Studying the behavior of parallel MPI applications

G. Markomanolis

INRIA, LIP, Avalon, ENS de Lyon

Working Group
Outline

1. Context and motivation
2. Introduction to Performance Engineering
3. Performance Application Programming Interface
4. Scalasca
5. TAU
6. PerfExpert
7. Score-P
8. Performance Analysis of Iterative Methods (PAIM)
9. Discuss about accuracy
Goals

- Overview of the programming tools suite
- Explain the functionality of the tools
- Presenting a tool about Performance Analysis of Iterative Methods
- Discussing about accuracy issues
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
Inclusive vs. Exclusive values

```
int foo()
{
    int a;
    a = 1 + 1;
    bar();
    a = a + 1;
    return a;
}
```
Sampling

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

```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);
}
```
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");
}
```
Critical issues

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

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

...
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-C)
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
Measurement event tracing & analysis

- Code instrumentation
- Measurements summarized by thread & call-path during execution
- Presentation of summary analysis
- Time-stamped events buffered for each thread
- Flushed to files
- Trace analysis
- Presentation of analysis report
Selective instrumentation

MPI_Init()
EPIK_PAUSE_START()
...
EPIK_PAUSE_END()
ssor(itmax)
EPIK_PAUSE_START()
...
EPIK_PAUSE_END()
MPI_Finalize()
Automatic instrumentation using PDT

- **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**
  
  ```
  BEGIN_EXCLUDE_LIST
  EXACT
  END_EXCLUDE_LIST
  ```
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 summary: LU benchmark, class A, 32 processors

- 45.22% of the time spent in MPI point-to-point communication
Scalasca trace: LU benchmark, class A, 32 processors

- 2.57% of the execution time corresponds to late sender
Scalasca trace: LU benchmark, class A, 32 processors

- 16.08% of the execution time corresponds to wrong order situation
LU summary analysis result scoring

% scalasca -examine -s epik_lu_a_32_sum...
...
Estimated aggregate size of event trace (total_tbc): 253721920 bytes
Estimated size of largest process trace (max_tbc): 9067400 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
...

- The estimated size of the traces is 242MB
- The maximum trace buffer is around to 9MB per process
  - If the available buffer is smaller than 9MB, then there will be bigger perturbation because of flushes to the hard disk during the measurement
Scalasca trace: LU benchmark, comparison B-32

- The difference between the optimization flags -O and -O3
## MPI Performance

<table>
<thead>
<tr>
<th></th>
<th>Processes</th>
<th>Class B</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI execution</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>93.7</td>
<td>203.64</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>124.56</td>
<td>439.52</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>201.12</td>
<td>482.3</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>311.44</td>
<td>649.68</td>
<td></td>
</tr>
<tr>
<td><strong>Late sender</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>26.68</td>
<td>54.2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11.19</td>
<td>46.22</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33.26</td>
<td>75.13</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>35.4</td>
<td>69.56</td>
<td></td>
</tr>
<tr>
<td><strong>Wrong source order</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9.54</td>
<td>17.87</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>31.26</td>
<td>110.9</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>38.36</td>
<td>96.46</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>72.24</td>
<td>142.69</td>
<td></td>
</tr>
</tbody>
</table>
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

- Performance profiling and tracing
- Instrumentation, measurement, analysis, visualization
- Performance data management and data mining
- TAU can automatically instrument your source code through PDT for routines, loops, I/O, memory, phases, etc.
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
Instrumentation, re-writing Binaries with MAQAO (beta)

Important

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

- **Paraprof:**

![TAU: ParaProf Manager](image)

![TAU: ParaProf: lu_a_4_intel_maqao.ppk](image)
Paraprof
<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLU (lu.f) {46,7}–{166,9}</td>
<td>0.437</td>
<td>97.336</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>INIT_COMM (init_comm.f) {5,7}–{57,9}</td>
<td>0</td>
<td>0.047</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MPI_Finalize()</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SSOR (ssor.f) {4,7}–{262,9}</td>
<td>3.021</td>
<td>96.851</td>
<td>1</td>
<td>100,260</td>
</tr>
<tr>
<td>BLTS (blts.f) {4,7}–{259,9}</td>
<td>15.478</td>
<td>16.91</td>
<td>25,000</td>
<td>50,000</td>
</tr>
<tr>
<td>EXCHANGE_1 (exchange_1.f) {5,7}–{177,9}</td>
<td>0.38</td>
<td>1.432</td>
<td>50,000</td>
<td>50,000</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>1.052</td>
<td>1.052</td>
<td>50,000</td>
<td>0</td>
</tr>
<tr>
<td>BUTS (buts.f) {4,7}–{259,9}</td>
<td>16.945</td>
<td>25.978</td>
<td>25,000</td>
<td>50,000</td>
</tr>
<tr>
<td>EXCHANGE_1 (exchange_1.f) {5,7}–{177,9}</td>
<td>0.483</td>
<td>9.032</td>
<td>50,000</td>
<td>50,000</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>8.549</td>
<td>8.549</td>
<td>50,000</td>
<td>0</td>
</tr>
<tr>
<td>JACLD (jacld.f) {5,7}–{385,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JACU (jacu.f) {5,7}–{381,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2NORM (l2norm.f) {4,7}–{68,9}</td>
<td>12.634</td>
<td>12.634</td>
<td>25,000</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Allreduce()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Allreduce()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHS (rhs.f) {5,7}–{506,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCHANGE_3 (exchange_3.f) {5,7}–{312,9}</td>
<td>0.465</td>
<td>4.205</td>
<td>502</td>
<td>1,506</td>
</tr>
<tr>
<td>MPI_Irecv()</td>
<td>0.003</td>
<td>0.003</td>
<td>502</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>1.209</td>
<td>1.209</td>
<td>502</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>2.527</td>
<td>2.527</td>
<td>502</td>
<td>0</td>
</tr>
<tr>
<td>TIMER_CLEAR (timers.f) {4,7}–{17,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER_READ (timers.f) {65,7}–{77,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER_START (timers.f) {23,7}–{37,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER_STOP (timers.f) {43,7}–{59,9}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Study the total instructions per function
Communication matrix display, function BLTS
View traces from the Jumpshot tool
Connection between various tools
Compare the duration of the functions while we increase the number of the processes (LU-B)

While we double the number of the processes the duration of the RHS function is decreased by 49.273%
Compare the total instructions of the functions while we increase the number of the processes (LU-B)

While we double the number of the processes the total instructions of the RHS function is decreased by 49.853%
Compare the duration of the functions for the rank 0 (LU-B)

While we double the number of the processes the duration of the RHS function is decreased by 48.685%
PerfExplorer, Total Execution Time for class B
PerfExplorer, Relative Efficiency by Event for class B (Time)
PerfExplorer, Relative Speedup by Event for class B (Time)
PerfExplorer, Runtime Breakdown for class B (Time)
PerfExplorer, Instructions per Second for class B

![Graph showing the mean of PAPI_TOT_INS / TIME for different number of processors.](image)
PerfExplorer, Relative Efficiency by Event for class C (Time)
PerfExplorer, Relative Efficiency by Event for class C (Stalled Cycles)
PerfExplorer, Instructions per Second for class C

Value Chart: (PAPI_TOT_INS/TIME)

Mean (PAPI_TOT_INS/TIME) - thousands (x 1E3)

Number of Processors

SSOR JACLD BUTS BLTS RHS JACU
Paraprof and dynamic phases for the LU benchmark, class B, 8 processes
We chose randomly the 112th iteration of the function RHS
Study the phase

We can observe that for the 112th iteration the variation of the total instructions is 8.33%
Study the phase II

- We can observe that for the 112th iteration the variation of the instructions per second is 6.4%
Conclusions

- The characteristics of a function can vary across different iteration.
- The metric of the stalled cycles on any resource is a good initial metric for identifying overhead but seems not to be enough.
- The class B scales better on 16 processes and more.
- Similar the class C for 32 processes.
PerfExpert
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
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**
  \[(\text{PAPI}_\text{LD}_\text{INS} \times \text{L1_dlat}) / \text{PAPI}_\text{TOT}_\text{INS}\]

- **Data Accesses, L2 data misses**
  \[((\text{PAPI}_\text{L2_TCM} - \text{PAPI}_\text{L2_ICM}) \times \text{mem_lat}) / \text{PAPI}_\text{TOT}_\text{INS}\]

- **Instruction Accesses, L2 instruction misses**
  \[
  \text{PAPI}_\text{L2_ICM} \times \text{mem_lat} / \text{PAPI}_\text{TOT}_\text{INS}
  \]
Output

Function rhs_() (25.8% of the total runtime)

<table>
<thead>
<tr>
<th>ratio to total instrns</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0....</td>
</tr>
<tr>
<td>- floating point</td>
<td>54</td>
</tr>
<tr>
<td>- data accesses</td>
<td>37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>performance assessment</th>
<th>LCPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>good...</td>
</tr>
<tr>
<td></td>
<td>okay...</td>
</tr>
<tr>
<td></td>
<td>fair...</td>
</tr>
<tr>
<td></td>
<td>poor...</td>
</tr>
<tr>
<td></td>
<td>bad...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>upper bound estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>data accesses</td>
</tr>
<tr>
<td>- L1d hits</td>
</tr>
<tr>
<td>- L2d hits</td>
</tr>
<tr>
<td>- L2d misses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>- L1i hits</td>
</tr>
<tr>
<td>- L2i hits</td>
</tr>
<tr>
<td>- L2i misses</td>
</tr>
</tbody>
</table>

| data TLB             | 0.0   |
| instruction TLB      | 0.0   |
| branch instructions  |
| - correctly predicted| 0.1   |
| - mispredicted       | 0.0   |
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

Call the autoscope

% 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
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

Event traces (OTF2)  \[\uparrow\]\n
Call-path profiles (CUBE4, TAU)  \[\uparrow\]\n
Hardware counter (PAPI, rusage)  \[\uparrow\]\n
Score-P measurement infrastructure

TAU adaptor

Instrumentation  \[\uparrow\]\n
MPI wrapper  \[\uparrow\]\n
Application (MPI, OpenMP, hybrid, serial)

Compiler  \[\uparrow\]\n
TAU instrumentor  \[\uparrow\]\n
OPARI 2  \[\uparrow\]\n
User  \[\uparrow\]\n
Instrumentation wrapper
Components

- Separate, stand-alone packages
- Common functionality factored out
- Automated builds and tests
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)
Performance Analysis of Iterative Methods (PAIM)
Why another one tool?

- The previous tools do not provide analytical information about the iterative methods
- One of the possible workloads of a scientific application is a loop, thus its behaviour should be studied further
- Think about an idea and implement it
Plotting all the iterations for the function SSOR (Computation time in ms)
Plotting all the iterations for the function SSOR (MPI_Send duration in ms)
Local Dynamic Phases

APPLU \{lu.f\} \{46,7\}-\{166,9\} \rightarrow SSOR \{ssor.f\} \{4,7\}-\{250,9\} \rightarrow ssor [2] \rightarrow RHS \{rhs.f\} \{5,7\}-\{506,9\}

\rightarrow RHS [2]

- What if a function is called more than once by SSOR

APPLU \{lu.f\} \{46,7\}-\{166,9\} \rightarrow SSOR \{ssor.f\} \{4,7\}-\{250,9\} \rightarrow ssor [2] \rightarrow JACU \{jacu.f\} \{5,7\}-\{384,9\}

- Analytical iterations: Execute again the benchmark and create the jacu [i] dynamic phases
- Local iterations: Aggregate the iterations to just one iteration per SSOR iteration
Plotting the local iterations for the function JACU (Time in ms)
Plotting the local iterations for the function JACU (Total Instructions)

This feature is called “per metric”, all the ranks per metric are included in one plot.
Plotting the local iterations for the function SSOR (MPI_Send duration in ms)
Plotting the iterations for the function RHS (Time in ms)
Comparing the instructions per second, for each function on rank 0

- Necessary to use different power rate for each function during the simulation
Comparing the instructions per second, for each function on rank 2

- Different power rate also across different processes
Comparing the total instructions, for each function on rank 0

- Function BUTS constitutes by almost 30% less total instructions than the function RHS.
Comparing the execution time for the computation parts, for each function on rank 0 (Time in ms)

- Function BUTS is almost 13% slower than function RHS
Comparing the stalled cycles on any resource, for each function on rank 0

- Functions BUTS and RHS have almost the same number of stalled cycles on any resource
Comparing the L1 data accesses on any resource, for each function on rank 0

- Function BUTS has almost 18% more L1 data accesses than function RHS
Scaling - LU benchmark, class B, for each function on rank 0 (Total Instructions)

Increasing the number of the processes by two, the total instructions are almost divided by two.
Scaling - LU benchmark, class B, zoom for 0-50 and 200-250 iterations (Total Instructions)
The workload is increasing by almost four times.
Actions between performance data

![Graph showing computation data with iterations and inclusive values. The graph includes multiple lines with labels: buts_TIME_B_8/TIME_B_16-0, buts_TIME_B_8/TIME_B_32-0, buts_TIME_B_8/TIME_B_64-0.](image-url)
Use statistics in the case of many processes
Using the optimization flag -O3
Compare five executions of the same instance
Accuracy: SkaMPI vs TAU vs Score-P

Score-P provides less overhead compared to TAU
Decreasing the overhead of the instrumentation

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

```plaintext
BEGIN_FILE_EXCLUDE_LIST
*
END_FILE_EXCLUDE_LIST
```
Thank you!
Questions?