Studying the behavior of parallel MPI applications

G. Markomanolis

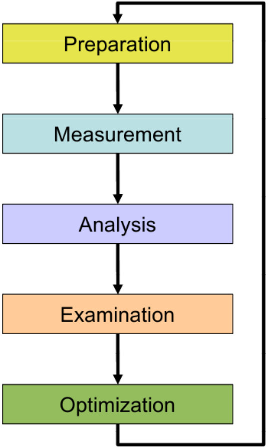
INRIA, LIP, Avalon, ENS de Lyon

Working Group

Outline

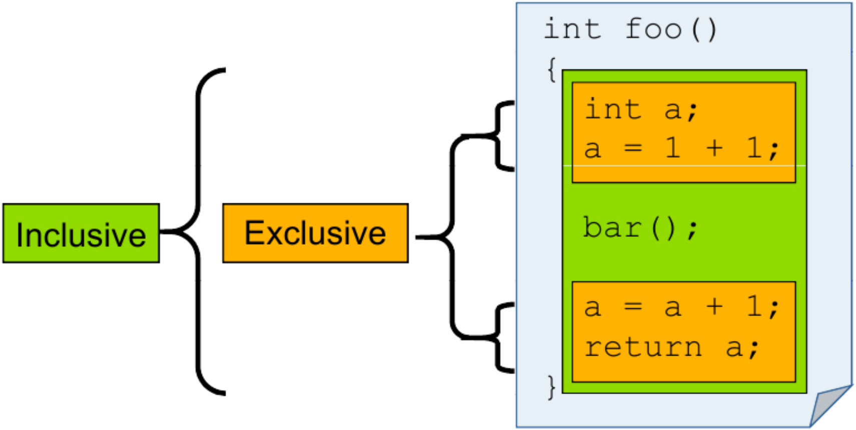
- Goals
  - Overview of the programming tools suite
  - Explain the functionality of the tools
  - Presenting a tool about Performance Analysis of Iterative Methods
  - Discussing about accuracy issues

Performance engineering workflow

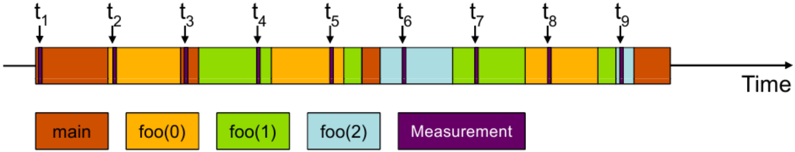


- Prepare application
- Collect the relevant data to the execution of the instrumented application
- Identification of performance metrics
- Presentation of results
- Modifications in order to reduce performance problems

Inclusive vs. Exclusive values



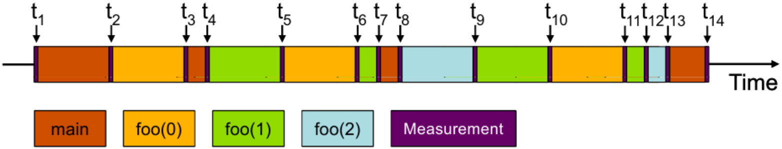
Sampling




```
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




```
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");
}
```

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

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

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

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:

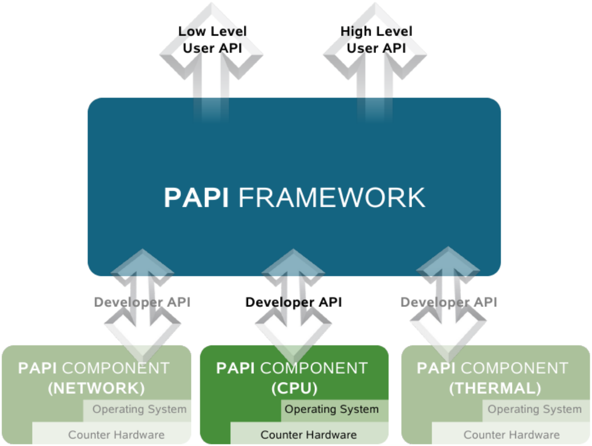
- Whole program timing
- Cache behaviors

...

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

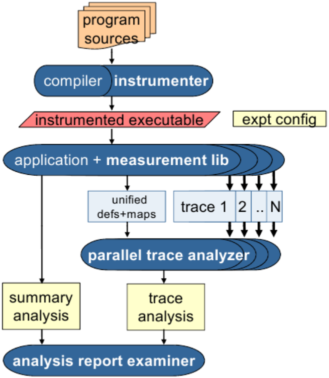
Scalasca

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 widlcard for a routine name and the mark * is a wildcard character
- Include functions for instrumentation

```
BEGIN_INCLUDE_LIST/END_INCLUDE_LIST
```

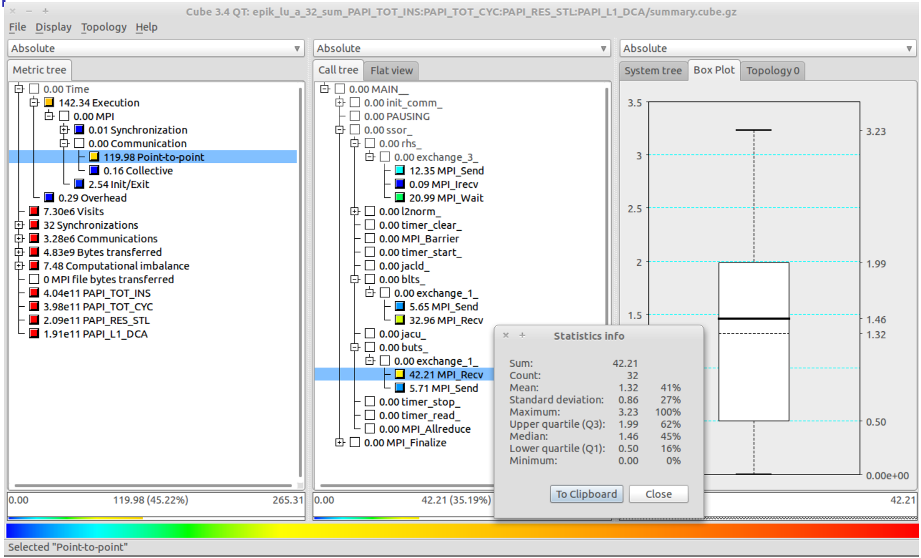
- Exclude the function EXACT from the LU benchmark

```
BEGIN_EXCLUDE_LIST
EXACT
END_EXCLUDE_LIST
```

NPB -MPI / LU

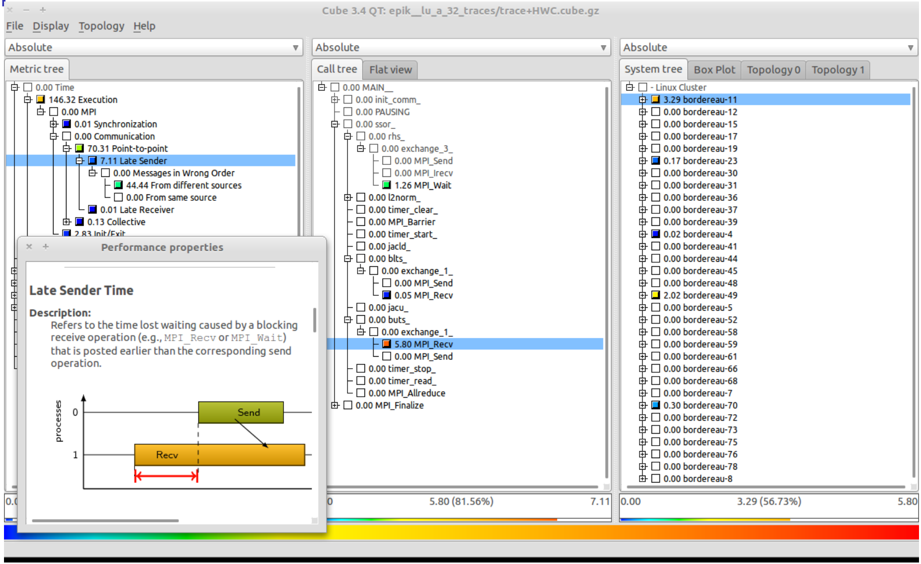
- Studying the MPI version of the LU benchmark from the NAS Parallel Benchmarks (NPB) suite
- Summary measurement & analysis
  - Automatic instrumentation
  - Summary analysis report examination
  - PAPI hardware counter metrics
- Trace measurement collection & analysis
  - Filter determination, specification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

Scalasca summary: LU benchmark, class A, 32 processors



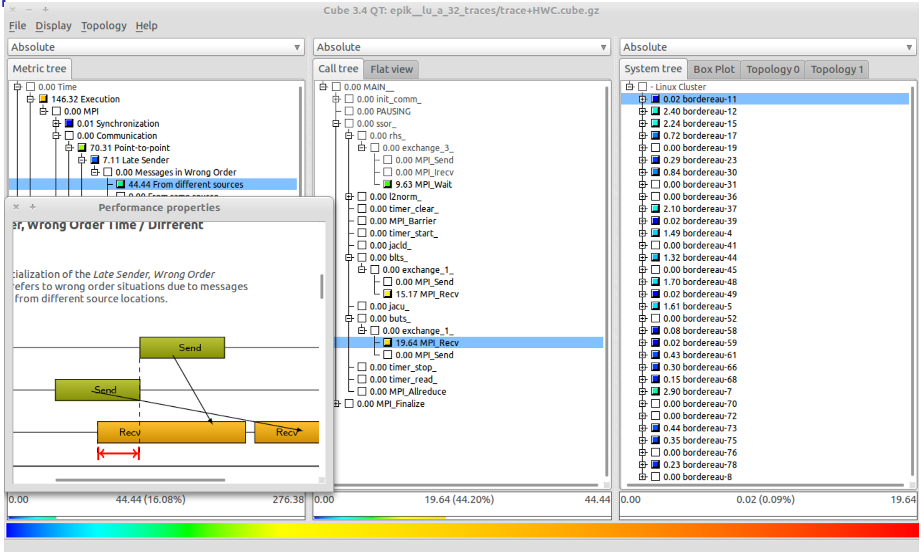
- 45.22% of the time spent in MPI point-to-point communication

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



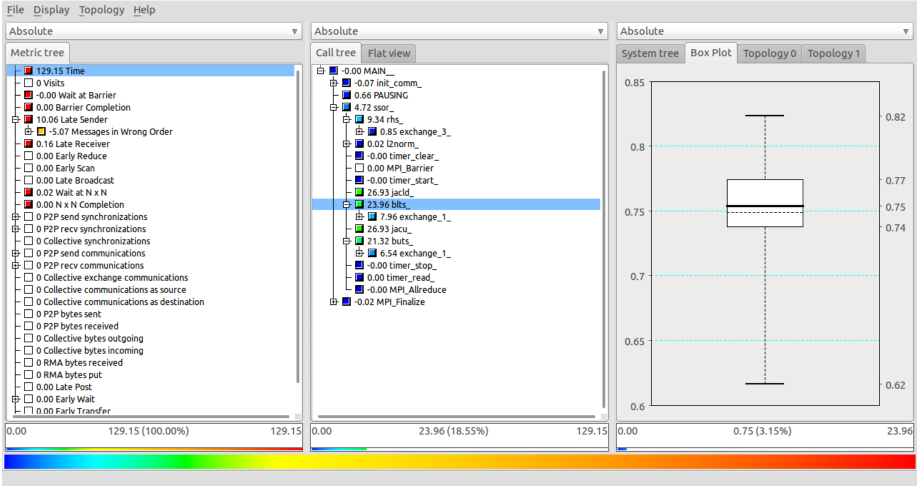
- 16.08% of the execution time corresponds to wrong order situation

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 different between the optimization flags -O and -O3

MPI Performance

	Processes	Class B	Class C
MPI execution	8	93.7	203.64
	16	124.56	439.52
	32	201.12	482.3
	64	311.44	649.68
Late sender	8	26.68	54.2
	16	11.19	46.22
	32	33.26	75.13
	64	35.4	69.56
Wrong source order	8	9.54	17.87
	16	31.26	110.9
	32	38.36	96.46
	64	72.24	142.69

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

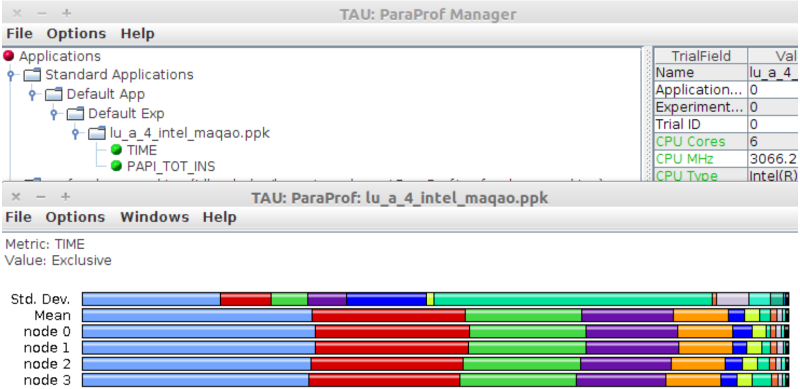
Instrumentation, re-writing Binaries with MAQAO (beta)

Important

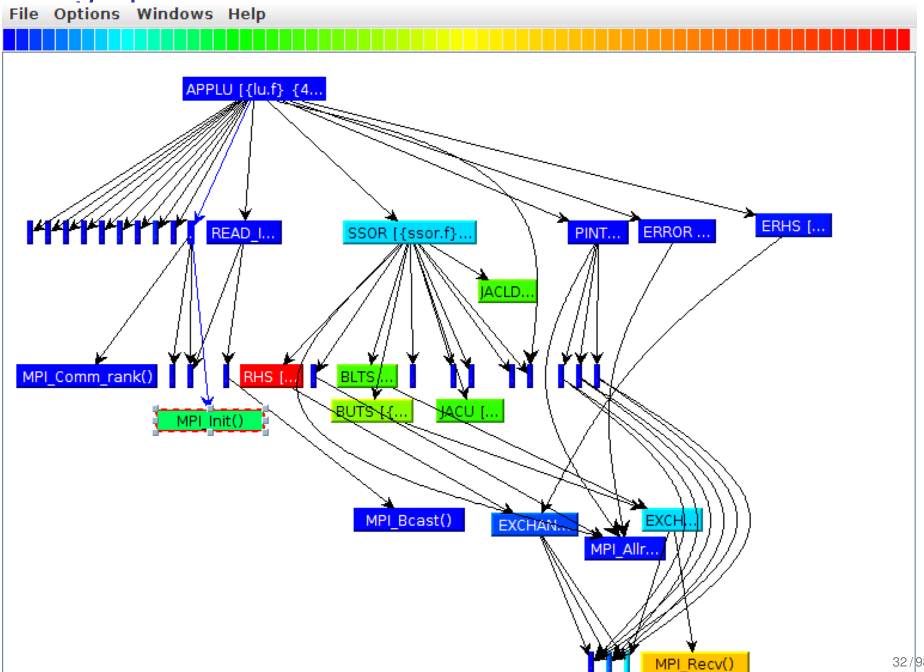
- Instrument:

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

- Paraprof:

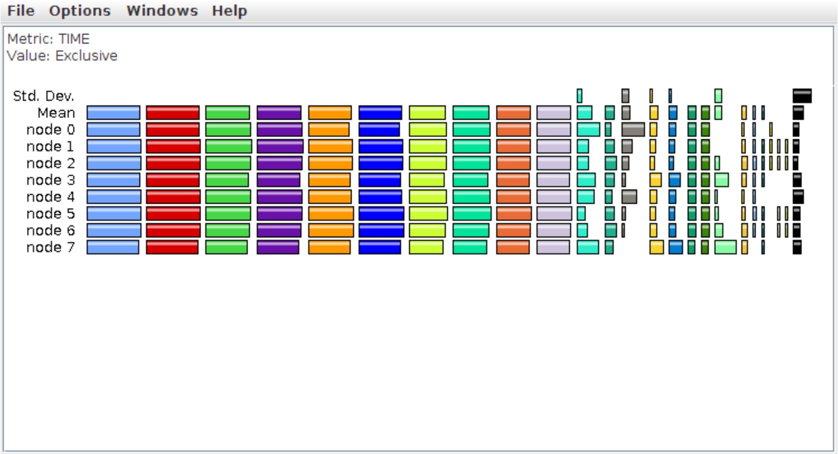


Call graph

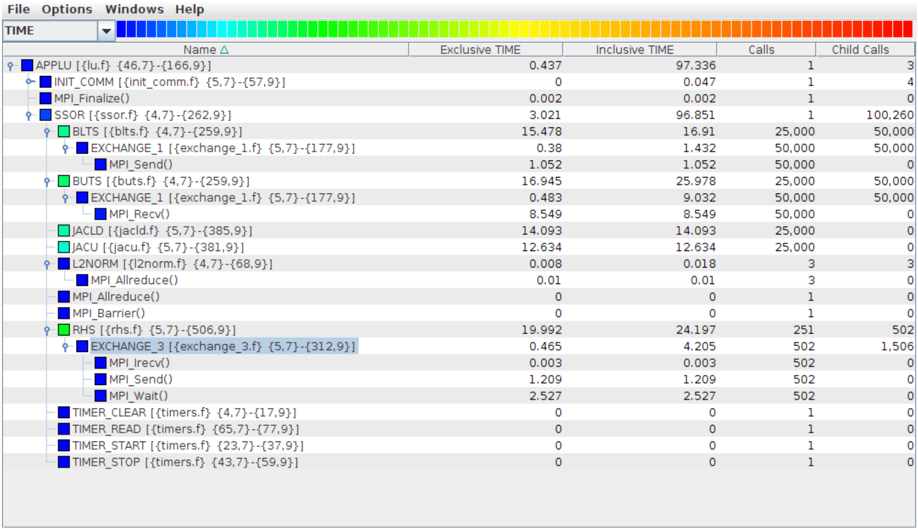


File Options Windows Help
APPLU [{lu.f} {4...
READ_I...
SSOR [{ssor.f}...
PINT...
ERROR ...
ERHS [...
JACLD...
MPI_Comm_rank()
RHS [...
BLTS [...
BUTS [{...
JACU [...
MPI_Init()
MPI_Bcast()
EXCHAN...
EXCH...
MPI_Allr...
MPI_Recv()

Paraprof

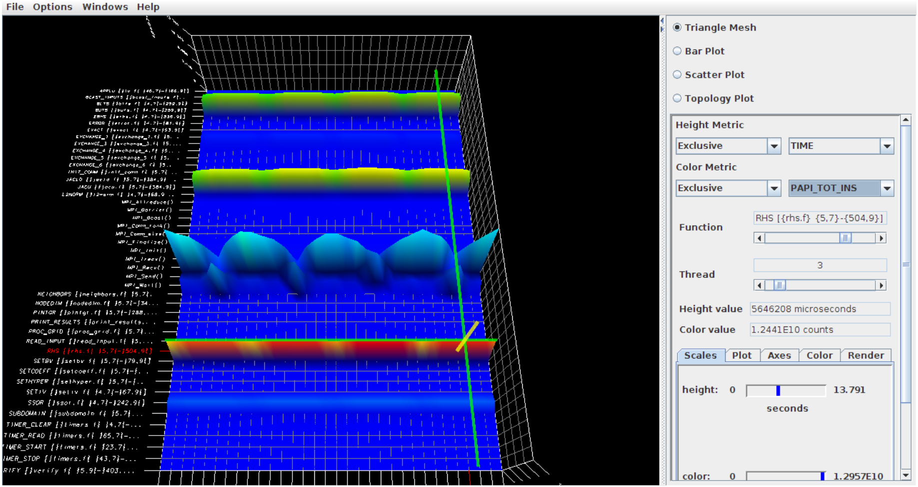

File Options Windows Help
Metric: TIME
Value: Exclusive
Std. Dev.
Mean
node 0
node 1
node 2
node 3
node 4
node 5
node 6
node 7

Paraprof II



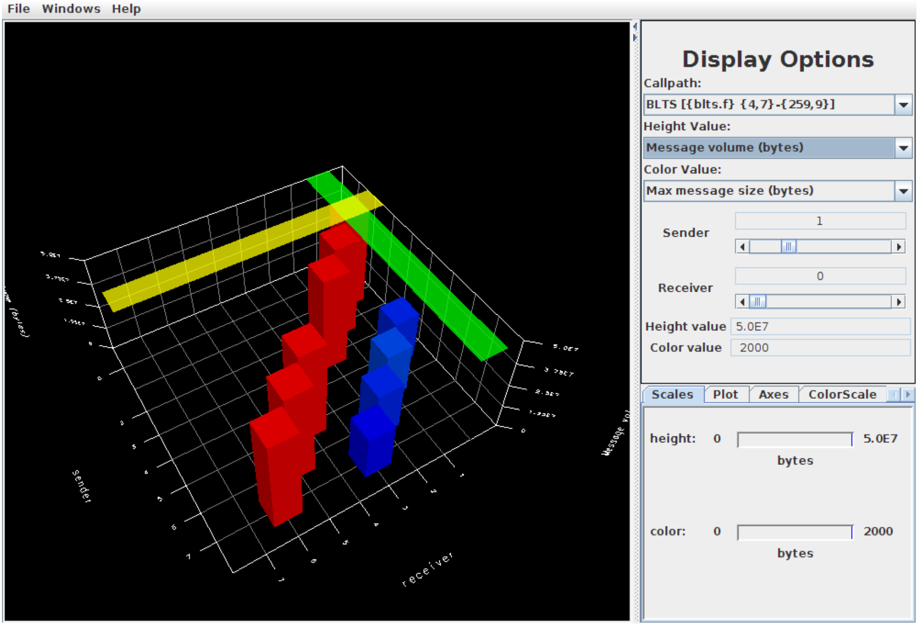
Name △	Exclusive TIME	Inclusive TIME	Calls	Child Calls
APPLU [{lu.f} {46,7}-{166,9}]	0.437	97.336	1	3
INIT_COMM [{init_comm.f} {5,7}-{57,9}]	0	0.047	1	4
MPI_Finalize()	0.002	0.002	1	0
SSOR [{ssor.f} {4,7}-{262,9}]	3.021	96.851	1	100,260
BLTS [{blts.f} {4,7}-{259,9}]	15.478	16.91	25,000	50,000
EXCHANGE_1 [{exchange_1.f} {5,7}-{177,9}]	0.38	1.432	50,000	50,000
MPI_Send()	1.052	1.052	50,000	0
BUTS [{buts.f} {4,7}-{259,9}]	16.945	25.978	25,000	50,000
EXCHANGE_1 [{exchange_1.f} {5,7}-{177,9}]	0.483	9.032	50,000	50,000
MPI_Recv()	8.549	8.549	50,000	0
JACLD [{jacld.f} {5,7}-{385,9}]	14.093	14.093	25,000	0
JACU [{jacu.f} {5,7}-{381,9}]	12.634	12.634	25,000	0
L2NORM [{l2norm.f} {4,7}-{68,9}]	0.008	0.018	3	3
MPI_Allreduce()	0.01	0.01	3	0
MPI_Allreduce()	0	0	1	0
MPI_Barrier()	0	0	1	0
RHS [{rhs.f} {5,7}-{506,9}]	19.992	24.197	251	502
EXCHANGE_3 [{exchange_3.f} {5,7}-{312,9}]	0.465	4.205	502	1,506
MPI_Irecv()	0.003	0.003	502	0
MPI_Send()	1.209	1.209	502	0
MPI_Wait()	2.527	2.527	502	0
TIMER_CLEAR [{timers.f} {4,7}-{17,9}]	0	0	1	0
TIMER_READ [{timers.f} {65,7}-{77,9}]	0	0	1	0
TIMER_START [{timers.f} {23,7}-{37,9}]	0	0	1	0
TIMER_STOP [{timers.f} {43,7}-{59,9}]	0	0	1	0

3D Visualization, time, total instructions

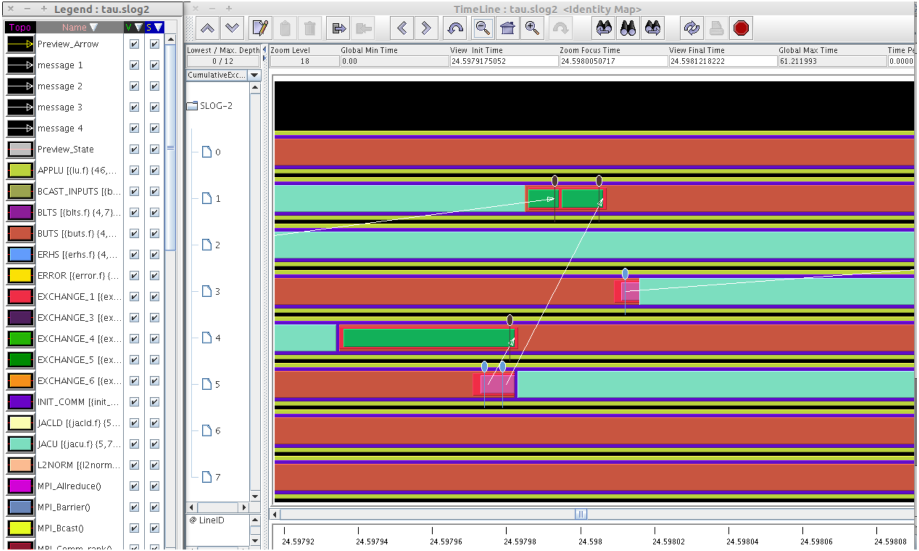


- Study the total instructions per function

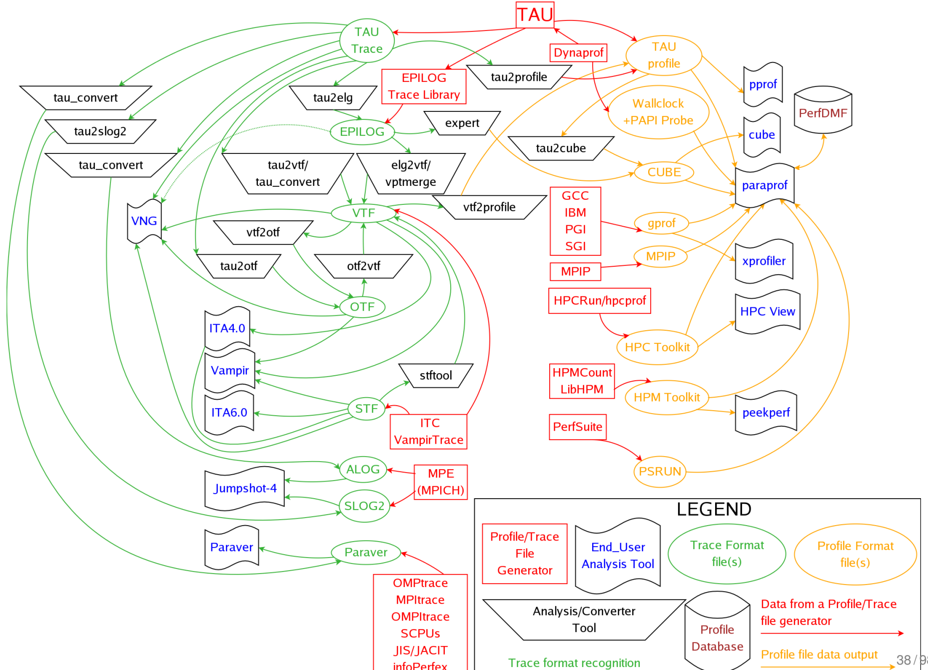
Communication matrix display, function BLTS



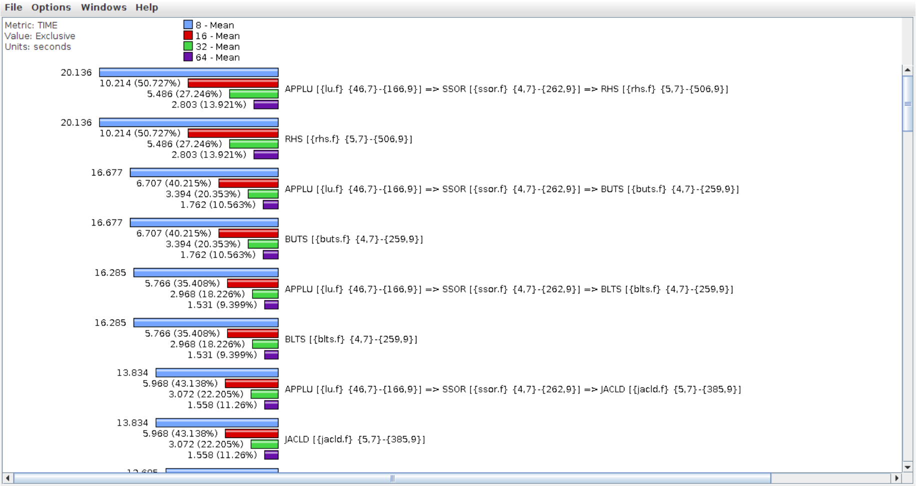
View traces from the Jumpshot tool



Connection between various tools

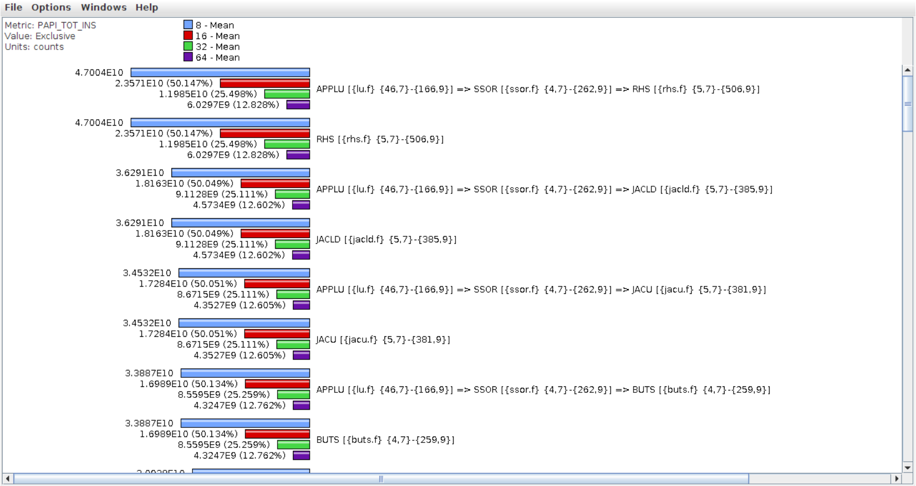


Compare the duration of the functions while we increase the number of the processes (LU-B)



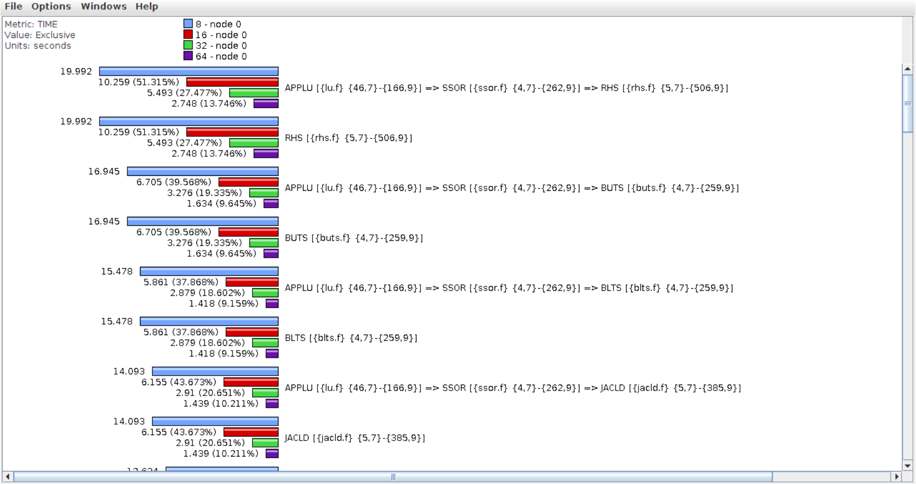
- While we double the number of the processes the duration of the RHS function is decreased by 49.273%

Compare the total instructions of the functions while we increase the number of the processes (LU-B)



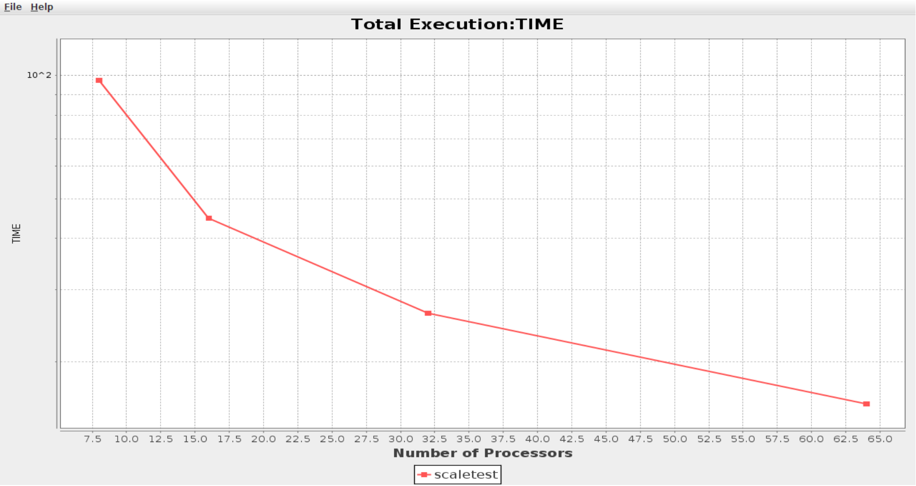
- While we double the number of the processes the total instructions of the RHS function is decreased by 49.853%

Compare the duration of the functions for the rank 0 (LU-B)

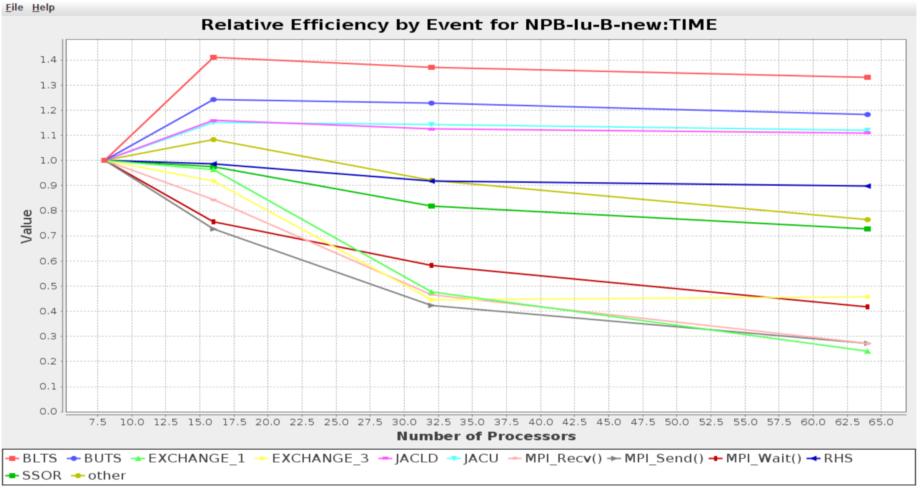


- While we double the number of the processes the duration of the RHS function is decreased by 48.685%

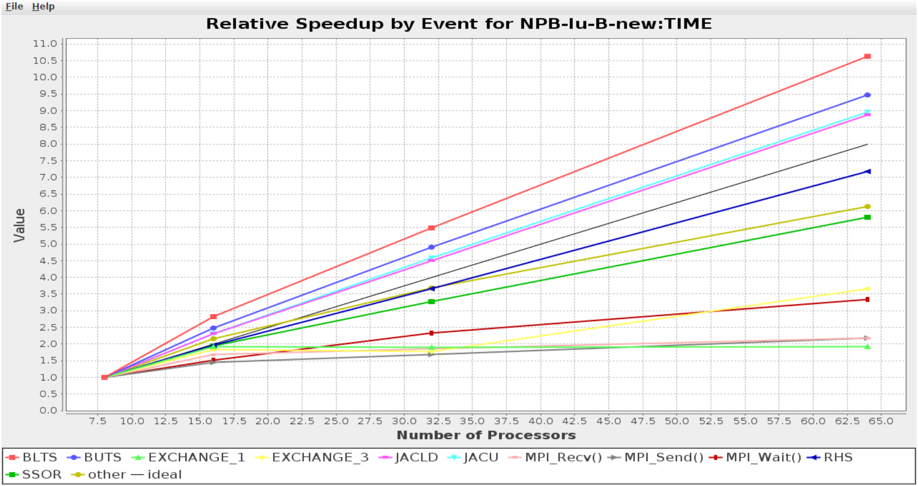
PerfExplorer, Total Execution Time for class B



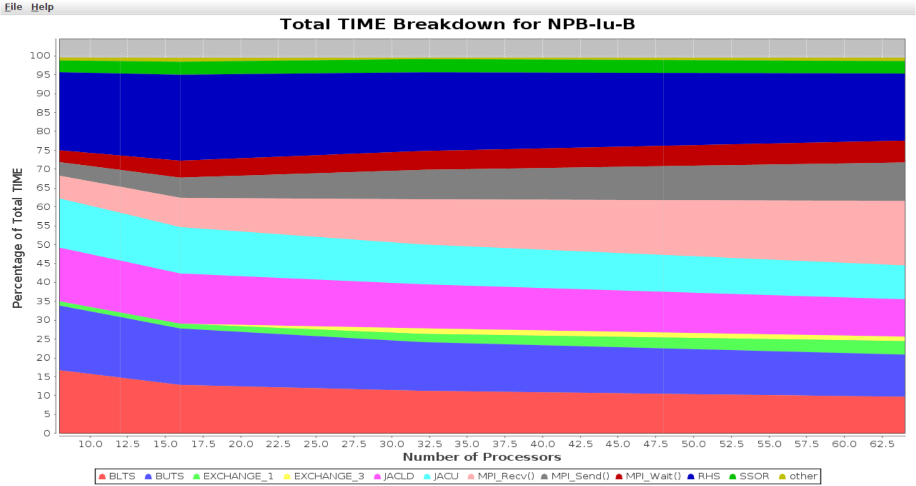
PerfExplorer, Relative Efficiency by Event for class B (Time)



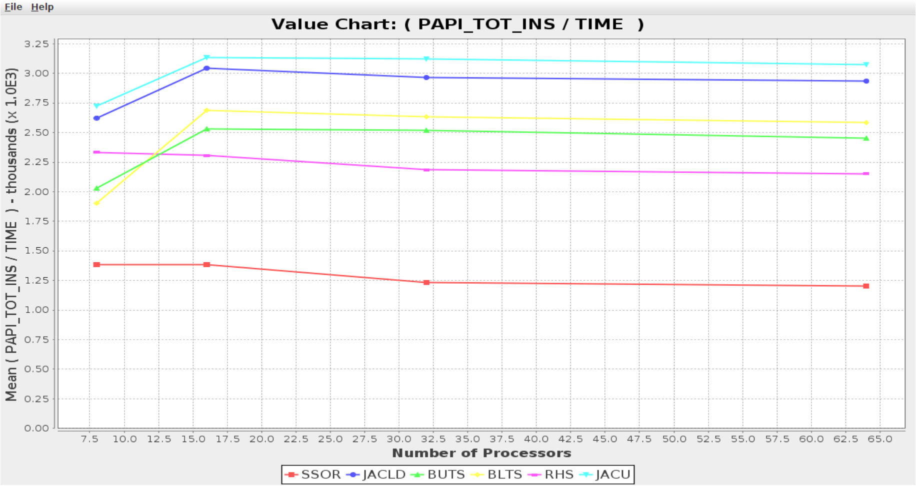
PerfExplorer, Relative Speedup by Event for class B (Time)



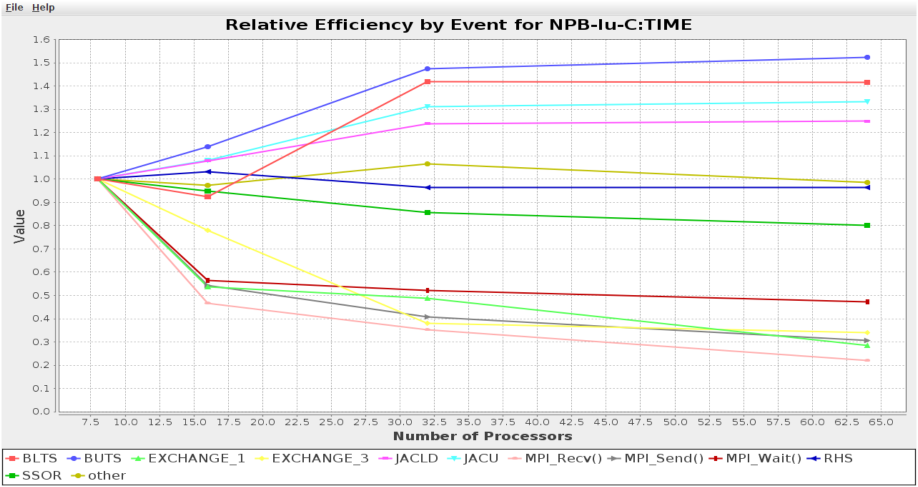
PerfExplorer, Runtime Breakdown for class B (Time)



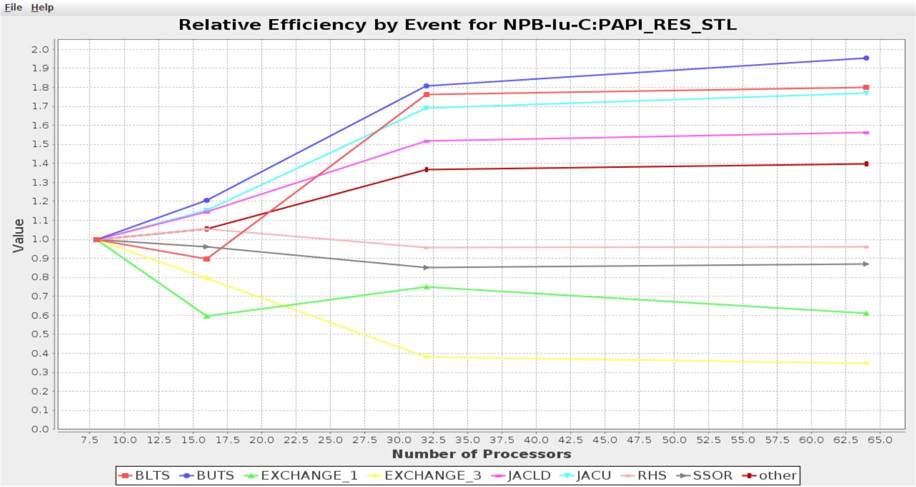
PerfExplorer, Instructions per Second for class B



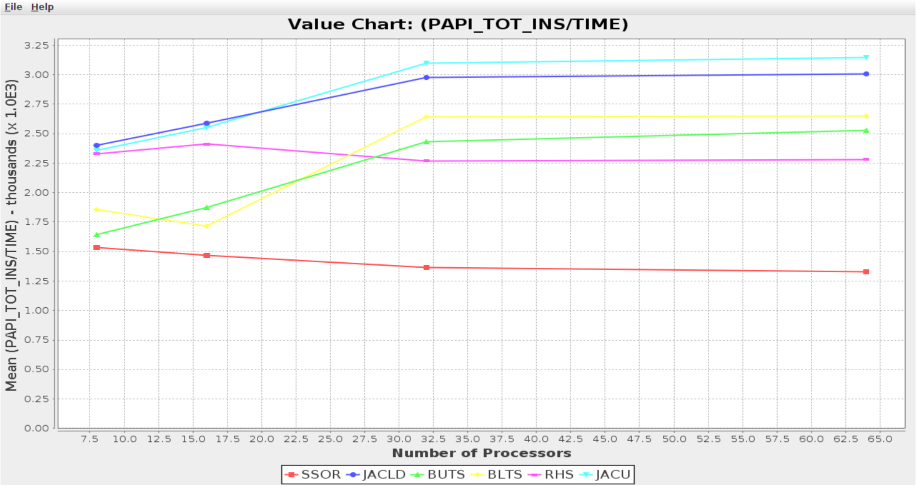
PerfExplorer, Relative Efficiency by Event for class C (Time)



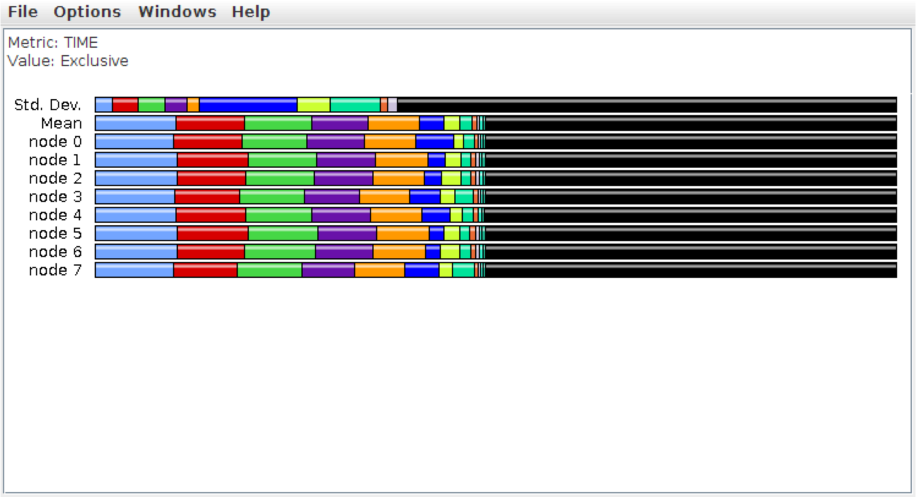
PerfExplorer, Relative Efficiency by Event for class C (Stalled Cycles)



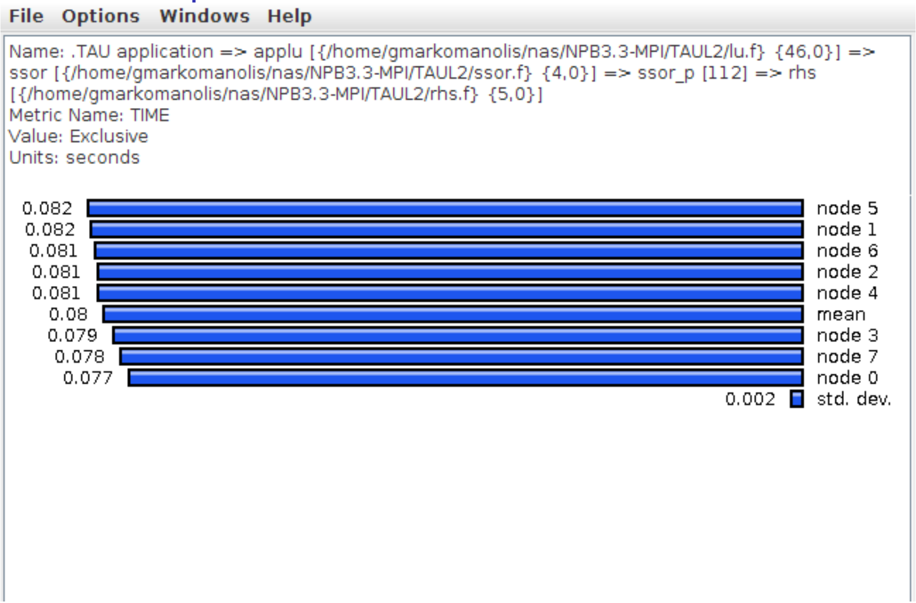
PerfExplorer, Instructions per Second for class C



Paraprof and dynamic phases for the LU benchmark, class B, 8 processes

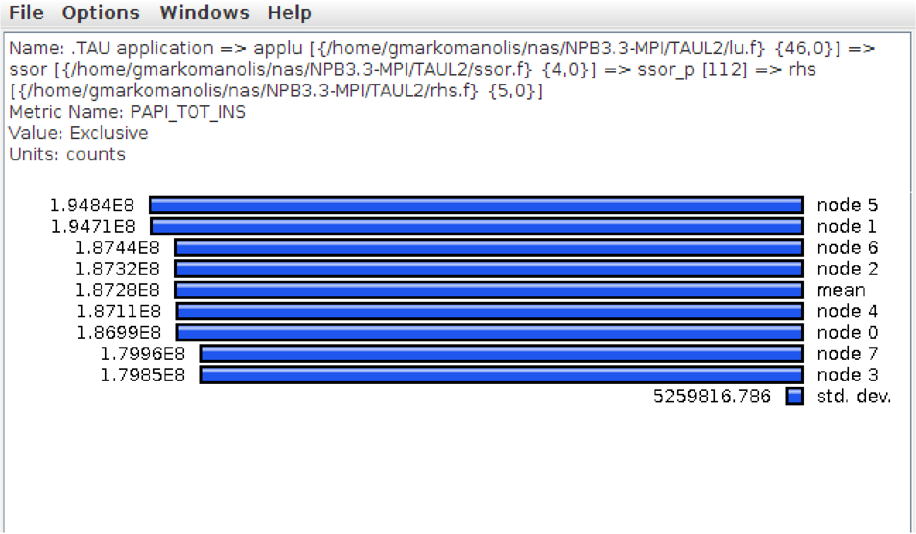


Profile of a phase



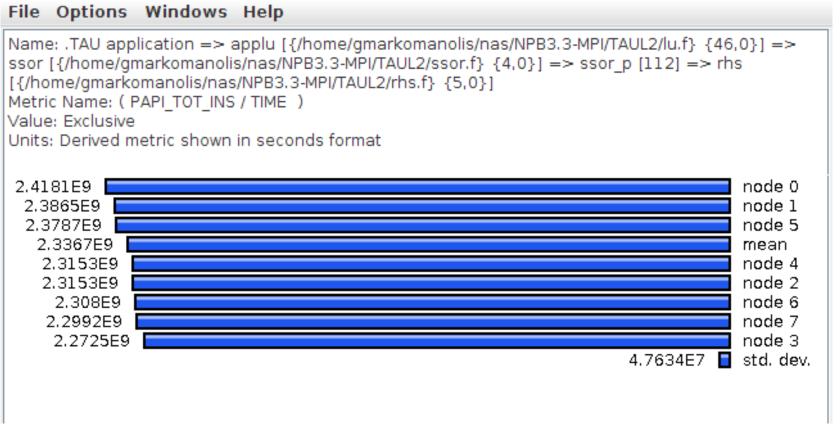
- We chose randomly the 112th iteration of the function RHS

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

  `(PAPI_LD_INS * L1_dlat) / PAPI_TOT_INS`
- Data Accesses, L2 data misses

  `((PAPI_L2_TCM - PAPI_L2_ICM) * mem_lat) / PAPI_TOT_INS`
- Instruction Accesses, L2 instruction misses

  `PAPI_L2_ICM * mem_lat / PAPI_TOT_INS`

Output

```
Function rhs_() (25.8% of the total runtime)
=============================================================================
ratio to total instrns     %  0.........25...........50.........75........100
   - floating point      :   54 *************************
   - data accesses       :   37 ******************
-------------------------------------------------------------------------------
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                :  1.1 >>>>>>>>>>>>>>>>>>>>>>>
upper bound estimates
* data accesses          :  1.4 >>>>>>>>>>>>>>>>>>>>>>
   - L1d hits            :  0.7 >>>>>>>>>>>>>>
   - L2d hits            :  0.7 >>>>>>>>>>>>>>
   - L2d misses          :  0.0 >
* instruction accesses   :  0.3 >>>>>>>
   - L1i hits            :  0.3 >>>>>>>
   - L2i hits            :  0.0 >
   - L2i misses          :  0.0 >
* data TLB               :  0.0 >
* instruction TLB        :  0.0 >
* branch instructions    :  0.1 >>
   - 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();
```

AutoSCOPE

```
* 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?

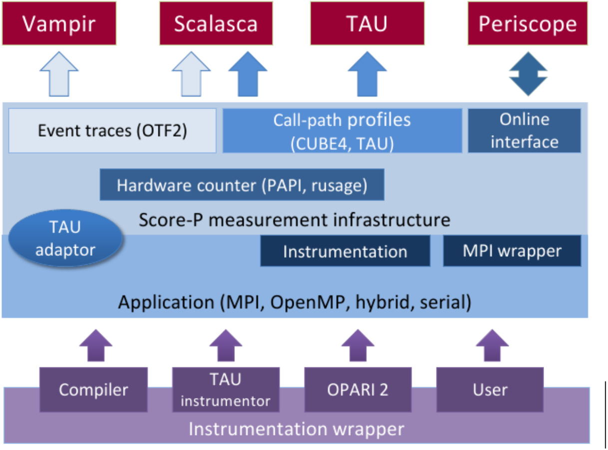
- Several performance tools co-exist
- Different measurement systems and output format
- Complementary features and overlapping functionality
- Redundant effort for development and maintenance
- Limited or expensive interoperability
- Complications for user experience, support, training

Vampir	Scalasca	TAU	Periscope
VampirTrace OTF	EPILOG / CUBE	TAU native formats	Online measurement

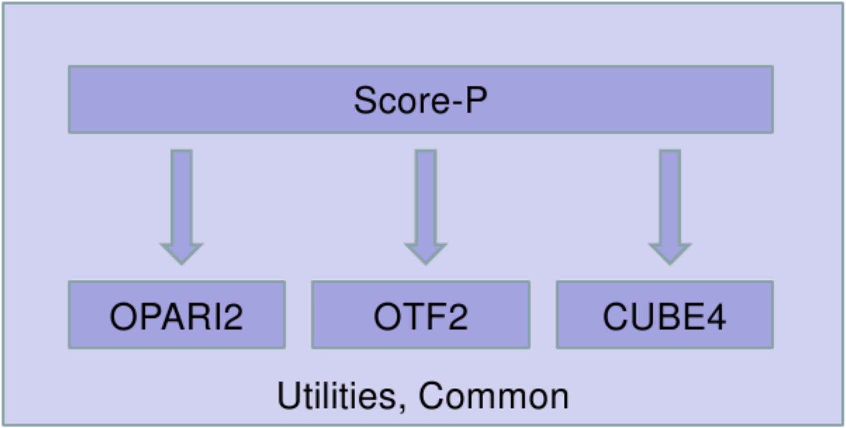
Idea

- 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



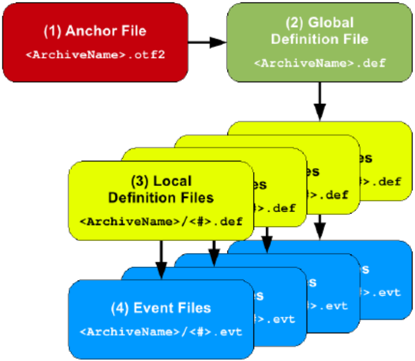
Components



- 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

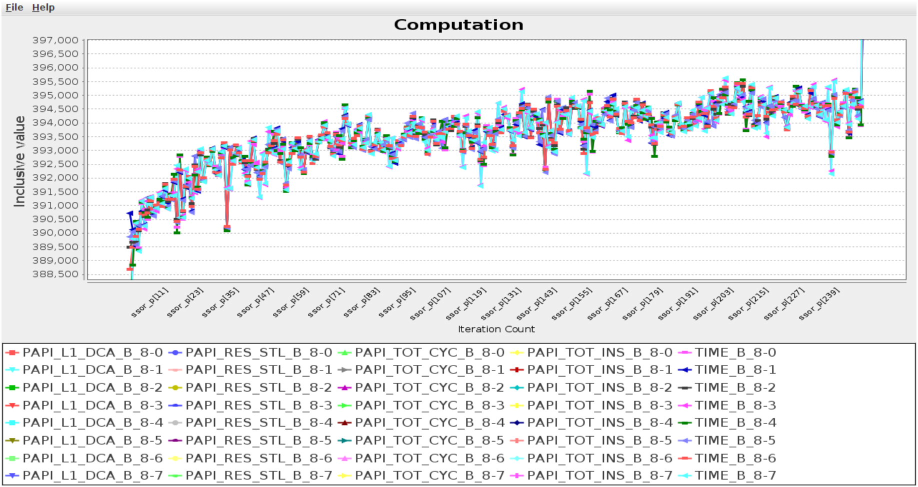
- Score-P allows to disable the instrumentation on specific parts of the code (SCOREP_RECORDING_OFF/ON)
- It allows online access for handling the data on the fly for profiling mode
- Parameters profiling, we can split-up the callpath for executions of different parameter values (INT64, UINT64, String)

Performance Analysis of Iterative Methods (PAIM)

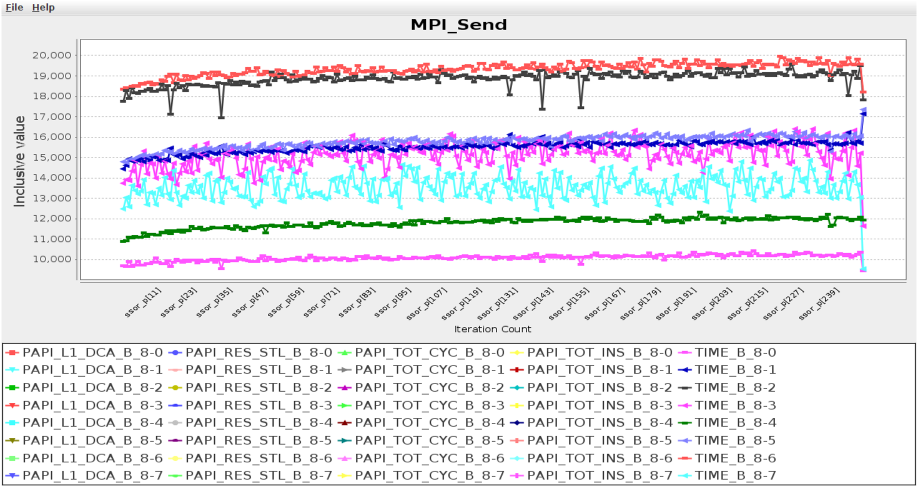
Why another one tool?

- 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}] => SSOR [{ssor.f} {4,7}-{250,9}]
=> ssor [2] => RHS [{rhs.f} {5,7}-{506,9}]
```

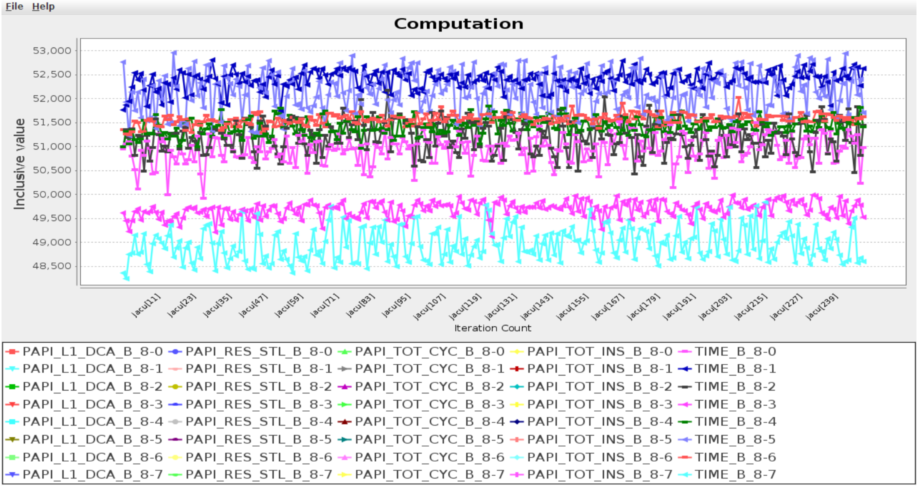
```
-> RHS [2]
```

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

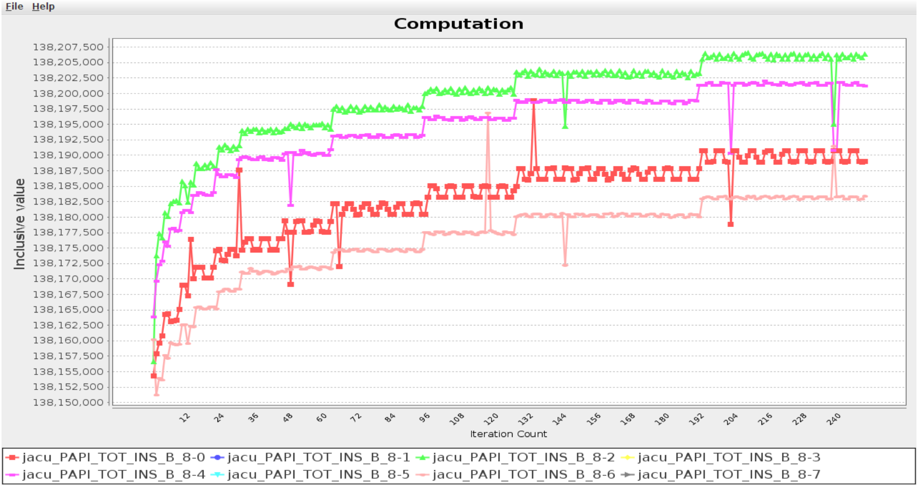
  ```
  APPLU [{lu.f} {46,7}-{166,9}] => SSOR [{ssor.f} {4,7}-{250,9}]
  => ssor [2] => JACU [{jacu.f} {5,7}-{384,9}]
  ```

  - 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)

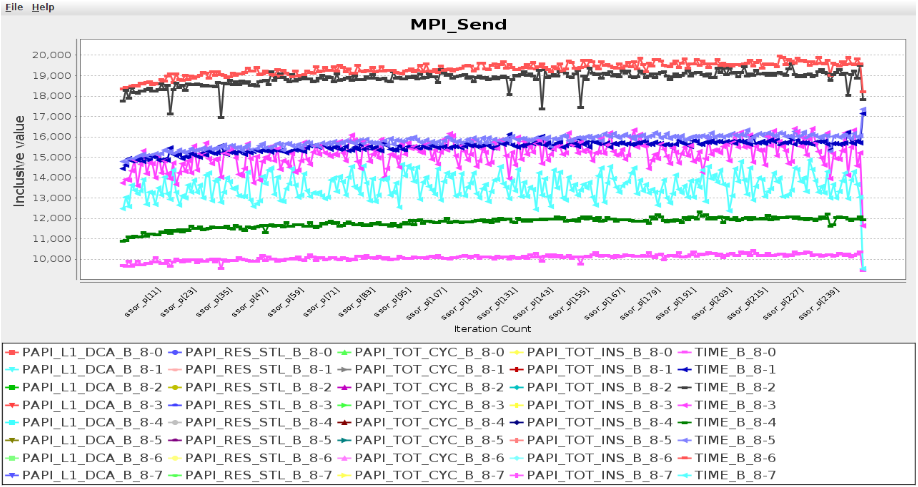


Plotting the local iterations for the function JACU (Total Instructions)

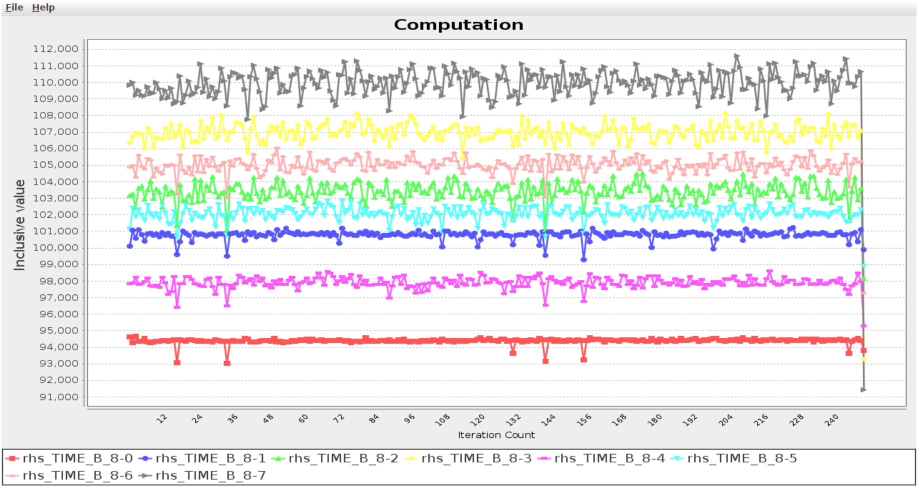


- This feature is called “per metric”, all the ranks per metric are included in one plot

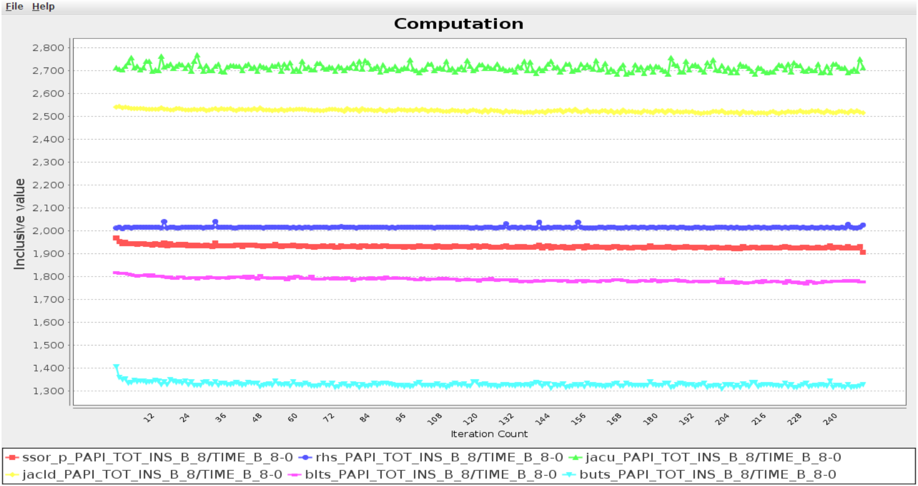
Plotting the local iterations for the function SSOR (MPI_Send duration in ms)



Plotting the iterations for the function RHS (Time in ms)

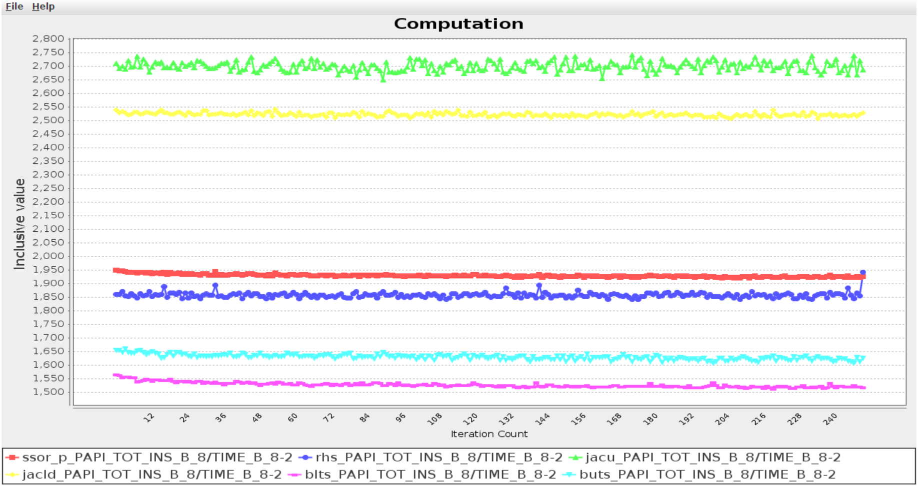


Comparing the instructions per second, for each function on rank 0



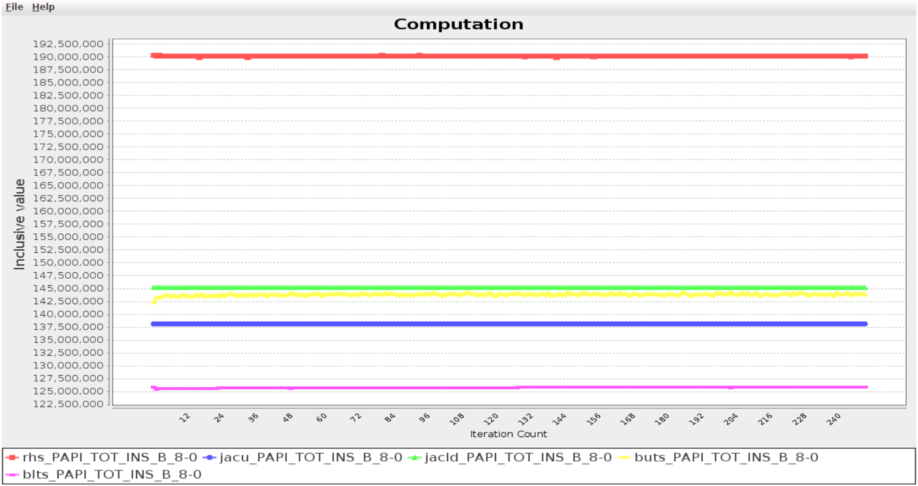
- Necessary to use different power rate for each function during the simulation

Comparing the instructions per second, for each function on rank 2



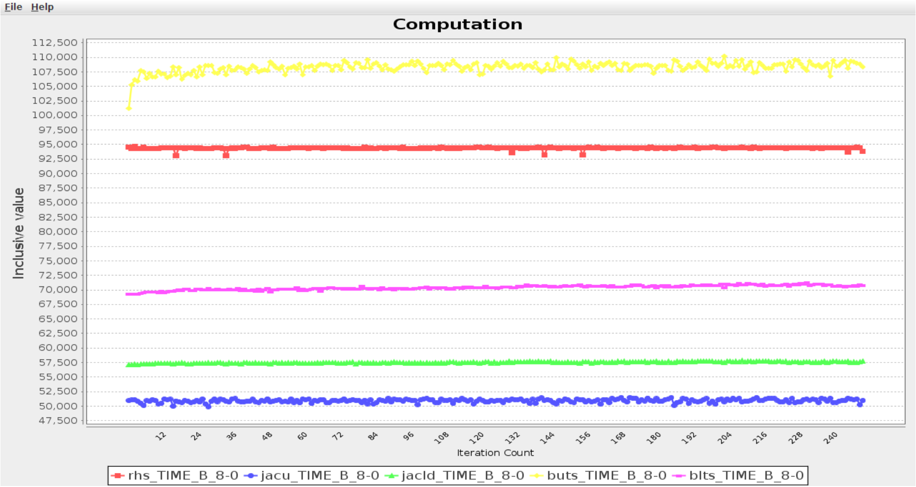
- Different power rate also across different processes

Comparing the total instructions, for each function on rank 0



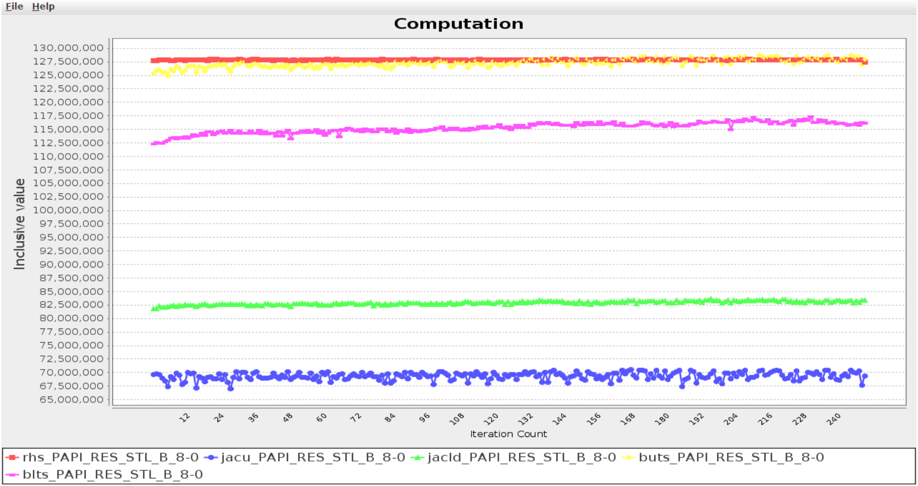
- Function BUTS constitutes by almost 30% less total instructions than the function RHS

Comparing the execution time for the computation parts, for each function on rank 0 (Time in ms)



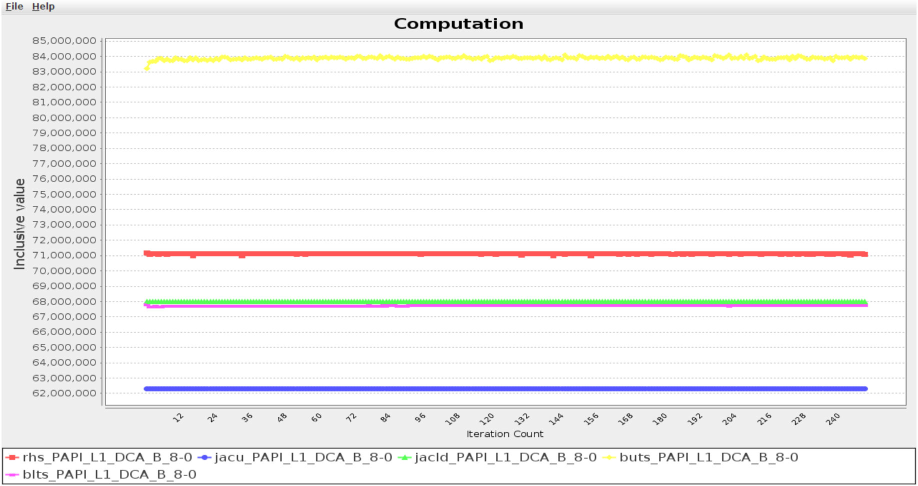
- Function BUTS is almost 13% slower than function RHS

Comparing the stalled cycles on any resource, for each function on rank 0



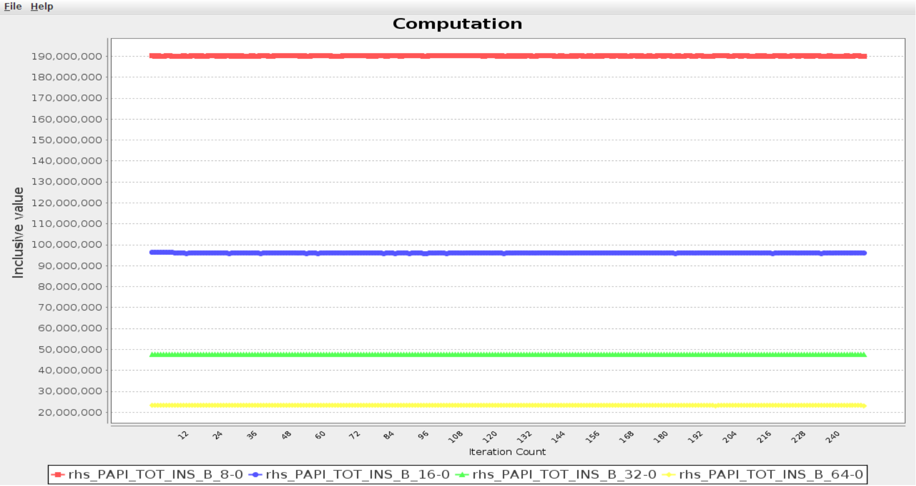
- Functions BUTS and RHS have almost the same number of stalled cycles on any resource

Comparing the L1 data accesses on any resource, for each function on rank 0



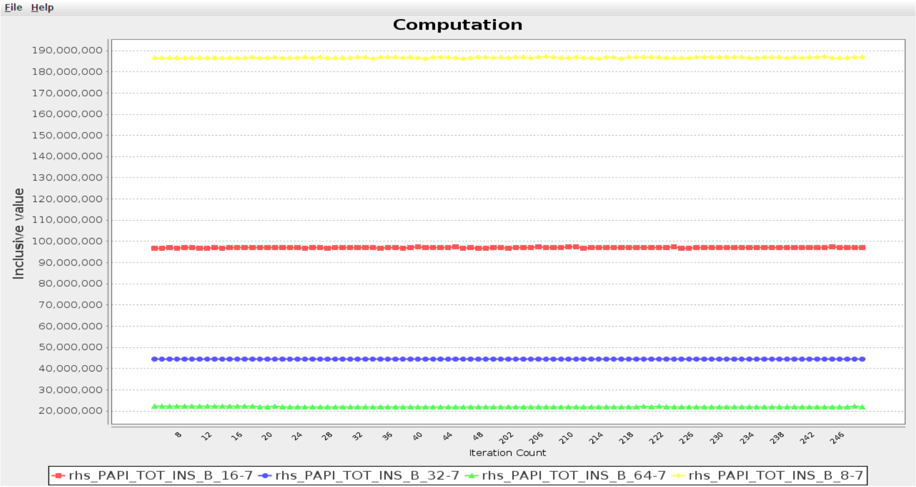
- Function BUTS has almost 18% more L1 data accesses than function RHS

Scaling - LU benchmark, class B, for each function on rank 0 (Total Instructions)

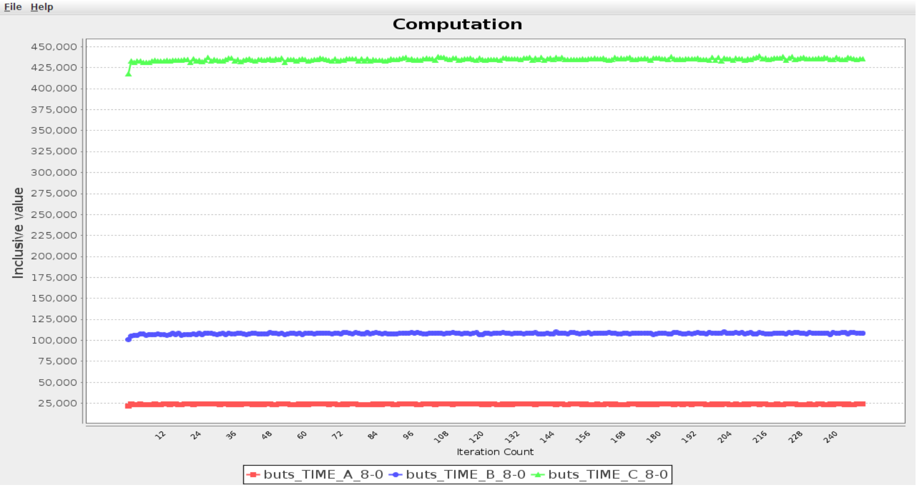


- Increasing the number of the processes by two, the total instructions are almost divided by two.

Scaling - LU benchmark, class B, zoom for 0-50 and 200-250 iterations (Total Instructions)

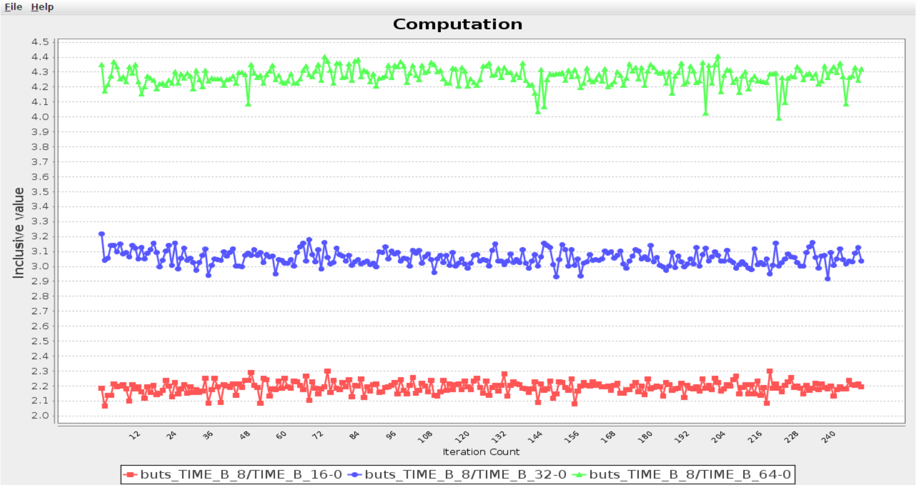


Scaling different instances (Time in ms)

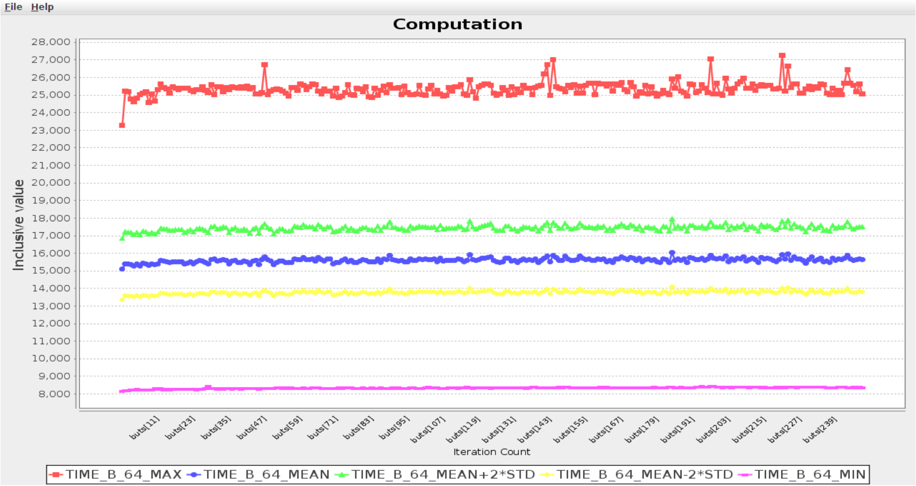


- The workload is increasing by almost four times.

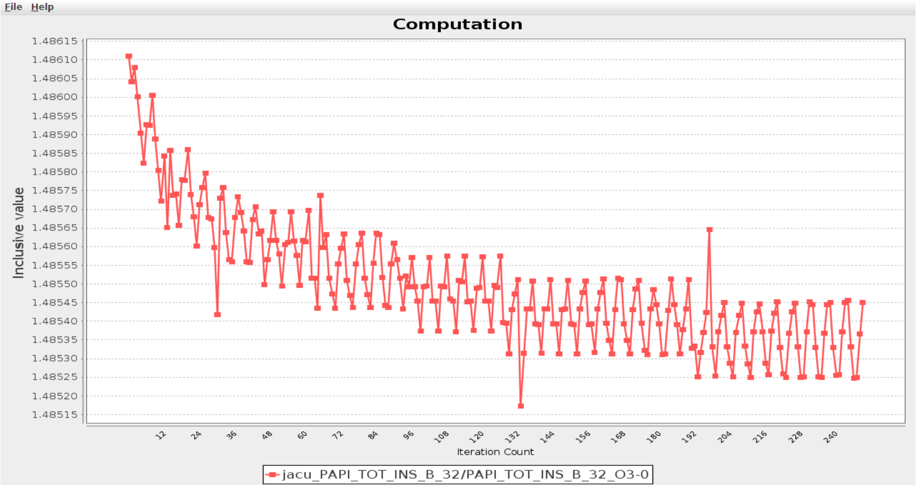
Actions between performance data



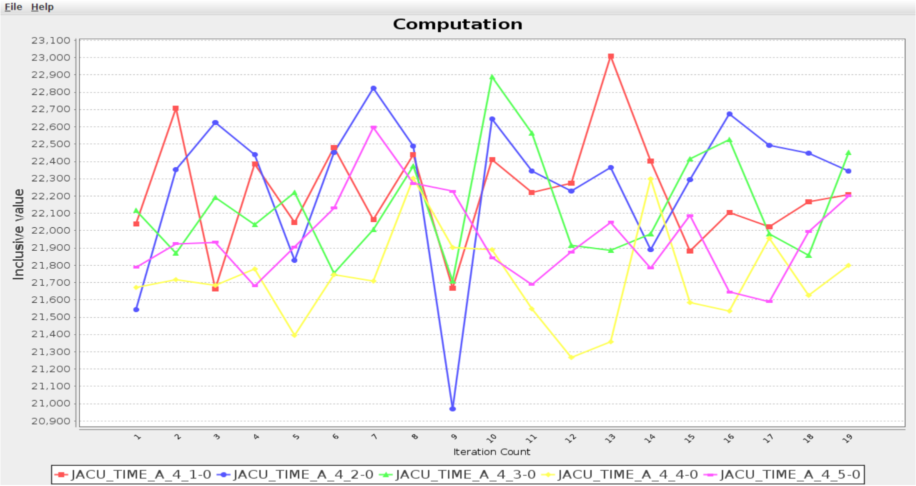
Use statistics in the case of many processes



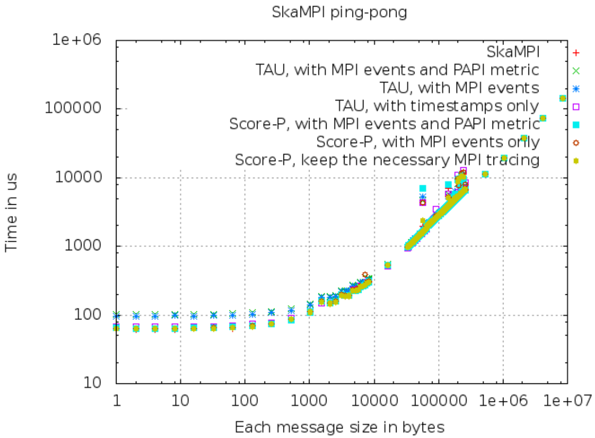
Using the optimization flag -O3



Compare five executions of the same instance



Accuracy: SkaMPI vs TAU vs Score-P

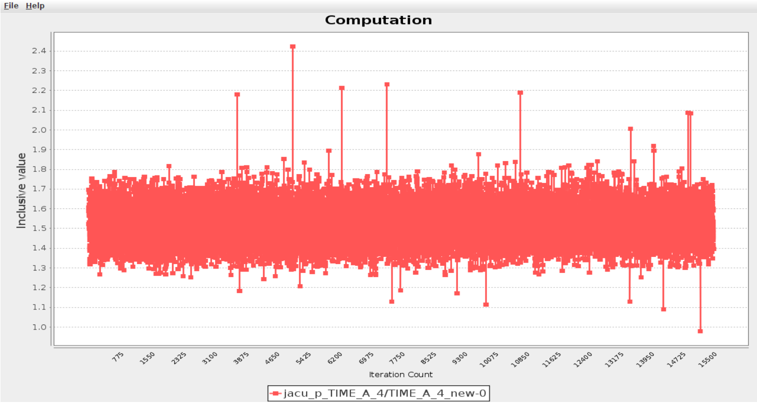


- Score-P provides less overhead compared to TAU

Decreasing the overhead of the instrumentation

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

```
BEGIN_FILE_EXCLUDE_LIST
*
END_FILE_EXCLUDE_LIST
```



Thank you!
Questions?