</td>
<td>0</td>
<td>Heap Memory Used (KB) at Entry: APPLU =&gt; MPI_Finalize()</td>
</tr>
</tbody>
</table>
TAU Instrumentation/Measurement

**Instrumentation**
- Source code
- Object code
- Library wrapper
- Binary code
- Virtual machine

**Measurement**

**Event creation and management**
- Event identifier
- Entry/exit events
- Atomic events
- Event mapping
- Event control

**Profiling**
- Statistics
- Atomic profiles
- I/O profiles
- Profile sampling

**Tracing**
- Trace buffering
- Record creation
- Trace I/O
- Timestamp generation
- Trace filtering
- Trace merging

**Performance data sources**
- Timing
- Hardware counters
- System counters
- Kernel

**OS and runtime system modules**
- Threading
- Interrupts
- Runtime system
- I/O
Direct Instrumentation Options in TAU

- Source code Instrumentation
  - Manual instrumentation
  - Automatic instrumentation (PDT)
    - compiler generates instrumented object code
- Library level instrumentation
- Runtime pre-loading and interception of library calls
- Binary code instrumentation
  - Rewrite the binary, runtime instrumentation
Instrument source code using PDT and PAPI

- Choose the appropriate TAU stub makefile

```
% export TAU_MAKEFILE=/srv/app/tau/x86_64/lib/Makefile.
tau-papi-mpi-pdt
% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
```
Time spent in each routine
Generating a flat profile with MPI

- Declare the appropriate environment variables
  
  ```
  % export TAU_MAKEFILE= /srv/app/tau/x86_64/lib/Makefile.tau-papi-mpi-pdt
  ```

- Declare the compiler (config/make.def)
  
  ```
  MPIF77=tau_f90.sh
  ```

- Compile the LU benchmark, class A, 4 processors
  
  ```
  % make clean
  % make LU NPROCS=4 CLASS=A
  ```

- Execute the benchmark
  
  ```
  % cd bin.tau
  % mpirun -np 4 lu.A.4
  ```

- Pack the profile data
  
  ```
  % paraprof --pack app.ppk
  % paraprof app.ppk
  ```

- Click on “node 0”
Automatic Instrumentation

- **Wrapper scripts**
  - Replace F77 (gfortran) with tau_f90.sh
  - Automatically instruments Fortran source code and links with TAU MPI Wrapper library
  - Use tau_cc.sh and tau_cxx.sh for C and C++

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC = mpicc</td>
<td>CC = tau_cc.sh</td>
</tr>
<tr>
<td>CXX = mpicxx</td>
<td>CXX = tau_cxx.sh</td>
</tr>
<tr>
<td>F90 = mpif90</td>
<td>F90 = tau_f90.sh</td>
</tr>
</tbody>
</table>
Instrumentation, re-writing Binaries with MAQAO (beta)

- Instrument:
  
  ```
  % tau_rewrite lu.A.4 -T papi,pdt -o lu.A.4.inst
  ```

- Perform measurement and execute it:
  
  ```
  % mpirun --bind-to-core -np 4 lu.A.4.inst
  ```
Paraprof with binary instrumentation through MAQAO
Paraprof with binary instrumentation through MAQAO

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Exclusive PAPI</th>
<th>Inclusive PAPI</th>
<th>Calls</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce()</td>
<td>0</td>
<td>0</td>
<td>155,322</td>
<td>155,322</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>MPI_BARRIER()</td>
<td>0</td>
<td>0</td>
<td>47,599</td>
<td>47,599</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Bcast()</td>
<td>0.001</td>
<td>0.001</td>
<td>50,145</td>
<td>50,145</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Comm_rank()</td>
<td>0</td>
<td>0</td>
<td>678</td>
<td>678</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Comm_size()</td>
<td>0</td>
<td>0</td>
<td>1,356</td>
<td>1,356</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Finalize()</td>
<td>0.002</td>
<td>0.002</td>
<td>717,712</td>
<td>717,712</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Init()</td>
<td>1.094</td>
<td>1.094</td>
<td>24,499,630</td>
<td>24,499,630</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Irecv()</td>
<td>0.001</td>
<td>0.001</td>
<td>469,645</td>
<td>469,645</td>
<td>512</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>0.37</td>
<td>0.37</td>
<td>275,765,385</td>
<td>275,765,385</td>
<td>31,124</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>0.109</td>
<td>0.109</td>
<td>80,934,587</td>
<td>80,934,587</td>
<td>31,032</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Status()</td>
<td>0.075</td>
<td>0.075</td>
<td>55,444,649</td>
<td>55,444,649</td>
<td>512</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>0</td>
<td>0</td>
<td>15,409</td>
<td>15,409</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bcast_inputs_[{}]</td>
<td>0</td>
<td>0.001</td>
<td>55,634</td>
<td>55,634</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bits_[{}][0,0]</td>
<td>2.339</td>
<td>2.415</td>
<td>7,698,462,532</td>
<td>7,753,627,007</td>
<td>15,562</td>
<td>31,124</td>
</tr>
<tr>
<td>rhs_[{}][0,0]</td>
<td>0.02</td>
<td>0.021</td>
<td>63,913,307</td>
<td>64,696,986</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>exact_[{}][0,0]</td>
<td>0.123</td>
<td>0.123</td>
<td>113,686,083</td>
<td>113,686,083</td>
<td>100,001</td>
<td>0</td>
</tr>
<tr>
<td>exchange_3_[{}][0,0]</td>
<td>0.064</td>
<td>0.174</td>
<td>65,008,391</td>
<td>146,756,002</td>
<td>508</td>
<td>1,524</td>
</tr>
<tr>
<td>exchange_4_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>5,640</td>
<td>10,187</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>init_comm_[{}][0,0]</td>
<td>0.001</td>
<td>1.006</td>
<td>37,007</td>
<td>24,541,386</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>jacth_[{}][0,0]</td>
<td>1.833</td>
<td>1.833</td>
<td>10,940,415,379</td>
<td>10,940,415,379</td>
<td>15,562</td>
<td>0</td>
</tr>
<tr>
<td>main_[{}][0,0]</td>
<td>0.043</td>
<td>14.099</td>
<td>62,062,555</td>
<td>55,164,451,293</td>
<td>1</td>
<td>8,173</td>
</tr>
<tr>
<td>nodedim_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>3,393</td>
<td>3,393</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rhs_[{}][0,0]</td>
<td>3.071</td>
<td>3.243</td>
<td>13,882,831,650</td>
<td>14,028,804,053</td>
<td>253</td>
<td>506</td>
</tr>
<tr>
<td>setiv_[{}][0,0]</td>
<td>0.293</td>
<td>0.403</td>
<td>609,776,974</td>
<td>716,492,794</td>
<td>2</td>
<td>93,857</td>
</tr>
<tr>
<td>ssor_[{}][0,0]</td>
<td>4.66</td>
<td>12.521</td>
<td>22,230,030,540</td>
<td>55,228,850,947</td>
<td>2</td>
<td>62,537</td>
</tr>
<tr>
<td>subdomain_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>866</td>
<td>866</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>timer_clear_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>25,760</td>
<td>25,760</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>timer_read_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>1,610</td>
<td>1,610</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>timer_start_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>5,491</td>
<td>5,491</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>timer_stop_[{}][0,0]</td>
<td>0</td>
<td>0</td>
<td>1,838</td>
<td>1,838</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Hands-on: NPB-MPI / LU

TAU
Declare compiler wrappers

- Enter the hands_on directory

\[
% \text{ cd } ~/\text{tutorial}/\text{NPB3.3-MPI}
\]

- Activate the TAU compiler wrappers

\[
% \text{ vim config/make.def} \\
# \text{MPIF77} = \text{mpif77} \\
\text{MPIF77} = \text{tau_f90.sh}
\]

- Re-compile

\[
% \text{ make clean} \\
% \text{ make LU CLASS=A NPROCS=4}
\]

- Execute the benchmark

\[
% \text{ cd bin.tau} \\
% \text{ mpirun -np 4 lu.A.4} \\
% \text{ paraprof &}
\]
## Compile-Time Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-optVerbose</td>
<td>Turn on verbose debugging messages</td>
</tr>
<tr>
<td>-optCompInst</td>
<td>Use compiler based instrumentation</td>
</tr>
<tr>
<td>-optNoCompInst</td>
<td>Do not revert to compiler instrumentation if source instrumentation fails</td>
</tr>
<tr>
<td>-optTrackIO</td>
<td>Wrap POSIX I/O call and calculates vol/bw of I/O operations</td>
</tr>
<tr>
<td>-optKeepFiles</td>
<td>Does not remove .pdb and .inst.* files</td>
</tr>
<tr>
<td>-optPreProcess</td>
<td>Preprocess Fortran sources before instrumentation</td>
</tr>
<tr>
<td>-optTauSelectFile=&quot;&lt;file&gt;&quot;</td>
<td>Specify selective instrumentation file for tau_instrumentor</td>
</tr>
<tr>
<td>-optTauWrapFile=&quot;&lt;file&gt;&quot;</td>
<td>Specify path to link_options.tau generated by tau_gen_wrapper</td>
</tr>
<tr>
<td>-optHeaderInst</td>
<td>Enable instrumentation of headers</td>
</tr>
<tr>
<td>-optLinking=&quot;&quot;</td>
<td>Options passed to the linker</td>
</tr>
<tr>
<td>-optCompile=&quot;&quot;</td>
<td>Options passed to the compiler</td>
</tr>
<tr>
<td>-optPdtF95Opts=&quot;&quot;</td>
<td>Add options for Fortran parser in PDT</td>
</tr>
<tr>
<td>-optPdtF95Reset=&quot;&quot;</td>
<td>Reset options for Fortran parser in PDT</td>
</tr>
<tr>
<td>-optPdtCxxOptions=&quot;&quot;</td>
<td>Options for C++ parser in PDT</td>
</tr>
<tr>
<td>-optPdtCOpts=&quot;&quot;</td>
<td>Options for C parser in PDT</td>
</tr>
</tbody>
</table>
Compiling Fortran Codes with TAU

- For using free format in .f files, use:
  
  ```
  % export TAU_OPTIONS=''-optPdtF95Opts=''-R free'''
  ```

- Use compiler based instrumentation instead of PDT:
  
  ```
  % export TAU_OPTIONS=''-optCompInst''
  ```

- Use C preprocessor directives in Fortran code:
  
  ```
  % export TAU_OPTIONS=''-optPreProcess -optDetectMemoryLeaks''
  ```

- Use an instrumentation specification file:
  
  ```
  % export TAU_OPTIONS=''-optTauSelectFile=select.tau''
  ```
## Runtime Environment Variables in TAU

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACE</td>
<td>0</td>
<td>Setting to 1 turns on tracing</td>
</tr>
<tr>
<td>TAU_CALLPATH</td>
<td>0</td>
<td>Setting to 1 turns on callpath profiling</td>
</tr>
<tr>
<td>TAU_TRACK_MEMORY_LEAKS</td>
<td>0</td>
<td>Setting to 1 turns on leak detection</td>
</tr>
<tr>
<td>TAU_TRACK_HEAP</td>
<td>0</td>
<td>Setting to 1 turns on heap memory/headroom at routine entry &amp; exit</td>
</tr>
<tr>
<td>TAU_CALLPATHDEPTH</td>
<td>2</td>
<td>Specifies depth of callpath</td>
</tr>
<tr>
<td>TAU_TRACK_IO_PARAMS</td>
<td>0</td>
<td>Setting to 1 with -optTrackIO</td>
</tr>
<tr>
<td>TAU_SAMPLING</td>
<td>1</td>
<td>Generates sample based profiles</td>
</tr>
<tr>
<td>TAU_COMM_MATRIX</td>
<td>0</td>
<td>Setting to 1 generates communication matrix display using context events</td>
</tr>
<tr>
<td>TAU_THROTTLE</td>
<td>1</td>
<td>Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently</td>
</tr>
<tr>
<td>TAU_THROTTLE_NUMCALLS</td>
<td>100000</td>
<td>Specifies the number of calls before testing for throttling</td>
</tr>
<tr>
<td>TAU_THROTTLE_PERCALL</td>
<td>10</td>
<td>Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time</td>
</tr>
<tr>
<td>TAU_COMPENSATE</td>
<td>0</td>
<td>Setting to 1 enables runtime compensation of instrumentation overhead</td>
</tr>
<tr>
<td>TAU_PROFILE_FORMAT</td>
<td>Profile</td>
<td>Setting to “merged” generates a single file, “snapshot” generates snapshot per thread</td>
</tr>
<tr>
<td>TAU_METRICS</td>
<td>TIME</td>
<td>Setting to a comma separated list (TIME:PAPI_TOT_INS:PAPI_FP_OPS)</td>
</tr>
</tbody>
</table>
Loop level profile

- Declare the options for TAU

```bash
% export TAU_PROFILE=1
% export TAU_PROFILE_FORMAT=Profile
% export TAU_OPTIONS='--optTaufSelectFile=select.tau'

% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine='''#'''
END_INSTRUMENT_SECTION
```

- Compile the benchmark

```bash
% make clean
% make LU NPROCS=4 CLASS=A
```

- Execute the benchmark

```bash
% cd bin.tau
% mpirun -np 4 lu.A.4
```

- Analyze the profiling data

```bash
% paraprof --pack lu_a_4.ppk
% paraprof lu_a_4.ppk
```
LU benchmark, loop profile
PAPI profile with 2 or more metrics

- Declare the environment variable TAU_METRICS
  ```
  % export TAU_METRICS=TIME:PAPI_FP_OPS:PAPI_TOT_INS
  ```

- Execute the benchmark
  ```
  % mpirun -np 4 lu.A.4
  ```

- Analyze the profiling data
  ```
  % paraprof --pack lu_a_4_papi.ppk
  % paraprof lu_a_4_papi.ppk
  ```

- Click Options -> Show Derived Metric Panel -> click PAPI_TOT_INS, click "/", click TIME, Apply, choose the new metric by double clicking
LU benchmark, loop profile, instructions per second
Enable the Callpath Profile

% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=10

Execute the benchmark

% mpirun -np 4 lu.A.4

Analyze the profiling data

% paraprof --pack lu_a_4_papi_callpath.ppk
% paraprof lu_a_4_papi_callpath.ppk
Call graph

- From the ParaProf window, Click Windows -> Click Thread -> Click Call Graph, select for which process you want to see the call graph.
Enable the communication matrix

% export TAU_COMM_MATRIX=1

Execute the benchmark

% mpirun -np 4 lu.A.4

Analyze the profiling data

% paraprof --pack lu_a_4_papi_comm.ppk
% paraprof lu_a_4_papi_comm.ppk

Click Windows -> Click 3D Communication Matrix
Communication Matrix Display, exclusive time
Inspect the duration of the MPI calls
Study the total instructions per function
Trace the LU benchmark and prepare them for the JumpShot

- **Enable the tracing feature**
  
  ```
  % export TAU_TRACE=1
  ```

- **Execute the benchmark**
  
  ```
  % mpirun -np 4 lu.A.4
  ```

- **Merge the tracefiles**
  
  ```
  % tau_treemerge.pl
  ```

- **Convert the traces to SLOG2 format**
  
  ```
  % tau2log2 tau.trc tau.edf -o app.slog2
  % jumpshot app.slog2
  ```

- **The following example is for the LU benchmark, class B and 8 processes**
View traces from the Jumpshot tool
View traces from the Jumpshot tool
View traces from the Jumpshot tool
View traces from the Jumpshot tool
View traces from the Jumpshot tool
Connection between various tools
TAU Analysis
Prepare and execute the experiments

% export TAU_METRICS=TIME:PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:\
PAPI_RES_STL
% export TAU_CALLPATH=0
% export TAU_PROFILE_FORMAT=profile

- Compile the LU benchmark for classes A,B and 2-32 processes

  % make clean; make LU NPROCS=2 CLASS=A
  % make clean; make LU NPROCS=4 CLASS=A
  ...
  % make clean; make LU NPROCS=2 CLASS=B
  % make clean; make LU NPROCS=4 CLASS=B
  ...

- Execute the experiments for class A and 4,8,16,32 processes (example for class A and 4-8 processes)

  % cd bin.tau
  % rm -r MULTI*  // if there are data from previous experiments
  % mpirun -np 4 --bind-to-core lu.A.4
  % paraprof --pack lu_a_4.ppk
  % rm -r MULTI*
  % mpirun -np 8 --bind-to-core lu.A.8
  % paraprof --pack lu_a_8.ppk
  % rm -r MULTI*
% paraprof lu_a_4.ppk

- Click Options -> Uncheck Stack Bars Together
Paraprof

File Options Windows Help

Name: RHS \{\{f.s\} \{5,7\}:\{511,9\}\}  
Metric Name: TIME  
Value: Exclusive  
Units: seconds

<table>
<thead>
<tr>
<th>Node</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 0</td>
<td>5.442</td>
</tr>
<tr>
<td>node 1</td>
<td>5.479</td>
</tr>
<tr>
<td>node 2</td>
<td>5.625</td>
</tr>
<tr>
<td>mean</td>
<td>5.414</td>
</tr>
<tr>
<td>node 3</td>
<td>5.112</td>
</tr>
<tr>
<td>std. dev.</td>
<td>0.186</td>
</tr>
</tbody>
</table>
**Paraprof**

- Click Windows -> Click Thread -> Click User Events Statistics -> Select a thread

<table>
<thead>
<tr>
<th>Total</th>
<th>NumSamples</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2163E0</td>
<td>3.036</td>
<td>163040</td>
<td>512</td>
<td>3860.804</td>
<td>20436.392</td>
<td>Message size received from all nodes</td>
</tr>
<tr>
<td>1.2182E0</td>
<td>3.032</td>
<td>163040</td>
<td>512</td>
<td>3851.305</td>
<td>20436.65</td>
<td>Message size sent to all nodes</td>
</tr>
<tr>
<td>8.3233E7</td>
<td>512</td>
<td>163040</td>
<td>512</td>
<td>1.62564.092</td>
<td>1.4379.085</td>
<td>Message size received in wait</td>
</tr>
<tr>
<td>8.2503E7</td>
<td>506</td>
<td>163040</td>
<td>163040</td>
<td>1.6364</td>
<td>0</td>
<td>Message size for all-reduce</td>
</tr>
<tr>
<td>260</td>
<td>16</td>
<td>49</td>
<td>8</td>
<td>24</td>
<td>16</td>
<td>Message size for broadcast</td>
</tr>
<tr>
<td>84</td>
<td>16</td>
<td>49</td>
<td>4</td>
<td>8.4</td>
<td>10.83</td>
<td>Message size for broadcast</td>
</tr>
<tr>
<td>327680</td>
<td>2</td>
<td>163840</td>
<td>163840</td>
<td>163840</td>
<td>0</td>
<td>Message size received in wait : APPLU [2]</td>
</tr>
<tr>
<td>1.256</td>
<td>2</td>
<td>544</td>
<td>512</td>
<td>528</td>
<td>16</td>
<td>Message size received in wait : APPLU [2]</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>0</td>
<td>Message size received in wait : APPLU [2]</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>0</td>
<td>Message size received in wait : APPLU [2]</td>
</tr>
</tbody>
</table>
Compare different executions per function for LU benchmark, class A

- Add all the data to paraprof
  Click File -> Click Open... -> Click Select File(s) and select your file
- Repeat the previous procedure for all the experiments for classes A and B
- Select the name of the first experiment for class A, do right click on it and select the option “Add Mean to Comparison Window”. A new window pops up, do not close it
- Repeat the procedure for all the experiments of class A
Compare the duration of the functions while we increase the number of the processes.

While we double the number of the processes the duration of the RHS function is decreased by around to 35%.
Compare the duration of the functions, studying the floating operations

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_FP_OPS

The value of the floating operations do not justify the execution time of the function RHS
Compare the duration of the functions, studying L2 cache misses

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_L2_TCM

Neither the L2 cache misses justify the execution time of the function RHS
Compare the duration of the functions, studying the total instructions

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_TOT_INS

The value of the total completed instructions can justify a part of the mentioned difference
Compare the duration of the functions, studying the stalled cycles on any resource

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_RES_STL

Moreover the stalled cycles did not decrease as expected, thus the difference can be caused by this reason also.
Compare different executions per function for LU benchmark, class B

- Select the name of the first experiment for class B, do right click on it and select the option “Add Mean to Comparison Window”. A new window pops up, do not close it
- Repeat the procedure for all the experiments of class B
Compare the duration of the functions while we increase the number of the processes

While we double the number of the processes the duration of the RHS function is decreased by around to 36%
Compare the duration of the functions, studying the floating operations

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_FP_OPS

The value of the floating operations do not justify the execution time of the function RHS
Compare the duration of the functions, studying L2 cache misses

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_L2_TCM

Neither the L2 cache misses justify the execution time of the function RHS
Compare the duration of the functions, studying the total instructions

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_TOT_INS

The value of the total completed instructions can justify a part of the mentioned difference
Compare the duration of the functions, studying the stalled cycles on any resource

- Click Options -> Click Select Metric -> Click Select Exclusive -> Click PAPI_RES_STL

Moreover the stalled cycles did not decrease as expected, thus the difference can be caused by this reason also.
Add the experiments on the default database

```bash
% perfdfm_loadtrial -a sc_lu_a -x scaletest -n 4 lu_a_4.ppk
% perfdfm_loadtrial -a sc_lu_a -x scaletest -n 8 lu_a_8.ppk
% perfdfm_loadtrial -a sc_lu_a -x scaletest -n 16 lu_a_16.ppk
% perfdfm_loadtrial -a sc_lu_a -x scaletest -n 32 lu_a_32.ppk
% perfdfm_loadtrial -a sc_lu_b -x scaletest -n 4 lu_b_4.ppk
% perfdfm_loadtrial -a sc_lu_b -x scaletest -n 8 lu_b_8.ppk
% perfdfm_loadtrial -a sc_lu_b -x scaletest -n 16 lu_b_16.ppk
% perfdfm_loadtrial -a sc_lu_b -x scaletest -n 32 lu_b_32.ppk
```
PerfExplorer, Total Execution Time for class A

- Expand the database perfdmf -> Expand the Application name sc_lu_a -> Select the experiment name scaletest
- Click Charts -> Click Total Execution Time -> Select the metric TIME ->

![Graph showing total execution time vs. number of processors]

Click OK
- Expand the database perfmdm -> Expand the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric TIME ->

Click OK
PerfExplorer, Stacked Bar Chart for class A and PAPI_FP_OPS

- Expand the database perfdmf -> Expand the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric PAPI_FP_OPS -> Click OK
PerfExplorer, Stacked Bar Chart for class A and PAPI_TOT_INS

- Expand the database perfdfm -> Explain the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric PAPI_TOT_INS -> Click OK
PerfExplorer, Relative Efficiency for class A

- Expand the database perfdfm -> Expand the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Relative Efficiency -> Select the “The problem size remains constant“ -> Click OK
PerfExplorer, Relative Efficiency for class A per event

- Expand the database perfdmf -> Expand the Application name sc.lu.a
  -> Select the experiment name scaletest

- Click Charts -> Click Relative Efficiency by event -> Select the metric TIME -> Click OK -> Select the “The problem size remains constant“->

Click OK
PerfExplorer, Relative Speedup for class A

- Expand the database perfdfm -> Expand the Application name sc_lu_a
- Select the experiment name scaletest

- Click Charts -> Click Relative Speedup -> Select the metric TIME -> Click OK
- Select the “The problem size remains constant“-> Click OK

![Relative Speedup - sc_lu_a:scaletest:TIME](image)
- Expand the database perfdfmf -> Explain the Application name sc_lu_a
  -> Select the experiment name scaletest

- Click Charts -> Click Relative Speedup -> Select the metric TIME -> Click OK
  -> Select the “The problem size remains constant“-> Click OK
PerfExplorer, MPI Time for class A

- Expand the database perfdfm -> Expand the Application name sc_lu_a -> Select the experiment name scaletest
- Click Charts -> Click Group % of Total Runtime -> Select the metric TIME -> Click OK -> Select MPI group -> Click OK

![Graph showing MPI Time / Total Time for sc_lu_a:scaletest](image-url)
PerfExplorer, Runtime Breakdown for class A

- Expand the database perfdmf -> Explain the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Runtime Breakdown -> Select the metric TIME ->

Click OK
PerfExplorer, Runtime Breakdown PAPI_TOT_INS, for class A

- Expand the database perfdmf -> Expand the Application name sc_lu_a -> Select the experiment name scalestest
- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_TOT_INS -> Click OK
PerfExplorer, Runtime Breakdown PAPI_L2_TCM, for class A

- Expand the database perfdfm -> Expand the Application name sc_lu_a -> Select the experiment name scaletest
- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_L2_TCM -> Click OK
PerfExplorer, Runtime Breakdown PAPI_RES_STL, for class A

- Expand the database perfdfm -> Expand the Application name sc_lu_a
  -> Select the experiment name scaletest
- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_RES_STL -> Click OK
PerfExplorer, Total Execution Time for class B

- Expand the database perfdfm -> Explain the Application name sc_lu_b
  -> Select the experiment name scaletest

- Click Charts -> Click Total Execution Time -> Select the metric TIME ->

![Graph showing Total Execution Time vs Number of Processors](image_url)

Click OK
PerfExplorer, Stacked Bar Chart for class B

- Expand the database perfdfm -> Expand the Application name sc_lu_b
  - Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric TIME

Click OK
PerfExplorer, Stacked Bar Chart for class B and PAPI_FP_OPS

- Expand the database perfdmf -> Explain the Application name sc_lu_b
  -> Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric PAPI_FP_OPS -> Click OK
PerfExplorer, Stacked Bar Chart for class B and PAPI_TOT_INS

- Expand the database perfdf -> Expand the Application name sc_lu_b -> Select the experiment name scaletest
- Click Charts -> Click Stacked Bar Chart -> Select the metric PAPI_TOT_INS -> Click OK
Expand the database perfdfm -> Expand the Application name sc_lu_b -> Select the experiment name scaletest

Click Charts -> Click Relative Efficiency -> Select the “The problem size remains constant” -> Click OK
PerfExplorer, Relative Efficiency for class B per event

- Expand the database perfdfm -> Expand the Application name sc_lu_b
  -> Select the experiment name scaletest

- Click Charts -> Click Relative Efficiency by event -> Select the metric TIME -> Click OK -> Select the ”The problem size remains constant“->

Click OK
PerfExplorer, Relative Speedup for class B

- Expand the database perfdfm -> Explain the Application name sc.lu.b
- Select the experiment name scaletest
- Click Charts -> Click Relative Speedup -> Select the metric TIME -> Click OK -> Select the “The problem size remains constant“ -> Click OK
PerfExplorer, Relative Speedup by event for class B

- Expand the database perfdmf -> Expand the Application name sc_lu_b
  -> Select the experiment name scaletest
- Click Charts -> Click Relative Speedup -> Select the metric TIME -> Click
  OK -> Select the “The problem size remains constant“-> Click OK
PerfExplorer, MPI Time for class B

- Expand the database perfdfm -> Expand the Application name sc.lu.b
  -> Select the experiment name scaletest
- Click Charts -> Click Group % of Total Runtime -> Select the metric TIME
  -> Click OK -> Select MPI group -> Click OK
PerfExplorer, Runtime Breakdown for class B

- Expand the database perfdfm -> Expand the Application name sc_lu_b
- Select the experiment name scalestest
- Click Charts -> Click Runtime Breakdown -> Select the metric TIME ->

Click OK
PerfExplorer, Runtime Breakdown PAPI_TOT_INS, for class B

- Expand the database perfdfm -> Expand the Application name sc_lu_b
  -> Select the experiment name scaletest

- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_TOT_INS -> Click OK
PerfExplorer, Runtime Breakdown PAPI_L2_TCM, for class B

- Expand the database perfdfm -> Expand the Application name sc_lu_b
  -> Select the experiment name scaletest

- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_L2_TCM -> Click OK
PerfExplorer, Runtime Breakdown PAPI_RES_STL, for class B

- Expand the database perfdfm -> Expand the Application name sc_lu_b
  -> Select the experiment name scaletest

- Click Charts -> Click Runtime Breakdown -> Select the metric PAPI_RES_STL -> Click OK
Apply dynamic phases on SSOR

- Declare the appropriate TAU stub Makefile

```bash
% vim ~/.bashrc
% export TAU_MAKEFILE=/srv/app/tau/x86_64/lib/Makefile.tau
   -phase-papi-mpi-pdt
```

- Declare where the phase starts and ends. The phase starts at the line 83 of the file `ssor.f` and ends at line 202. Moreover it is called "iteration"

```bash
% vim dyn_phase.pdt
BEGIN_INSTRUMENT_SECTION
dynamic phase name="iteration" file="ssor.f" line=83 to line=202
END_INSTRUMENT_SECTION
```

- Declare the appropriate options

```bash
% vim ~/.bashrc
export TAU_OPTIONS=-optPDTInst -optTauSelectFile=/path/ \
   dyn_phase.pdt'
```

- Compile the instances for 4-32 processes from the roof folder of NAS

```bash
% make clean; make LU NPROCS=4 CLASS=A
% make clean; make LU NPROCS=8 CLASS=A
...
% make clean; make LU NPROCS=4 CLASS=B
% make clean; make LU NPROCS=8 CLASS=B
...
```
Apply dynamic phases on SSOR

- Execute the benchmarks and pack the performance data
  ```
  % cd bin.tau
  % rm -r MULTI*
  % mpirun --bind-to-core -np 4 lu.A.4
  % paraprof --pack lu_a_4_phases.ppk
  % rm -r MULTI*
  % mpirun --bind-to-core -np 8 lu.A.8
  % paraprof --pack lu_a_8_phases.ppk
  ...
  ```

- View the data with the Paraprof tool
  ```
  % paraprof lu_a_4_phases.ppk
  ```
Paraprof and dynamic phases for the LU benchmark, class B, 32 processes
Create an expression of a new metric

- Close the Paraprof sub-window (the one on top of the Paraprof Manager)
- From the Paraprof Manager, Click Options -> Click Show Derived Metric Panel
- Expand the lu_a_4_phases experiment (example in the screenshot for the lu_b_32_phases)
- Select the PAPI_TOT_INS metric, after click the symvol '/' from the Derived Metric Panel and select the metric PAPI_TOT_CYC
- Click Apply

```
Expression: "PAPI_TOT_INS"/"PAPI_TOT_CYC"
```
Profile of a phase

- Double click on the metric TIME of the lu_a_4_phases.ppk
- Right click on any iteration (small continuous areas) and select Open Profile for this Phase

- We chose randomly the 38th iteration
Study the phase

- Select the first group of bar charts, and you can see the following

We can observe that for the 38th iteration the duration of the function RHS varies from 0.021 to 0.028 seconds (33%)
Study the instructions of a specific iteration

- Click Options -> Select Metric -> Exclusive -> PAPI_TOT_INS

- We can observe that for the 38th iteration the value of the total completed instructions for the function RHS varies from $2.7475 \times 10^7$ to $4.7341 \times 10^7$ (72.3%)
Study the floating operations of a specific iteration

- Click Options -> Select Metric -> Exclusive -> PAPI_FP_OPS

We can observe that for the 38th iteration the value of the total completed instructions for the function RHS varies from $1.16E7$ to $1.49E7$ (28.4%).

This indicates a computational imbalance at least for this iteration.
Study the cycles of a specific iteration

- Click Options -> Select Metric -> Exclusive -> PAPI_TOT_CYC

We can observe that for the 38th iteration the value of the cycles for the function RHS varies from $4.5E7$ to $6.746E7$ (49.9%)
Study the instructions per cycle of a specific iteration

- Click Options -> Select Metric -> Exclusive -> (PAPI_TOT_INS / PAPI_TOT_CYC)

We can observe that for the 38th iteration the value of the IPC for the function RHS varies from 0.56 to 0.758 (35.3%)
Study the instructions per second of a specific iteration

- Click Options -> Select Metric -> Exclusive -> (PAPI_TOT_INS / TIME)

We can observe that for the 38th iteration the value of the IPC for the function RHS varies from 1.3E9 to 1.73E9 (33%)
Study the stalled cycles of a specific iteration

- Click Options -> Select Metric -> Exclusive -> PAPI_RES_STL

We can observe that for the 38th iteration the value of the cycles for the function RHS varies from 3.58E7 to 5.59E7 (56%)
Compare specific iterations

- Double click the metric PAPI_TOT_INS from the Paraprof Manager window
- On the new window right click for example on node 3 and select ”Show Thread Statistics Table“

Now we can choose an iteration and expand it

- Sort by the ”Inclusive PAPI_TOT_INS“ by clicking on the head of the column. The minimum value is 1.32E8 and the maximum 2.45E8 (85.6%) so there is computational imbalance
Which functions cause the previous difference

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive PAPI TOT INS</th>
<th>Inclusive PAPI TOT INS Δ</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration [12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Irecv()</td>
<td>3,175,674</td>
<td>132,566,014</td>
<td>1</td>
<td>401</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>15.275</td>
<td>15.275</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Exchange_3 [{exchange_3.f} {5,7}-{312,9}]</td>
<td>328,608</td>
<td>328,608</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Exchange_1 [{exchange_1.f} {5,7}-{177,9}]</td>
<td>409,009</td>
<td>1,275,856</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>2,653,081</td>
<td>2,653,081</td>
<td>303</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>8,249,127</td>
<td>8,249,127</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>Exchange_1 [{exchange_1.f} {5,7}-{177,9}]</td>
<td>3,576,372</td>
<td>13,955,616</td>
<td>400</td>
<td>600</td>
</tr>
<tr>
<td>JACLD [{jacld.f} {5,7}-{387,9}]</td>
<td>18,693,737</td>
<td>18,693,737</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>JACU [{jacu.f} {5,7}-{387,9}]</td>
<td>24,301,553</td>
<td>24,301,553</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>BLTS [{blts.f} {4,7}-{267,9}]</td>
<td>18,179,002</td>
<td>27,095,361</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>BUTS [{buts.f} {4,7}-{267,9}]</td>
<td>22,132,139</td>
<td>27,171,396</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>RHS [{rhs.f} {5,7}-{511,9}]</td>
<td>30,852,437</td>
<td>32,128,293</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Iteration [134]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Irecv()</td>
<td>6,169,579</td>
<td>245,612,510</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>79,483</td>
<td>79,483</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>1,253,398</td>
<td>1,253,398</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>9,928,747</td>
<td>9,928,747</td>
<td>303</td>
<td>0</td>
</tr>
<tr>
<td>Exchange_3 [{exchange_3.f} {5,7}-{312,9}]</td>
<td>2,907,553</td>
<td>11,149,897</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>JACLD [{jacld.f} {5,7}-{387,9}]</td>
<td>17,778,250</td>
<td>17,778,250</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>JACU [{jacu.f} {5,7}-{387,9}]</td>
<td>18,501,008</td>
<td>18,501,008</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>BLTS [{blts.f} {4,7}-{267,9}]</td>
<td>18,425,833</td>
<td>28,253,445</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>RHS [{rhs.f} {5,7}-{511,9}]</td>
<td>61,709,191</td>
<td>72,859,088</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>80,224,087</td>
<td>80,224,087</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>Exchange_1 [{exchange_1.f} {5,7}-{177,9}]</td>
<td>5,793,175</td>
<td>89,036,546</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>BUTS [{buts.f} {4,7}-{267,9}]</td>
<td>22,842,206</td>
<td>102,051,140</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
Conclusions

- In general be sure that you are trusting the hardware
- Be careful about your measurements. Identify any strange result that is obvious
- Plotting the characteristics of a function can be different related to each loop
- Create multiple dynamic phases for identifying strange behavior on iterative procedures (be careful about the overhead)
- The metric of the stalled cycles on any resource is a good one for investigating if there is any overhead
PerfExpert tool

- Not only measures but also analyses performance
  - Tell us where the slow code sections are as well why they perform poorly
  - Suggests source-code changes (unfortunately only for icc compiler for now)
  - Simple to use
PerfExpert tool

- Identification of potential causes for slow speed
  - We can find a lot of information through various tools
- How can we decide if a value is big or not?
  - There are 25,578,391 L2 cache misses in a loop, is it good?
  - How can we reduce it?
PerfExpert tool

- It uses the HPCToolkit
- It executes the application many times for measuring various metrics
- In every execution the total completed instructions are measured in order to be able to compare the different execution in the case of any variation
- It identifies and characterizes the causes of each bottleneck in each code segment
- Local Cycles Per Instruction (LCPI) introduced
Typical optimization workflow with profiling tools

[mostly manual]

- Selecting performance counters
- Running multiple measurements
- Collecting performance data
- Identifying bottlenecks
- Searching for proper optimization method
- Implementing optimization

Optimization workflow with PerfExpert

[mostly automated]

- Fully automatic (for core, chip, & node-level bottlenecks)
  performance counter selection, measurement execution,
  data collection, bottleneck analysis, and optimization suggestion

- Selecting and implementing optimization
PerfExpert tool

- During the installation, PerfExpert measures various architecture parameters, L1 data access latency etc.
- The LCPI values are a combination of PAPI metrics and architecture parameters
Local Cycles Per Instruction

- Data Accesses, L1 data hits
  \[
  \frac{(\text{PAPI\textunderscore LD\textunderscore INS} \times \text{L1\_dlat})}{\text{PAPI\textunderscore TOT\textunderscore INS}}
  \]

- Data Accesses, L2 data misses
  \[
  \frac{(\text{PAPI\textunderscore L2\textunderscore TCM} - \text{PAPI\textunderscore L2\textunderscore ICM}) \times \text{mem\_lat}}{\text{PAPI\textunderscore TOT\textunderscore INS}}
  \]

- Instruction Accesses, L2 instruction misses
  \[
  \frac{\text{PAPI\textunderscore L2\textunderscore ICM} \times \text{mem\_lat}}{\text{PAPI\textunderscore TOT\textunderscore INS}}
  \]
PerfExpert tool

- Easy to use, no need to re-compile
- Compile your application as you already do
- Execute the LU benchmark with PerfExpert

```bash
% mpirun --bind-to-core -np 4 perfexpert_run_exp ./lu.A.4
```

- An XML file (experiment.xml) is created and we can see the output of the functions which consume at least the 10% of the total execution with the following command

```bash
perfexpert 0.1 experiment.xml
```
### Output

Function rhs() (19.4% of the total runtime)

<table>
<thead>
<tr>
<th>ratio to total instrns</th>
<th>%</th>
<th>0.........25.........50.........75.........100</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating point</td>
<td></td>
<td>50 ***********************************************</td>
</tr>
<tr>
<td>data accesses</td>
<td></td>
<td>40 ***********************************************</td>
</tr>
<tr>
<td>GFLOPS (% max)</td>
<td></td>
<td>13 *****</td>
</tr>
</tbody>
</table>

Performance assessment: LCPI good.......okay......fair......poor......bad....

* overall: 0.9 >>>>>>>>>>>>>>>>>>>>

Upper bound estimates:

* data accesses: 1.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - L1d hits: 0.4 >>>>>>>>
  - L2d hits: 0.4 >>>>>>>>
  - L2d misses: 0.4 >>>>>>>>
* instruction accesses: 1.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - L1i hits: 0.2 >>>>
  - L2i hits: 0.0 >
  - L2i misses: 1.0 >>>>>>>>>>>>>>>>>>>>>>>>
* data TLB: 0.0 >
* instruction TLB: 0.0 >
* branch instructions: 0.1 >
  - correctly predicted: 0.0 >
  - mispredicted: 0.0 >
* floating-point instr: 1.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - fast FP instr: 1.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - slow FP instr: 0.4 >>>>>>>>>>

229/253
Explanation

- In general if the LCPI value is less than 0.5 then it is considered as a good value.
- Compared to the value of the total completed instructions there are 50% floating point operations and 40% data accesses.
- The value GFLOPS represent the percentage of the maximum possible GFLOP value for the specific machine.
- The overall performance is the cycles per instruction which is not very good in this example.
- It seems that the overhead is distributed on L1 data hits, L2 data hits and L2 cache misses but is not too much each one.
- However the L2 instructions misses cost is really big.
- The fast FP instr includes the floating point multiply and add instructions.
- The slow FP instr includes the floating point divide instructions.

Now we know where we should look for identifying the reason of the overhead.
Compare two executions

- Rename your previous experiment file
  ```bash
  % mv experiment.xml perf_lu_a_4.xml
  ```

- Execute the LU benchmark for class A and 8 processes
  ```bash
  % mpirun --bind-to-core -np 8 perfexpert_run_exp ./lu.A.8
  ```

- Compare your data
  ```bash
  % perfexpert 0.1 perf_lu_a_4.xml experiment.xml
  ```
The value 1 or 2 at the end of the evaluation means which application has bigger value on this metric.
AutoSCOPE

- **Status**
  - Know that there is a performance problem
  - Know why it performs poorly
  - Do not know how to improve the performance

- **AutoSCOPE**
  - Suggests remedies based on analysis results
    - Including code examples and compiler flags
    - For the moment only for Intel compiler (soon for gcc?)
Use AutoSCOPE

- Save the output of the `perfexpert` call

```bash
% perfexpert 0.1 perf_lu_a_4.xml > output_lu_a_4
```

- Call the autoscope

```bash
% autoscope output_lu_a_4
```

Function `rhs_()` (19.4% of the total runtime)

* eliminate floating-point operations through distributivity
  - example: \( d[i] = a[i] * b[i] + a[i] * c[i]; \) ->
    \[ d[i] = a[i] * (b[i] + c[i]); \]

* eliminate floating-point operations through associativity
  - example: \( d[i] = (a[i] * b[i]) * c[i]; \) \( y[i] = (x[i] * a[i]) * b[i]; \) ->
    \[ temp = a[i] * b[i]; \] \( d[i] = temp * c[i]; \) \( y[i] = x[i] * temp; \]

* use trace scheduling to reduce the branch taken frequency
  - example: if (likely_condition) \( f(); \) else \( g(); \) \( h(); \) ->
    \[ void s() \{ g(); h(); \} ... if (!likely_condition) \{ s(); \} \] \( f(); \) \( h(); \)
* factor out common code into subroutines
   - example: ... same_code ... same_code ... ->
     void f() {same_code;} ... f() ... f() ...;

* allow inlining only for subroutines with one call site or very short bodies
  - compiler flag: use the "-nolib-inline", "-fno-inline",
    "-fno-inline-functions", or "-finline-limit=" (with a small ) compiler flags

* make subroutines more general and use them more
  - example: void f() {statements1; same_code;}
    void g() {statements2; same_code;} ->
    void fg(int flag) {if (flag) {statements1;} else {statements2;}
    same_code;}

* split off cold code into separate subroutines and place them at the end of the source file
  - example: if (unlikely_condition) {lots_of_code} ->
    void f() {lots_of_code} ... if (unlikely_condition) f();

* reduce the code size
  - compiler flag: use the "-Os" or "-O1" compiler flag
AutoSCOPE for the loop of RHS function

Loop in function rhs_() (19.4% of the total runtime)
======================================================================
* move loop invariant computations out of loop
  - example: loop i {x = x + a * b * c[i];} ->
    temp = a * b; loop i {x = x + temp * c[i];}

* lower the loop unroll factor
  - example: loop i step 4 {code_i; code_i+1; code_i+2; code_i+3;} ->
    loop i step 2 {code_i; code_i+1;}
  - compiler flag: use the "-no-unroll-aggressive" compiler flag
Score-P - A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU and Vampir
Why a new tool?

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

Vampir
Scalasca
TAU
Periscope

VampirTrace OTF
EPILOG / CUBE
TAU native formats
Online measurement
Idea

- Common infrastructure and effort
- Common data formats OTF2 and CUBE4
- Sharing ideas and implement faster
- No effort for maintenance, testing etc for various tools
- Single learning curve
Score-P Architecture

- Vampir
- Scalasca
- TAU
- Periscope

Score-P measurement infrastructure:
- Event traces (OTF2)
- Call-path profiles (CUBE4, TAU)
- Hardware counter (PAPI, rusage)
- Online interface
- Application (MPI, OpenMP, hybrid, serial)

TAU adaptor

Instrumentation
MPI wrapper

Compiler
TAU instrumentor
OPARI 2
User

Instrumentation wrapper
Components

- Separate, stand-alone packages
- Common functionality factored out
- Automated builds and tests
Score-P

- Instrumenter *scorep*
- Links application to measurement library
  
  *libscorep_(serial|omp|mpi|mpi_omp)*

- Records time, visits, communication metrics, hardware counters
  - Efficient utilization of available memory
  - Minimize perturbation/overhead
  - Useful for unification
  - Access data during runtime

- Switch modes (tracing, profiling, online) without recompilation
Score-P Instrumentation

Instrument

mpicc -c foo.c -> scorep mpicc -c foo.c

Help

% scorep --help
...
--user Enables manual user instrumentation.
--nouser Disables manual user instrumentation. Is disabled by default.
--pdt Enables source code instrumentation with PDT using the TAU instrumentor. It will automatically enable the user instrumentation and disable compiler instrumentation.

Instrument with PDT

scorep --pdt mpicc -c foo.c

Automatic detect serial/OpenMP/MPI/hybrid
Score-P Run-Time Recording

- Uncomment the appropriate MPIF77 command in config/make.def of NAS benchmarks
  \[ MPIF77 = \text{scorep mpif77} \]
- Customize via environment variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOREP_ENABLE_PROFILING</td>
<td>1</td>
<td>Setting to 0 turns off profiling</td>
</tr>
<tr>
<td>SCOREP_ENABLE_TRACING</td>
<td>1</td>
<td>Setting to 0 turns off tracing</td>
</tr>
<tr>
<td>SCOREP_TOTAL_MEMORY</td>
<td>1200k</td>
<td>Total memory in bytes for the measurement system excluding trace memory</td>
</tr>
<tr>
<td>SCOREP_EXPERIMENT_DIRECTORY</td>
<td>&quot;&quot;</td>
<td>Name of the experiment directory</td>
</tr>
<tr>
<td>SCOREP_MPI_ENABLE_GROUPS</td>
<td>DEFAULT</td>
<td>The names of the function groups which are measured (CG, P2P...)</td>
</tr>
<tr>
<td>SCOREP_SELECTIVE_CONFIG_FILE</td>
<td>&quot;&quot;</td>
<td>A file name which configures selective tracing</td>
</tr>
<tr>
<td>SCOREP_FILTER_FILE</td>
<td>&quot;&quot;</td>
<td>A file name which contain the filter rules</td>
</tr>
<tr>
<td>SCOREP_PROFILING_MAX_CALLPATH_DEPTH</td>
<td>30</td>
<td>Maximum depth of the calltree</td>
</tr>
<tr>
<td>SCOREP_PROFILING_FORMAT</td>
<td>DEFAULT</td>
<td>Profile output format (NONE, TAU_SNAPSHOT, CUBE4, DEFAULT)</td>
</tr>
<tr>
<td>SCOREP_METRIC_PAPI</td>
<td>&quot;&quot;</td>
<td>PAPI metric names</td>
</tr>
</tbody>
</table>

- It supports selective tracing
The Open Trace Format Version 2 (OTF2)

- Event trace data format
  - Event record types + definition record types
- Multi-file format
  - Anchor file
  - Global and local definitions + mappings
  - Event files
- OTF2 API
Re-design OTF2

- One process/thread per file
- Memory event trace buffer becomes part of trace format
- No re-write for unification, mapping tables
- Forward/Backward reading
Selective Tracing

- Score-P allows to disable the instrumentation on specific parts of the code (SCOREP_RECORDING_OFF/ON)
- It allows online access for handling the data on the fly for profiling mode
- Parameters profiling, we can split-up the callpath for executions of different parameter values (INT64, UINT64, String)
Future features

- Scalability to maximum available CPU core count
- Support for sampling, binary instrumentation
- Support for new architectures
- Allow experimental versions of new features or research
- Future integration in Open MPI releases
Accuracy: SkaMPI vs TAU vs Score-P

Score-P provides less overhead compared to TAU
Comparing the tracing of Score-P with and without PAPI and visualize the traces through Paje format. PAPI measurement adds some overhead.
Accuracy: Hardware counters

- Apply selective instrumentation for capturing only MPI events with PAPI without any info for the computation

```
BEGIN_FILE_EXCLUDE_LIST
*
END_FILE_EXCLUDE_LIST
```

- Execution of the LU benchmark, class A and 4 processes on the cluster bordereau (Grid’5000): 36.24 seconds, 82.36 billions instructions
- Without the exclusion: 46.4 seconds, 92.9 billions instructions
- 12.79% of the instructions caused by the instrumentation tool!
Accuracy: Scalasca and periodicity data

Compare the 126 actions per period –
Execution time

Time variation of the executed code which maps only to computation,
Thank you!
Questions?