SC’16 Tutorial: 
Hands-on Practical Hybrid Parallel Application Performance Engineering

Christian Feld  
Jülich Supercomputing Centre

Markus Geimer  
Jülich Supercomputing Centre

Sameer Shende  
University of Oregon

Bert Wesarg  
TU Dresden

Brian Wylie  
Jülich Supercomputing Centre
# Agenda (morning)

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Introduction to VI-HPS &amp; parallel performance engineering</td>
<td>Wylie</td>
</tr>
<tr>
<td>09:15</td>
<td>Setup for hands-on exercises with Live-ISO/OVA &amp; Stampede</td>
<td>all</td>
</tr>
<tr>
<td>09:30</td>
<td>Instrumentation &amp; measurement of applications with <strong>Score-P</strong></td>
<td>Feld/Wesarg</td>
</tr>
<tr>
<td>10:00</td>
<td>Coffee break</td>
<td></td>
</tr>
<tr>
<td>10:30</td>
<td>Exploration &amp; visualization of call-path profiles with <strong>CUBE</strong></td>
<td>Geimer</td>
</tr>
<tr>
<td>11:00</td>
<td>Configuration &amp; customization of <strong>Score-P</strong> measurements</td>
<td>Feld/Wesarg</td>
</tr>
<tr>
<td>11:30</td>
<td>Examination &amp; visualization of profiles with <strong>TAU</strong></td>
<td>Shende</td>
</tr>
<tr>
<td>12:00</td>
<td>Lunch break</td>
<td></td>
</tr>
</tbody>
</table>
## Agenda (afternoon)

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:30</td>
<td>Automated analysis of traces for inefficiencies with <strong>Scalasca</strong></td>
<td>Geimer</td>
</tr>
<tr>
<td>14:15</td>
<td>Interactive visualization and time-interval statistics with <strong>Vampir</strong></td>
<td>Wesarg</td>
</tr>
<tr>
<td>15:00</td>
<td><strong>Coffee break</strong></td>
<td></td>
</tr>
<tr>
<td>15:30</td>
<td>Specialized <strong>Score-P</strong> measurements and analyses</td>
<td>Feld</td>
</tr>
<tr>
<td>16:00</td>
<td>Performance data management with <strong>TAU PerfExplorer</strong></td>
<td>Shende</td>
</tr>
<tr>
<td>16:15</td>
<td>Finding typical parallel performance bottlenecks</td>
<td>Wesarg</td>
</tr>
<tr>
<td>16:45</td>
<td>Review &amp; conclusion</td>
<td>Wylie</td>
</tr>
<tr>
<td>17:00</td>
<td><strong>Adjourn</strong></td>
<td></td>
</tr>
</tbody>
</table>
Virtual Institute – High Productivity Supercomputing

- **Goal**: Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems
- **Start-up funding (2006–2011)**
  by Helmholtz Association of German Research Centres
- **Activities**
  - Development and integration of HPC programming tools
    - Correctness checking & performance analysis
  - Academic workshops
  - Training workshops
  - Service
    - Support email lists
    - Application engagement

http://www.vi-hps.org
VI-HPS partners (founders)

Forschungszentrum Jülich
- Jülich Supercomputing Centre

RWTH Aachen University
- Centre for Computing & Communication

Technische Universität Dresden
- Centre for Information Services & HPC

University of Tennessee (Knoxville)
- Innovative Computing Laboratory
### VI-HPS partners (cont.)

<table>
<thead>
<tr>
<th>Barcelona Supercomputing Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centro Nacional de Supercomputación</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lawrence Livermore National Lab.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center for Applied Scientific Computing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical University of Darmstadt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laboratory for Parallel Programming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical University of Munich</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair for Computer Architecture</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Oregon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Research Laboratory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Stuttgart</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC Centre</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Versailles St-Quentin</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRC ITACA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allinea Software Ltd</th>
</tr>
</thead>
</table>

---

**SC'16 TUTORIAL: HANDS-ON PRACTICAL HYBRID PARALLEL APPLICATION PERFORMANCE ENGINEERING (SALT LAKE CITY, UT, NOVEMBER 14, 2016)**
Productivity tools

- MUST / ARCHER
  - MPI & OpenMP usage correctness checking
- PAPI
  - Interfacing to hardware performance counters
- Periscope Tuning Framework
  - Automatic analysis and tuning
- Scalasca
  - Large-scale parallel performance analysis
- TAU
  - Integrated parallel performance system
- Vampir
  - Interactive graphical trace visualization & analysis
- Score-P
  - Community-developed instrumentation & measurement infrastructure

For a brief overview of tools consult the VI-HPS Tools Guide:
Productivity tools (cont.)

- **DDT/MAP/PR**: Parallel debugging, profiling & performance reports
  - \textit{tut124}
- **Extra-P**: Automated performance modelling
  - \textit{tut130}
- **Kcachegrind**: Callgraph-based cache analysis [x86 only]
- **MAQAO**: Assembly instrumentation & optimization [x86-64 only]
- **mpiP/mpiPview**: MPI profiling tool and analysis viewer
- **Open MPI**: Integrated memory checking
- **Open|SpeedShop**: Integrated parallel performance analysis environment
  - \textit{tut161}
- **Paraver/Dimemas/Extrae**: Event tracing and graphical trace visualization & analysis
- **Rubik**: Process mapping generation & optimization [BG only]
- **SIONlib/Spindle**: Optimized native parallel file I/O & shared library loading
- **STAT**: Stack trace analysis tools
- **SysMon**: Batch system monitor plugin for Eclipse PTP
Technologies and their integration

- **KCACHEGRIND**
  - Visual trace analysis
  - Hardware monitoring

- **LWM2 / MAP / MPIP / O|SS / MAQAO / PR**
  - Automatic profile & trace analysis

- **PAPI**
  - Debugging, error & anomaly detection

- **MUST / ARCHER**
  - Execution

- **DDT**
  - Optimization

- **STAT**
  - VAMPIR
  - PARAVER

- **PERISCOPE**
  - SCALASCA

- **TAU**
  - SCORE-P

- **VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING**
  - **Techologies and their integration**

SC'16 TUTORIAL: HANDS-ON PRACTICAL HYBRID PARALLEL APPLICATION PERFORMANCE ENGINEERING (SALT LAKE CITY, UT, NOVEMBER 14, 2016)
Introduction to Parallel Performance Engineering

Brian Wylie
Jülich Supercomputing Centre

(with content used with permission from tutorials by Bernd Mohr/JSC and Luiz DeRose/Cray)
Performance: an old problem

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

Charles Babbage
1791 – 1871

Difference Engine
Today: the “free lunch” is over

- Moore's law is still in charge, but
  - Clock rates no longer increase
  - Performance gains only through increased parallelism
- Optimizations of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical networks / memory
    - More CPUs / multi-core

◊ Every doubling of scale reveals a new bottleneck!
Performance factors of parallel applications

■ “Sequential” performance factors
  ■ Computation
    ☐ Choose right algorithm, use optimizing compiler
  ■ Cache and memory
    ☐ Tough! Only limited tool support, hope compiler gets it right
  ■ Input / output
    ☐ Often not given enough attention

■ “Parallel” performance factors
  ■ Partitioning / decomposition
  ■ Communication (i.e., message passing)
  ■ Multithreading
  ■ Synchronization / locking
    ☐ More or less understood, good tool support
Tuning basics

■ Successful engineering is a combination of
  ■ Careful setting of various tuning parameters
  ■ The right algorithms and libraries
  ■ Compiler flags and directives
  ■ ...
  ■ Thinking !!!

■ Measurement is better than guessing
  ■ To determine performance bottlenecks
  ■ To compare alternatives
  ■ To validate tuning decisions and optimizations
    ✜ After each step!
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results

- Optimization
  - Modifications intended to eliminate/reduce performance problem
The 80/20 rule

- Programs typically spend 80% of their time in 20% of the code

- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  
  ⚫ Know when to stop!

- Don't optimize what does not matter
  
  ⚫ Make the common case fast!

“If you optimize everything, you will always be unhappy.”

Donald E. Knuth
Metrics of performance

What can be measured?

- A **count** of how often an event occurs
  - E.g., the number of MPI point-to-point messages sent
- The **duration** of some interval
  - E.g., the time spent these send calls
- The **size** of some parameter
  - E.g., the number of bytes transmitted by these calls

Derived metrics

- E.g., rates / throughput
- Needed for normalization
Example metrics

- Execution time
- Number of function calls
- CPI
  - CPU cycles per instruction
- FLOPS
  - Floating-point operations executed per second

“math” Operations?
HW Operations?
HW Instructions?
32-/64-bit? ...
Execution time

- **Wall-clock time**
  - Includes waiting time: I/O, memory, other system activities
  - In time-sharing environments also the time consumed by other applications

- **CPU time**
  - Time spent by the CPU to execute the application
  - Does not include time the program was context-switched out
    - Problem: Does not include inherent waiting time (e.g., I/O)
    - Problem: Portability? What is user, what is system time?

- **Problem: Execution time is non-deterministic**
  - Use mean or minimum of several runs
Inclusive vs. Exclusive values

- **Inclusive**
  - Information of all sub-elements aggregated into single value

- **Exclusive**
  - Information cannot be subdivided further
Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem
Sampling

- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

```
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

- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

```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**
  - Program is instrumented prior to execution

- **Dynamic instrumentation**
  - Program is instrumented at runtime

- **Code is inserted**
  - Manually
  - Automatically
    - By a preprocessor / source-to-source translation tool
    - By a compiler
    - By linking against a pre-instrumented library / runtime system
    - By binary-rewrite / dynamic instrumentation tool
Critical issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters

- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

-tradeoff: Accuracy vs. Expressiveness of data
Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem
Profiling / Runtime summarization

- Recording of aggregated information
  - Total, maximum, minimum, ...

- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters

- Over program and system entities
  - Functions, call sites, basic blocks, loops, ...
  - Processes, threads

 PROFILE = summarization of events over execution interval
Types of profiles

- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account

- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)

- Special-purpose profiles
  - Focus on specific aspects, e.g., MPI calls or OpenMP constructs
  - Comparing processes/threads
Tracing

- Recording detailed information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, ...)
  - Send / receive a message, ...
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, ...)
- Abstract execution model on level of defined events

\[ \text{Event trace} = \text{Chronologically ordered sequence of event records} \]
Process A

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

Process B

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

Event tracing

Local trace A

- 58 ENTER foo
- 62 SEND to B
- 64 EXIT foo
- ...

Local trace B

- ...
- 60 ENTER bar
- 68 RECV from A
- 69 EXIT bar
- ...

Global trace view

- ...
- 58 A ENTER foo
- 60 B ENTER bar
- 62 A SEND to B
- 64 A EXIT foo
- 68 B RECV from A
- 69 B EXIT bar
- ...

(Virtual merge)
Tracing Pros & Cons

- **Tracing advantages**
  - Event traces preserve the *temporal* and *spatial* relationships among individual events (context)
  - Allows reconstruction of *dynamic* application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces

- **Disadvantages**
  - Traces can very quickly become extremely large
  - Writing events to file at runtime may cause perturbation
Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem
Online analysis

- Performance data is processed during measurement run
  - Process-local profile aggregation
  - Requires formalized knowledge about performance bottlenecks
  - More sophisticated inter-process analysis using
    - “Piggyback” messages
    - Hierarchical network of analysis agents

- Online analysis often involves application steering to interrupt and re-configure the measurement
**Post-mortem analysis**

- Performance data is stored at end of measurement run
- Data analysis is performed afterwards
  - Automatic search for bottlenecks
  - Visual trace analysis
  - Calculation of statistics
Example: Time-line visualization

Global trace view

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER foo</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER bar</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND to B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT foo</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV from A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT bar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
No single solution is sufficient!

A combination of different methods, tools and techniques is typically needed!

- **Analysis**
  - Statistics, visualization, automatic analysis, data mining, ...

- **Measurement**
  - Sampling / instrumentation, profiling / tracing, ...

- **Instrumentation**
  - Source code / binary, manual / automatic, ...
Typical performance analysis procedure

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements

- What is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling

- Where is the key bottleneck?
  - Call-path profiling, detailed basic block profiling

- Why is it there?
  - Hardware counter analysis, trace selected parts to keep trace size manageable

- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function
Hands-on:
NPB-MZ-MPI / BT

VI-HPS Team
Tutorial exercise objectives

- Familiarize with usage of VI-HPS tools
  - Complementary tools’ capabilities & interoperability
- Prepare to apply tools productively to your application(s)
- Exercise is based on a small portable benchmark code
  - Unlikely to have significant optimization opportunities

Optional (recommended) exercise extensions
- Analyze performance of alternative configurations
- Investigate effectiveness of system-specific compiler/MPI optimizations and/or placement/binding/affinity capabilities
- Investigate scalability and analyze scalability limiters
- Compare performance on different HPC platforms
- ...
Access to Stampede

- Logging in to Stampede

  ```
  # Connect to the Stampede login node
  % ssh -Y userid@stampede.tacc.utexas.edu
  ```

- File systems & directories

  - Use `$SCRATCH` for the tutorial
  - Fast Lustre file system, 8.5 PB
  - No backup
  - Files may be automatically purged 10 days after last modification

- More extensive documentation:

  - [https://portal.tacc.utexas.edu/user-guides/stampede](https://portal.tacc.utexas.edu/user-guides/stampede)
Compiling & job submission

- Development environment: Intel compiler with Intel MPI
  - To cross-compile for the Xeon Phi (KNC), use `-mmic` compiler flag
  - Use Intel’s MPI compiler wrappers
    - mpiicc
    - mpiicpc
    - mpiifort

- Stampede uses the SLURM batch system
  - Jobs submitted from tutorial accounts will automatically be run in a reservation

```
% sbatch jobscript.sbatch
% squeue -u $USER
% scancel <jobid>
```

← Submit job
← View job queue
← Cancel job
Local installation

- VI-HPS tools not yet installed system-wide
  - Source provided shell code snippet to add local tool installations to $PATH
  - Required for each shell session

```% source ~/tg828282/Tutorial/vihps.sh
% cd $SCRATCH
% tar zxvf ~/tg828282/Tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI```

- Copy tutorial sources to your working directory, ideally on a parallel file system (recommended: $SCRATCH)

```% cd $SCRATCH
% tar zxvf ~/tg828282/Tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI```
NPB-MZ-MPI suite

- The NAS Parallel Benchmark suite (MPI+OpenMP version)
  - Available from http://www.nas.nasa.gov/Software/NPB
  - 3 benchmarks in Fortran77
  - Configurable for various sizes & classes
- Move into the NPB3.3-MZ-MPI root directory

```
% ls
bin/  common/  jobscript/  Makefile  README.install  SP-MZ/
BT-MZ/  config/  LU-MZ/     README    README.tutorial  sys/
```

- Subdirectories contain source code for each benchmark
  - Plus additional configuration and common code
- The provided distribution has already been configured for the tutorial, such that it is ready to “make” one or more of the benchmarks and install them into a (tool-specific) “bin” subdirectory
Building an NPB-MZ-MPI benchmark

To make a NAS multi-zone benchmark type

```
make <benchmark-name> CLASS=<class> NPROCS=<nprocs>
```

where `<benchmark-name>` is “bt-mz”, “lu-mz”, or “sp-mz”
`<class>` is “S”, “W”, “A” through “F”
`<nprocs>` is number of processes

[...]

* Custom build configuration is specified in config/make.def
* Suggested tutorial exercise configuration for Stampede:
  * `make bt-mz CLASS=B NPROCS=30`

Type “make” for instructions
Building an NPB-MZ-MPI benchmark

```bash
% make bt-mz CLASS=B NPROCS=30
make[1]: Entering directory `BT-MZ'
make[2]: Entering directory `sys'
icc -o setparams setparams.c -lm
make[2]: Leaving directory `sys'
..sys/setparams bt-mz 30 B
make[2]: Entering directory `../BT-MZ'
mpiifort -mmic -c -O3 -openmp bt.f
[...]
mpiifort -mmic -c -O3 -openmp mpi_setup.f
cd ../common; mpiifort -mmic -c -O3 -openmp print_results.f
cd ../common; mpiifort -mmic -c -O3 -openmp timers.f
mpiifort -mmic -O3 -openmp -o ../bin/bt-mz_B.30 bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_gbc.o solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o ../common/print_results.o
../common/timers.o
make[2]: Leaving directory `BT-MZ'
Built executable ../bin/bt-mz_B.30
make[1]: Leaving directory `BT-MZ'
```

- Specify the benchmark configuration
  - benchmark name: `bt-mz`, `lu-mz`, `sp-mz`
  - the number of MPI processes: `NPROCS=30`
  - the benchmark class (S, W, A, B, C, D, E): `CLASS=B`

Shortcut: `% make suite`
NPB-MZ-MPI / BT (Block Tridiagonal Solver)

- What does it do?
  - Solves a discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions
  - Performs 200 time-steps on a regular 3-dimensional grid
  - Implemented in 20 or so Fortran77 source modules

- Uses MPI & OpenMP in combination
  - Proposed hands-on setup on Stampede:
    - 2 compute nodes with 1 Xeon Phi accelerator each
    - 30 MPI processes with 4 OpenMP threads each running on the Xeon Phis only
    - bt-mz_B.30 should run in around 30 seconds
NPB-MZ-MPI / BT reference execution

% cd bin
% cp ../jobscript/stampede/reference_mic.sbatch .
% less reference_mic.sbatch
% sbatch reference_mic.sbatch
% less mzmpibt.o<job_id>

NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
Number of zones:  8 x  8
Iterations:   200   dt:   0.000300
Number of active processes:    30
Total number of threads:      120  (  4.0 threads/process)

Time step  1
Time step  20
[...]  
Time step 180
Time step 200
Verification Successful

BT-MZ Benchmark Completed.
Time in seconds = 32.84

- Copy jobsheet and launch as a hybrid MPI+OpenMP application

Hint: save the benchmark output (or note the run time) to be able to refer to it later
Tutorial exercise steps

- Edit `config/make.def` to adjust build configuration
  - Modify specification of compiler/linker: `MPIF77`
  - See next slide for details
- Make clean and build new tool-specific executable

```
% make clean
% make bt-mz CLASS=B NPROCS=30
Built executable ../bin.$(TOOL)/bt-mz_B.30
```

- Change to the directory containing the new executable before running it with the desired tool configuration

```
% cd bin.$(TOOL)
% cp ../jobscript/stampede/$(TOOL)_mic.sbatch .
% sbatch $(TOOL)_mic.sbatch
```
# SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS.

# Configured for generic MPI with INTEL compiler

#OPENMP = -fopenmp  # GCC compiler
OPENMP = -openmp       # Intel compiler

... 

# The Fortran compiler used for MPI programs

MPIF77 = mpiifort -mmic  # Intel compiler cross-compiling for Xeon Phi

# Alternative variant to perform instrumentation
#MPIF77 = scorep --user --static mpiifort -mmic

# PREP is a generic preposition macro for instrumentation preparation
#MPIF77 = $(PREP) mpiifort -mmic

...
Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes
Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)

- Collection of performance data
- Aggregation of performance data

- Calculation of metrics
- Identification of performance problems
- Presentation of results

- Modifications intended to eliminate/reduce performance problem

Preparation
Measurement
Analysis
Optimization
Fragmentation of tools landscape

- Several performance tools co-exist
  - Separate measurement systems and output formats
- 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
Score-P project idea

- Start a community effort for a common infrastructure
  - Score-P instrumentation and measurement system
  - Common data formats OTF2 and CUBE4

- Developer perspective:
  - Save manpower by sharing development resources
  - Invest in new analysis functionality and scalability
  - Save efforts for maintenance, testing, porting, support, training

- User perspective:
  - Single learning curve
  - Single installation, fewer version updates
  - Interoperability and data exchange

- Project funded by BMBF
- Close collaboration PRIMA project funded by DOE
Partners

- Forschungszentrum Jülich, Germany
- Gesellschaft für numerische Simulation mbH Braunschweig, Germany
- RWTH Aachen, Germany
- Technische Universität Darmstadt, Germany
- Technische Universität Dresden, Germany
- Technische Universität München, Germany
- University of Oregon, Eugene, USA
Design goals

- Functional requirements
  - Generation of call-path profiles and event traces
  - Using direct instrumentation and sampling
  - Flexible measurement without re-compilation
  - Recording time, visits, communication data, hardware counters
  - Access and reconfiguration also at runtime
  - Support for MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC and their valid combinations
  - Highly scalable I/O

- Non-functional requirements
  - Portability: all major HPC platforms
  - Scalability: petascale
  - Low measurement overhead
  - Robustness
  - Open Source: 3-clause BSD license
Score-P overview

- Vampir
- Scalasca
- CUBE
- TAU
- TAUdb
- Periscope

Score-P measurement infrastructure

- Event traces (OTF2)
- Call-path profiles (CUBE4, TAU)
- Online interface

Instrumentation wrapper

- Hardware counter (PAPI, rusage, PERF, plugins)
- Source code instrumentation (Compiler, PDT, User)
- Sampling interrupts (PAPI, PERF)

Application

- Process-level parallelism (MPI, SHMEM)
- Thread-level parallelism (OpenMP, Pthreads)
- Accelerator-based parallelism (CUDA, OpenCL, OpenACC)
Future features and management

- Scalability to maximum available CPU core count
- Support for binary instrumentation
- Support for new programming models, e.g., PGAS
- Support for new architectures

- Ensure a single official release version at all times which will always work with the tools
- Allow experimental versions for new features or research

- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
Hands-on:
NPB-MZ-MPI / BT

Score-P
Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
  - 1.1 Summary measurement collection
  - 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
  - 2.1 Summary measurement collection with filtering
  - 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
  - 3.1 Event trace examination & analysis
Recap: Local installation

- VI-HPS tools not yet installed system-wide
  - Source provided shell code snippet to add local tool installations to $PATH
  - Required for each shell session

```
% source ~tg828282/Tutorial/vihps.sh
```

- Copy tutorial sources to your working directory, ideally on a parallel file system (recommended: $SCRATCH)

```
% cd $SCRATCH
% tar zxvf ~tg828282/Tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI
```
NPB-MZ-MPI / BT instrumentation

```
#---------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------
#MPIF77 = mpiifort -mmic

# Alternative variants to perform instrumentation
...
MPIF77 = scorep --user --static mpiifort -mmic

# This links MPI Fortran programs; usually the same as ${MPIFLINK}
FLINK   = $(MPIF77)
...
```

- Edit config/make.def to adjust build configuration
- Modify specification of compiler/linker: MPIF77

Uncomment the Score-P compiler wrapper specification
NPB-MZ-MPI / BT instrumented build

```
% make clean

% make bt-mz CLASS=B NPROCS=30
cd BT-MZ; make CLASS=B NPROCS=30 VERSION=
make: Entering directory 'BT-MZ'
cd ..;/sys; icc -o setparams setparams.c -lm
..;/sys/setparams bt-mz 30 B
scorep --user --static mpiifort -mmic -c -O3 -openmp bt.f
[...]
cd ..;/common; scorep --user --static mpiifort -mmic -c -O3 
-openmp timers.f
scorep --user --static mpiifort -mmic -O3 -openmp 
-o ..;/bin.scorep/bt-mz_B.30 
bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o 
adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o 
solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o 
..;/common/print_results.o ..;/common/timers.o
Built executable ..;/bin.scorep/bt-mz_B.30
make: Leaving directory 'BT-MZ'
```

- Return to root directory and clean-up
- Re-build executable using Score-P compiler wrapper
Measurement configuration: scorep-info

% scorep-info config-vars --full
SCOREP_ENABLE_PROFILING
  Description: Enable profiling
  [...]  
SCOREP_ENABLE_TRACING
  Description: Enable tracing
  [...]  
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
  [...]  
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
  [...]  
SCOREP_FILTERING_FILE
  Description: A file name which contain the filter rules
  [...]  
SCOREP_METRIC_PAPI
  Description: PAPI metric names to measure
  [...]  
SCOREP_METRIC_RUSAGE
  Description: Resource usage metric names to measure
  [... More configuration variables ...]

- Score-P measurements are configured via environmental variables
Summary measurement collection

- Change to the directory containing the new executable before running it with the desired configuration
- Check settings
- Submit job

```
% cd bin.scorep
% cp ..:/jobs/script/stampede/scorep_mic.sbatch .
% vim scorep_mic.sbatch
export MIC_PPN=15
export MIC_OMP_NUM_THREADS=4
PROCS=30
CLASS=B
export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum
#export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=32M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC
#export SCOREP_ENABLE_TRACING=true

# launch
ibrun.symm -m ./bt-mz_${CLASS}.${PROCS}

% sbatch ./scorep_mic.sbatch
```

Leave these lines commented out for the moment
Summary measurement collection

% less mzmpibt.o<job_id>

NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP

> Benchmark

Number of zones: 8 x 8
Iterations: 200 dt: 0.000300
Number of active processes: 30

Use the default load factors with threads
Total number of threads: 120 (4.0 threads/process)

Calculated speedup = 71.69

Time step 1

[... More application output ...]
BT-MZ summary analysis report examination

- Creates experiment directory including:
  - A record of the measurement configuration (scorep.cfg)
  - The analysis report that was collated after measurement (profile.cubex)

- Interactive exploration with Cube:

  ```
  ls
  bt-mz_B.30 mzmpibt.e<job_id> mzmpibt.o<job_id>
  scorep_bt-mz_sum
  ls scorep_bt-mz_sum
  profile.cubex scorep.cfg
  ```

  ```
  cube scorep_bt-mz_sum/profile.cubex
  ```

  [CUBE GUI showing summary analysis report]

  **Hint:**
  Copy ‘profile.cubex’ to Live-DVD environment using ‘scp’ to improve responsiveness of GUI
Analysis report examination with Cube

Markus Geimer
Jülich Supercomputing Centre
Cube

- Parallel program analysis report exploration tools
  - Libraries for XML+binary report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
    - Requires Qt4 ≥4.6 or Qt5

- Originally developed as part of the Scalasca toolset

- Now available as a separate component
  - Can be installed independently of Score-P, e.g., on laptop or desktop
  - Latest release: Cube 4.3.4 (April 2016)
Analysis presentation and exploration

- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call path (program location)
  - System location (process/thread)

- Three coupled tree browsers

- Cube displays severities
  - As value: for precise comparison
  - As color: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display modes
### Analysis presentation

<table>
<thead>
<tr>
<th>Metric tree</th>
<th>Call tree</th>
<th>Flat view</th>
<th>System tree</th>
<th>Box Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.63e9 Visits</td>
<td>0.01 MAIN_</td>
<td>0.04 adi_</td>
<td>- generic cluster</td>
<td></td>
</tr>
<tr>
<td>767.48 Time</td>
<td>0.82 mpi_setup_</td>
<td>39.91 compute_rhs_</td>
<td>- i06r01c20</td>
<td></td>
</tr>
<tr>
<td>0.00 Minimum Inclusive Time</td>
<td>0.00 MPI_Bcast</td>
<td>233.49 x_solve_</td>
<td>- MPI Rank 0</td>
<td></td>
</tr>
<tr>
<td>48.58 Maximum Inclusive Time</td>
<td>0.00 env_setup_</td>
<td>239.34 y_solve_</td>
<td>3.81 CPU thread 0</td>
<td></td>
</tr>
<tr>
<td>5.27e8 bytes_sent</td>
<td>0.00 zone_setup_</td>
<td>0.07 z_solve_</td>
<td>3.70 CPU thread 1</td>
<td></td>
</tr>
<tr>
<td>5.27e8 bytes_received</td>
<td>0.00 map_zones_</td>
<td>0.04 !omp parallel @z_solve.f43</td>
<td>3.64 CPU thread 2</td>
<td></td>
</tr>
</tbody>
</table>

**Questions:**

- **What kind of performance metric?**
- **Where is it in the source code?**
- **In what context?**
- **How is it distributed across the processes/threads?**
Inclusive vs. exclusive values

- Inclusive
  - Information of all sub-elements aggregated into single value
- Exclusive
  - Information cannot be subdivided further

```c
int foo()
{
  int a;
  a = 1 + 1;
  bar();
  a = a + 1;
  return a;
}
```
Score-P analysis report exploration (opening view)
Metric selection

Selecting the “Time” metric shows total execution time.
Expanding the system tree

Distribution of selected metric for call path by process/thread
Expanding the call tree

Distribution of selected metric across the call tree

Collapsed: inclusive value
Expanded: exclusive value
Selecting a call path

![Selection updates metric values shown in columns to the right]
Source-code view via context menu

Right-click opens context menu

Shows the source code of the clicked item
Note: This feature depends on file and line number information provided by the instrumentation, i.e., it may not always be available.
Flat profile view

Select flat view tab, expand all nodes, and sort by exclusive value.
Box plot view

Box plot shows distribution across the system; with min/max/avg/median/quartiles.
Alternative display modes

Data can be shown in various percentage modes.
Important display modes

- **Absolute**
  - Absolute value shown in seconds/bytes/counts

- **Selection percent**
  - Value shown as percentage w.r.t. the selected node
  - “on the left” (metric/call path)

- **Peer percent (system tree only)**
  - Value shown as percentage relative to the maximum peer value
Multiple selection

Select multiple nodes with Ctrl-click
Context-sensitive help available for all GUI items
Derived metrics

- Derived metrics are defined using CubePL expressions, e.g.:
  \[
  \text{metric::time(i)}/\text{metric::visits(e)}
  \]

- Values of derived metrics are not stored, but calculated on-the-fly

- Types of derived metrics:
  - Prederived: evaluation of the CubePL expression is performed before aggregation
  - Postderived: evaluation of the CubePL expression is performed after aggregation

- Examples:
  - “Average execution time”: Postderived metric with expression
    \[
    \text{metric::time(i)}/\text{metric::visits(e)}
    \]
  - “Number of FLOP per second”: Postderived metric with expression
    \[
    \text{metric::FLOP()}/\text{metric::time()}
    \]
Derived metrics in Cube GUI

- Collection of derived metrics
- Parameters of the derived metric
- CubePL expression
Example: FLOPS based on PAPI_FP_OPS and time
CUBE algebra utilities

- Extracting solver sub-tree from analysis report

```bash
% cube_cut -r '<<ITERATION>>' scorep_bt-mz_B_mic15p30x4_sum/profile.cubex
Writing cut.cubex... done.
```

- Calculating difference of two reports

```bash
% cube_diff scorep_bt-mz_B_mic15p30x4_sum/profile.cubex cut.cubex
Writing diff.cubex... done.
```

- Additional utilities for merging, calculating mean, etc.
- Default output of cube_utility is a new report utility.cubex
- Further utilities for report scoring & statistics
- Run utility with `-h` (or no arguments) for brief usage info
Iteration profiling

- Show time dependent behavior by “unrolling” iterations

- Preparations:
  - Mark loop body by using Score-P instrumentation API in your source code
    
    ```
    SCOREP_USER_REGION_DEFINE( scorep_bt_loop )
    SCOREP_USER_REGION_BEGIN( scorep_bt_loop, "<<bt_iter>>", SCOREP_USER_REGION_TYPE_DYNAMIC )
    SCOREP_USER_REGION_END( scorep_bt_loop )
    ```

- Result in the Cube profile:
  - Iterations shown as separate call trees
  - Useful for checking results for specific iterations
    - or
  - Select your user-instrumented region and mark it as loop
  - Choose “Hide iterations”
  - View the Barplot statistics or the (thread x iterations) Heatmap
Iteration profiling: Barplot

![Aggregation selection]

![Iterations]
Iteration profiling: Heatmap

![Heatmap Diagram]

The diagram shows the execution time breakdown for iterations and threads. The Heatmap visualizes the performance of different parts of the application over various iterations.
Cube: Further information

- Parallel program analysis report exploration tools
  - Libraries for Cube report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration

- Available under 3-clause BSD open-source license

- Documentation & sources:
  - http://www.scalasca.org

- User guide also part of installation:
  - `cube-config --cube-dir`/share/doc/CubeGuide.pdf

- Contact:
  - mailto: scalasca@fz-juelich.de
Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes
Congratulations!? 

- If you made it this far, you successfully used Score-P to
  - instrument the application
  - analyze its execution with a summary measurement, and
  - examine it with one the interactive analysis report explorer GUIs
- ... revealing the call-path profile annotated with
  - the “Time” metric
  - Visit counts
  - MPI message statistics (bytes sent/received)
- ... but how *good* was the measurement?
  - The measured execution produced the desired valid result
  - however, the execution took rather longer than expected!
    - even when ignoring measurement start-up/completion, therefore
    - it was probably dilated by instrumentation/measurement overhead
Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
  - 1.1 Summary measurement collection
  - 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
  - 2.1 Summary measurement collection with filtering
  - 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
  - 3.1 Event trace examination & analysis
BT-MZ summary analysis result scoring

- Report scoring as textual output

- Region/callpath classification
  - **MPI** pure MPI functions
  - **OMP** pure OpenMP regions
  - **USR** user-level computation
  - **COM** “combined” USR+OpenMP/MPI
  - **ANY/ALL** aggregate of all region types

% scorep-score scorep_bt-mz_sum/profile.cubex

Estimated aggregate size of event trace:
Estimated requirements for largest trace buffer (max_buf):
Estimated memory requirements (SCOREP_TOTAL_MEMORY):
(hint: When tracing set SCOREP_TOTAL_MEMORY=2373MB to avoid intermediate flushes or reduce requirements using USR regions filters.)

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>2,479,514,724</td>
<td>1,634,202,275</td>
<td>9763.89</td>
<td>100.0</td>
<td>5.97</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>2,477,923,448</td>
<td>1,631,143,401</td>
<td>3837.81</td>
<td>39.3</td>
<td>2.35</td>
<td>USR</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>4,129,716</td>
<td>2,743,808</td>
<td>4414.15</td>
<td>45.2</td>
<td>1608.77</td>
<td>OMP</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>372,430</td>
<td>128,436</td>
<td>1499.04</td>
<td>15.4</td>
<td>11671.49</td>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>225,290</td>
<td>186,630</td>
<td>12.90</td>
<td>0.1</td>
<td>69.13</td>
<td>COM</td>
<td></td>
</tr>
</tbody>
</table>

40 GB total memory
2.3 GB per rank!
BT-MZ summary analysis report breakdown

```bash
% scorep-score -r scorep_bt-mz_sum/profile.cubex

[...]

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>2,479,514,724</td>
<td>1,634,202,275</td>
<td>9763.89</td>
<td>100.0</td>
<td>5.97</td>
<td>ALL</td>
</tr>
<tr>
<td>USR</td>
<td>2,477,923,448</td>
<td>1,631,143,401</td>
<td>3837.81</td>
<td>39.3</td>
<td>2.35</td>
<td>USR</td>
</tr>
<tr>
<td>OMP</td>
<td>4,129,716</td>
<td>2,743,808</td>
<td>4414.15</td>
<td>45.2</td>
<td>1608.77</td>
<td>OMP</td>
</tr>
<tr>
<td>MPI</td>
<td>372,430</td>
<td>128,436</td>
<td>1499.04</td>
<td>15.4</td>
<td>11671.49</td>
<td>MPI</td>
</tr>
<tr>
<td>COM</td>
<td>225,290</td>
<td>186,630</td>
<td>12.90</td>
<td>0.1</td>
<td>69.13</td>
<td>COM</td>
</tr>
</tbody>
</table>

USR 800,074,470 522,844,416 970.09 9.9 1.86 matvec_sub_
USR 800,074,470 522,844,416 1064.74 10.9 2.04 matmul_sub_
USR 800,074,470 522,844,416 1651.75 16.9 3.16 binvcrhs_
USR 26,365,170 22,692,096 56.39 0.6 2.49 binvrhs_
USR 26,365,170 22,692,096 63.45 0.6 2.80 lhsinit_
USR 24,964,368 17,219,840 31.35 0.3 1.82 exact_solution

More than 2.3 GB just for these 6 regions
BT-MZ summary analysis score

- Summary measurement analysis score reveals
  - Total size of event trace would be ~40 GB
  - Maximum trace buffer size would be ~2.3 GB per rank
    - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
  - 99.9% of the trace requirements are for USR regions
    - purely computational routines never found on COM call-paths common to communication routines or OpenMP parallel regions
  - These USR regions contribute around 39% of total time
    - however, much of that is very likely to be measurement overhead for frequently-executed small routines
- Advisable to tune measurement configuration
  - Specify an adequate trace buffer size
  - Specify a filter file listing (USR) regions not to be measured
BT-MZ summary analysis report filtering

- Report scoring with prospective filter listing 6 USR regions

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
binvcrhs*
matmul_sub*
matvec_sub*
exact_solution*
binvrhs*
lhs*init*
timer_*

% scorep-score -f ../config/scorep.filt -c 2 \scorep_bt-mz_sum/profile.cubex
```

Estimated aggregate size of event trace: 242 MB
Estimated requirements for largest trace buffer (max_buf): 12 MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 20 MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=20MB to avoid >intermediate flushes or reduce requirements using USR regions filters.)

242 MB of memory in total, 20 MB per rank!
(Including 2 metric values)
**BT-MZ summary analysis report filtering**

```bash
% scorep-score -r -f ../config/scorep.filt ./scorep_bt-mz_sum/profile.cubex
```

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>2,479,514,724</td>
<td>1,634,202,275</td>
<td>9763.89</td>
<td>100.0</td>
<td>5.97</td>
<td>ALL</td>
</tr>
<tr>
<td>USR</td>
<td>2,477,923,448</td>
<td>1,631,143,401</td>
<td>3837.81</td>
<td>39.3</td>
<td>2.35</td>
<td>USR</td>
</tr>
<tr>
<td>OMP</td>
<td>4,129,716</td>
<td>2,743,808</td>
<td>4414.15</td>
<td>15.4</td>
<td>1608.77</td>
<td>OMP</td>
</tr>
<tr>
<td>MPI</td>
<td>372,430</td>
<td>128,436</td>
<td>1499.04</td>
<td>15.4</td>
<td>1671.49</td>
<td>MPI</td>
</tr>
<tr>
<td>COM</td>
<td>225,290</td>
<td>186,630</td>
<td>12.90</td>
<td>0.1</td>
<td>69.13</td>
<td>COM</td>
</tr>
</tbody>
</table>

* ALL     | 4,732,090     | 3,064,245    | 5926.12 | 60.7    | 1933.96        | ALL-FLT      |
+ FLT     | 2,477,918,768 | 1,631,138,030| 3837.78 | 39.3    | 2.35           | FLT          |
- OMP     | 4,129,716     | 2,743,808    | 4414.15 | 15.4    | 1671.49        | OMP-FLT      |
- MPI     | 372,430       | 128,436      | 1499.04 | 15.4    | 11671.49       | MPI-FLT      |
* COM     | 225,290       | 186,630      | 12.90   | 0.1     | 69.13          | COM-FLT      |
* USR     | 4,680         | 5,371        | 0.03    | 0.0     | 5.78           | USR-FLT      |
+ USR     | 800,074,470   | 522,844,416  | 970.09  | 9.9     | 1.86           | matvec_sub_  |
+ USR     | 800,074,470   | 522,844,416  | 1064.74 | 10.9    | 2.04           | matmul_sub_  |
+ USR     | 800,074,470   | 522,844,416  | 1651.75 | 16.9    | 3.16           | binvcrhs_    |
+ USR     | 26,365,170    | 22,692,096   | 56.39   | 0.6     | 2.49           | binvrhs_     |
+ USR     | 26,365,170    | 22,692,096   | 63.45   | 0.6     | 2.80           | lhsinit_     |
+ USR     | 24,964,368    | 17,219,840   | 31.35   | 0.3     | 1.82           | exact_solution_ |

- Score report breakdown by region

Filtered routines marked with ‘+’
BT-MZ filtered summary measurement

- Set new experiment directory and re-run measurement with new filter configuration
- Submit job
Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods)
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under New BSD open-source license
- Documentation & Sources:
  - [http://www.score-p.org](http://www.score-p.org)
  - User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date
Examination and Visualization of profiles with TAU

Sameer Shende
sameer@cs.uoregon.edu
University of Oregon
http://tau.uoregon.edu
TAU Performance System®

- Parallel performance framework and toolkit
  - Supports all HPC platforms, compilers, runtime system
  - Provides portable instrumentation, measurement, analysis
TAU Performance System

- Instrumentation
  - Fortran, C++, C, UPC, Java, Python, Chapel
  - Automatic instrumentation

- Measurement and analysis support
  - MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - pthreads, OpenMP, OMPT interface, hybrid, other thread models
  - GPU, CUDA, OpenCL, OpenACC
  - Parallel profiling and tracing
  - Use of Score-P for native OTF2 and CUBEX generation
  - Efficient callpath profiles and trace generation using Score-P

- Analysis
  - Parallel profile analysis (ParaProf), data mining (PerfExplorer)
  - Performance database technology (TAUdb)
  - 3D profile browser
TAU Performance System

- Instrumentation
  - Fortran, C++, C, UPC, Java, Python, Chapel
  - Automatic instrumentation

- Measurement and analysis support
  - MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - pthreads, OpenMP, OMPT interface, hybrid, other thread models
  - GPU, CUDA, OpenCL, OpenACC
  - Parallel profiling and tracing
  - Use of Score-P for native OTF2 and CUBEX generation
  - Efficient callpath profiles and trace generation using Score-P

- Analysis
  - Parallel profile analysis (ParaProf), data mining (PerfExplorer)
  - Performance database technology (TAUdb)
  - 3D profile browser
TAU Performance System

- TAU supports both sampling and direct instrumentation
- Memory debugging as well as I/O performance evaluation
- Profiling as well as tracing
- Interfaces with Score-P for more efficient measurements
- TAU’s instrumentation covers:
  - Runtime library interposition (tau_exec)
  - Compiler-based instrumentation
  - PDT based Source level instrumentation: routine & loop
  - Event based sampling (TAU_SAMPLING=1 or tau_exec -ebs)
  - Callstack unwinding with sampling (TAU_EBS_UNWIND=1)
  - OpenMP Tools Interface (OMPT, tau_exec -T ompt)
  - CUDA CUPTI, OpenCL (tau_exec -T cupti -cupti)
Application Performance Engineering using TAU

• How much time is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*? What is the time spent in OpenMP loops?

• How many instructions are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken? What is the extent of vectorization for loops on Intel MIC?

• What is the memory usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks? What is the memory footprint of the application? What is the memory high water mark?

• How much energy does the application use in Joules? What is the peak power usage?

• What are the I/O characteristics of the code? What is the peak read and write *bandwidth* of individual calls, total volume?

• What is the contribution of each *phase* of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?

• How does the application *scale*? What is the efficiency, runtime breakdown of performance across different core counts?
Using TAU

- TAU supports several compilers, measurement, and thread options
  - Intel compilers, profiling with hardware counters using PAPI, MPI library, CUDA...
  - Each measurement configuration of TAU corresponds to a unique stub makefile (configuration file) and library that is generated when you configure it
- To instrument source code automatically using PDT
  - Choose an appropriate TAU stub makefile in `<arch>/lib`:
    - `% module load tau`
    - `% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari`
    - `% export TAU_OPTIONS=‘-optVerbose …’ (see tau_compiler.sh)`
    - Use `tau_f90.sh`, `tau_cxx.sh`, `tau_upc.sh`, or `tau_cc.sh` as F90, C++, UPC, or C compilers respectively:
      - `% mpiif90 foo.f90` changes to
      - `% tau_f90.sh foo.f90`
- Set runtime environment variables, execute application and analyze performance data:
  - `% pprof` (for text based profile display)
  - `% paraprof` (for GUI)
Installing and Configuring TAU

- Installing PDT:
  - `wget http://tau.uoregon.edu/pdt_lite.tgz`
  - `./configure --prefix=<dir>; make; make install`

- Installing TAU:
  - `wget http://tau.uoregon.edu/tau.tgz`
  - `./configure --scorep=download --arch=x86_64 -bfd=download -pdt=<dir> -papi=<dir> ...`
  - For MIC (KNC):
    - `./configure --scorep=download --arch=mic_linux -pdt=<dir> -pdt_c++=g++ -papi=dir ...`
  - `make install`

- Using TAU:
  - `export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-<TAGS>`
  - `make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh`
Different Makefiles for TAU Compiler and Runtime Options

% . /home1/03529/tg828282/Tutorial/vihps.sh
% export TAU=/home1/03529/tg828282/Software/tau_latest/mic_linux/lib;
ls $TAU/Makefile.*
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp

- For an MPI+OpenMP+F90 application with Intel MPI, you may choose
  Makefile.tau-icpc-papi-mpi-pdt
    - Supports MPI instrumentation & PDT for automatic source instrumentation
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
% tau_f90.sh matmult.f90 -o matmult
% mpirun -np 256 ./matmult
% paraprof

For x86_64:
% ls /home1/03529/tg828282/Software/tau_latest/x86_64/lib/Makefile*
Compile-Time Options

Optional parameters for the TAU_OPTIONS environment variable:

% tau_compiler.sh

- optVerbose
   Turn on verbose debugging messages

- optCompInst
   Use compiler based instrumentation

- optNoCompInst
   Do not revert to compiler instrumentation if source instrumentation fails.

- optTrackIO
   Wrap POSIX I/O call and calculates vol/bw of I/O operations (configure TAU with –iowrapper)

- optTrackGOMP
   Enable tracking GNU OpenMP runtime layer (used without –opari)

- optMemDbg
   Enable runtime bounds checking (see TAU_MEMDBG_* env vars)

- optKeepFiles
   Does not remove intermediate .pdb and .inst.* files

- optPreProcess
   Preprocess sources (OpenMP, Fortran) before instrumentation

- optTauSelectFile="<file>"
   Specify selective instrumentation file for tau_instrumentor

- optTauWrapFile="<file>"
   Specify path to link_options.tau generated by tau_gen_wrapper

- optHeaderInst
   Enable Instrumentation of headers

- optTrackUPCR
   Track UPC runtime layer routines (used with tau_upc.sh)

- optLinking=""
   Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)

- optCompile=""
   Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

- optPdtF95Opts=""
   Add options for Fortran parser in PDT (f95parse/gfparsel) …
Compile-Time Options (contd.)

- Optional parameters for the TAU_OPTIONS environment variable:

  % tau_compiler.sh

  -optMICOffload         Links code for Intel MIC offloading, requires both host and
                         MIC TAU libraries
  -optShared             Use TAU’s shared library (libTAU.so) instead of static library (default)
  -optPdtCxxOpts=""       Options for C++ parser in PDT (cxxparse).
  -optPdtF90Parser=""     Specify a different Fortran parser
  -optPdtCleanscapeParser Specify the Cleanscape Fortran parser instead of GNU gfparser
  -optTau=""             Specify options to the tau_instrumentor
  -optTrackDMAPP         Enable instrumentation of low-level DMAPP API calls on Cray
  -optTrackPthread       Enable instrumentation of pthread calls

  See tau_compiler.sh for a full list of TAU_OPTIONS.
Compiling Fortran Codes with TAU

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:
  % export TAU_OPTIONS=’-optPdtF95Opts=-R free’ -optVerbose’

- To use the compiler based instrumentation instead of PDT (source-based):
  % export TAU_OPTIONS=’-optCompInst -optVerbose’

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):
  % export TAU_OPTIONS=’-optPreProcess -optVerbose -optDetectMemoryLeaks’

- To use an instrumentation specification file:
  % export TAU_OPTIONS=’-optTauSelectFile=select.tau -optVerbose -optPreProcess’
  % cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine=“#”
  # this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.
  END_INSTRUMENT_SECTION
# Runtime Environment Variables

<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_FOOTPRINT</td>
<td>0</td>
<td>Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high water mark of memory usage</td>
</tr>
<tr>
<td>TAU_TRACK_POWER</td>
<td>0</td>
<td>Tracks power usage by sampling periodically.</td>
</tr>
<tr>
<td>TAU_CALLPATH_DEPTH</td>
<td>2</td>
<td>Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)</td>
</tr>
<tr>
<td>TAU_SAMPLING</td>
<td>1</td>
<td>Setting to 1 enables event-based sampling.</td>
</tr>
<tr>
<td>TAU_TRACK_SIGNALS</td>
<td>0</td>
<td>Setting to 1 generate debugging callstack info when a program crashes</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. Throttles 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 per call</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 xml format</td>
</tr>
<tr>
<td>TAU_METRICS</td>
<td>TIME</td>
<td>Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_&lt;event&gt;:&lt;subevent&gt;)</td>
</tr>
</tbody>
</table>
## Runtime Environment Variables (contd.)

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACK_MEMORY_LEAKS</td>
<td>0</td>
<td>Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_EBS_SOURCE</td>
<td>TIME</td>
<td>Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1)</td>
</tr>
<tr>
<td>TAU_EBS_PERIOD</td>
<td>100000</td>
<td>Specifies the overflow count for interrupts</td>
</tr>
<tr>
<td>TAU_MEMDBG_ALLOC_MIN/MAX</td>
<td>0</td>
<td>Byte size minimum and maximum subject to bounds checking (used with TAU_MEMDBG_PROTECT_*)</td>
</tr>
<tr>
<td>TAU_MEMDBG_OVERHEAD</td>
<td>0</td>
<td>Specifies the number of bytes for TAU’s memory overhead for memory debugging.</td>
</tr>
<tr>
<td>TAU_MEMDBG_PROTECT_BELOW/ABOVE</td>
<td>0</td>
<td>Setting to 1 enables tracking runtime bounds checking below or above the array bounds (requires –optMemDbg while building or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_MEMDBG_ZERO_MALLOC</td>
<td>0</td>
<td>Setting to 1 enables tracking zero byte allocations as invalid memory allocations.</td>
</tr>
<tr>
<td>TAU_MEMDBG_PROTECT_FREE</td>
<td>0</td>
<td>Setting to 1 detects invalid accesses to deallocated memory that should not be referenced until it is reallocated (requires –optMemDbg or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_MEMDBG_ATTEMPT_CONTINUE</td>
<td>0</td>
<td>Setting to 1 allows TAU to record and continue execution when a memory error occurs at runtime.</td>
</tr>
<tr>
<td>TAU_MEMDBG_FILL_GAP</td>
<td>Undefined</td>
<td>Initial value for gap bytes</td>
</tr>
<tr>
<td>TAU_MEMDBG_ALIGNMENT</td>
<td>Sizeof(int)</td>
<td>Byte alignment for memory allocations</td>
</tr>
<tr>
<td>TAU_EVENT_THRESHOLD</td>
<td>0.5</td>
<td>Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max</td>
</tr>
</tbody>
</table>
Simplifying TAU’s usage (tau_exec)

- **Uninstrumented execution**
  - `% mpirun -np 4 ./a.out`  
- **Track MPI performance**
  - `% mpirun -np 4 tau_exec ./a.out`
- **Track POSIX I/O and MPI performance (MPI enabled by default)**
  - `% mpirun -np 4 tau_exec -T mpi,pdt -io ./a.out`
- **Track memory operations**
  - `% export TAU_TRACK_MEMORY_LEAKS=1`
  - `% mpirun -np 8 tau_exec -memory_debug ./a.out` (bounds check)
- **Use event based sampling (compile with -g)**
  - `% mpirun -np 8 tau_exec -ebs ./a.out`
  - Also `-ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>`
- **Load wrapper interposition library**
  - `% mpirun -np 8 tau_exec -loadlib=<path/libwrapper.so> ./a.out`
- **Track GPGPU operations**
  - `% mpirun -np 8 tau_exec -cupti ./a.out`
  - `% mpirun -np 8 tau_exec -opencl ./a.out`
  - `% mpirun -np 8 tau_exec -openacc ./a.out`
Binary Rewriting Instrumentation

- Support for both static and dynamic executables
- Specify a list of routines to instrument
- Specify the TAU measurement library to be injected
- MAQAO [UVSQ, Intel Exascale Labs]:
  
  \%
  tau_rewrite -T [tags] a.out -o a.inst

- DyninstAPI [U. Maryland and U. Wisconsin, Madison]:
  
  \%
  tau_run -T [tags] a.out -o a.inst

- Pebil [UC San Diego]:
  
  \%
  tau_pebil_rewrite -T [tags] a.out -o a.inst

- Execute the application to get measurement data:
  
  \%
  mpirun -np 256 ./a.inst
TAU Analysis

![Diagram of TAU Analysis process]

1. **Instrumentation**: Selection of events to measure.
2. **Measurement**: Collecting event information.
3. **Analysis**: Profiles and traces.
   - **Profile Data Management (PerfDMF)**: Profile translators, Metadata (XML), profile database.
   - **Profile Analysis (ParaProf)**: Tools for visual analysis.
   - **Profile Data Mining (PerfExplorer)**: Tools for data mining.
4. **Trace Data Management**: Trace translators, trace storage.
5. **Trace Visualizers**: Vampir, JumpShot, Paraver.
6. **Trace Analyzers**: Expert, ProfileGen, Vampir Server.

TAU Portal connects to the various components, facilitating the process from data collection to analysis.
ParaProf Profile Analysis Framework

[Diagram showing the flow from Performance Data to ParaProf with stages of Parsers and Importers, Basic Analysis and Derived Data, Internal Representation, Profile Data, and Visualization with Scripting Interface and Jython.]
Parallel Profile Visualization: ParaProf
ParaProf 3D Communication Matrix

% export TAU_COMM_MATRIX=1
TAU tutorial exercise objectives

- Familiarise with usage of TAU tools
  - complementary tools’ capabilities & interoperability
- Prepare to apply tools productively to your applications(s)
- Exercise is based on a small portable benchmark code
  - unlikely to have significant optimisation opportunities

- Optional (recommended) exercise extensions
  - analyse performance of alternative configurations
  - investigate effectiveness of system-specific compiler/MPI optimisations and/or placement/binding/affinity capabilities
  - investigate scalability and analyse scalability limiters
  - compare performance on different HPC platforms
  - ...
Local Installation (Stampede, TACC)

- Setup preferred program environment compilers
  - Default set Intel Compilers with Intel MPI
  - Generate profile files using Score-P

```
% . /home1/03529/tg828282/Tutorial/vihps.sh
% paraprof profile.cubex &
```

For PerfExplorer:

```
% wget http://tau.uoregon.edu/data.tgz; tar zxf data.tgz; cd data
% cat README
And follow the steps
```
NPB-MZ-MPI Suite

- The NAS Parallel Benchmark suite (MPI+OpenMP version)
  - Available from: http://www.nas.nasa.gov/Software/NPB
- 3 benchmarks in Fortran77
- Configurable for various sizes & classes
- Move into the NPB3.3-MZ-MPI root directory

```
% ls
bin/    common/  jobscript/  Makefile  README.install  SP-MZ/
BT-MZ/  config/  LU-MZ/      README    README.tutorial  sys/
```

- Subdirectories contain source code for each benchmark
  - plus additional configuration and common code
- The provided distribution has already been configured for the tutorial, such that it's ready to “make” one or more of the benchmarks and install them into a (tool-specific) “bin” subdirectory
# SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS.
#
# Configured for generic MPI with GCC compiler
#OPENMP = -fopenmp # GCC compiler
OPENMP = -openmp # Intel compiler

... # The Fortran compiler used for MPI programs
#MPIF77 = mpiifort # Intel compiler

# Alternative variant to perform instrumentation
MPIF77 = tau_f90.sh -tau_makefile=<path>/Makefile.tau-[options]

# PREP is a generic preposition macro for instrumentation preparation
#MPIF77 = $(PREP) mpif77 -f77=ifort
#MPIF77 = scorep ...

Default (no instrumentation)

Uncomment TAU’s compiler wrapper to do source instrumentation with TAU
Comment out Score-P wrapper
Building an NPB-MZ-MPI Benchmark

% make

============================================
= NAS PARALLEL BENCHMARKS 3.3 =
= MPI+OpenMP Multi-Zone Versions =
= F77 =
= ============================================

To make a NAS multi-zone benchmark type

make <benchmark-name> CLASS=<class> NPROCS=<nprocs>

where <benchmark-name> is “bt-mz”, “lu-mz”, or “sp-mz”
<class> is “S”, “W”, “A” through “F”
<nprocs> is number of processes

[...]

***************************************************************
* Custom build configuration is specified in config/make.def *
* Suggested tutorial exercise configuration for HPC systems: *
* make bt-mz CLASS=B NPROCS=30 *
***************************************************************

Type “make” for instructions
Building an NPB-MZ-MPI Benchmark

- Specify the benchmark configuration
  - benchmark name: `bt-mz`, `lu-mz`, `sp-mz`
  - the number of MPI processes: NPROCS=30
  - the benchmark class (S, W, A, B, C, D, E): CLASS=B

% make suite
make[1]: Entering directory `BT-MZ'
make[2]: Entering directory `sys'
cc -o setparams setparams.c -lm
make[2]: Leaving directory `sys'
../sys/setparams bt-mz 30 B
make[2]: Entering directory `../BT-MZ'
tau_f90.sh -c -O3 -g -openmp bt.f
 [...]
tau_f90.sh -c -O3 -g -openmp mpi_setup.f
cd ../common; mpiifort -c -O3 -g -openmp print_results.f
cd ../common; mpiifort -c -O3 -g -openmp timers.f
tau_f90.sh -O3 -g -openmp -o ../bin.tau/bt-mz_C.8 bt.o
initialize.o exact_solution.o exact_rhs.o set_constants.o adi.o
rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o solve_subs.o
z_solve.o add.o error.o verify.o mpi_setup.o ../common/print_results.o
../common/timers.o
make[2]: Leaving directory `BT-MZ'
Built executable ../bin.tau/bt-mz_B.30
make[1]: Leaving directory `BT-MZ'
Shortcut: % make suite
NPB-MZ-MPI / BT (Block Tridiagonal Solver)

- What does it do?
  - Solves a discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions
  - Performs 200 time-steps on a regular 3-dimensional grid
  - Implemented in 20 or so Fortran77 source modules

- Uses MPI & OpenMP in combination
  - 30 processes each with 4 threads should be reasonable
  - bt-mz_B.30 should take around 40 seconds
**TAU Source Instrumentation**

- Edit `config/make.def` to adjust build configuration
  - Uncomment specification of compiler/linker: `MPIF77 = tau_f90.sh`
- Make clean and build new tool-specific executable
  ```
  % make clean
  % make bt-mz CLASS=B NPROCS=30
  Built executable `../bin.tau/bt-mz_B.30`
  ```
- Change to the directory containing the new executable before running it with the desired tool configuration
  ```
  % cd `bin.tau`
  % cp ../jobscript/stampede/tau_mic.sbatch .
  % sbatch tau_mic.sbatch
  ```
NPB-MZ-MPI / BT with TAU

```
% cd bin
% cp ../jobscript/stampede/tau_mic.sbatch .
% sbatch tau_mic.sbatch
% cat mzmpibt.o<job_id>
NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
Number of zones:  8 x 8
Iterations:  200   dt:   0.000300
Number of active processes:     30
Total number of threads:       120 (  4.0 threads/process)

Time step   1
Time step   20
[...]
Time step 180
Time step 200
Verification Successful

BT-MZ Benchmark Completed.
Time in seconds = 39.03
% paraprof &
% paraprof --pack bt.ppk
<Copy file over to desktop using scp>
% paraprof bt.ppk &
```

- Copy jobscript and launch as a hybrid MPI+OpenMP application

Hint: save the benchmark output (or note the run time) to be able to refer to it later
$ tau_exec

Usage: tau_exec [options] [--] <exe> <exe options>

Options:
- v            Verbose mode
- s            Show what will be done but don’t actually do anything (dryrun)
- qsub         Use qsub mode (BG/P only, see below)
- io           Track I/O
- memory       Track memory allocation/deallocation
- memory_debug Enable memory debugger
- cuda         Track GPU events via CUDA
- cupti        Track GPU events via CUPTI (Also see env. variable TAU_CUPTI_API)
- opencl       Track GPU events via OpenCL
- openacc      Track GPU events via OpenACC (currently PGI only)
- ompt         Track OpenMP events via OMPT interface
- armci        Track ARMCI events via PARMCI
- ebs          Enable event-based sampling
- ebs_period=<count> Sampling period (default 1000)
- ebs_source=<counter> Counter (default itimer)
- um           Enable Unified Memory events via CUPTI
- T <DISABLE,GNU,ICPC,MPI,OMPT,OPENMP,PAPI,PDT,PROFILE,PTHREAD,SCOREP,SERIAL> : Specify TAU tags
- loadlib=<file.so> : Specify additional load library
- XrunTAUsh-<options> : Specify TAU library directly
- gdb          Run program in the gdb debugger

Notes:
  Defaults if unspecified: -T MPI
  MPI is assumed unless SERIAL is specified

- Tau_exec preloads the TAU wrapper libraries and performs measurements.

No need to recompile the application!
tau_exec Example (continued)

Example:
```
mpirun -np 2 tau_exec -T icpc,ompt,mpi -ompt ./a.out
mpirun -np 2 tau_exec -io ./a.out
```

Example - event-based sampling with samples taken every 1,000,000 FP instructions
```
mpirun -np 8 tau_exec -ebs -ebs_period=1000000 -ebs_source=PAPI_FP_INS ./ring
```

Examples - GPU:
```
tau_exec -T serial,cupti -cupti ./matmult (Preferred for CUDA 4.1 or later)
tau_exec -openacc ./a.out
tau_exec -T serial -opencl ./a.out (OPENCL)
mpirun -np 2 tau_exec -T mpi,cupti,papi -cupti -um ./a.out (Unified Virtual Memory in CUDA 6.0+)
```

qsub mode (IBM BG/Q only):
Original:
```
qsub -n 1 --mode smp -t 10 ./a.out
```
With TAU:
```
tau_exec -qsub -io -memory -- qsub -n 1 ... -t 10 ./a.out
```

Memory Debugging:
-memory option:
Tracks heap allocation/deallocation and memory leaks.
-memory_debug option:
Detects memory leaks, checks for invalid alignment, and checks for array overflow. This is exactly like setting TAU_TRACK_MEMORY_LEAKS=1 and TAU_MEMDBG_PROTECT_ABOVE=1 and running with -memory

- tau_exec can enable event based sampling while launching the executable using the -ebs flag!
- On stampede, you need to put perl-mic/bin in your path
- ibrun.symm -m test.sh
- Within test.sh call tau_exec -T ompt

TAU Analysis Tools: paraprof

- Launch paraprof

% paraprof

Metric
Paraprof main window

Colors represent code regions

Options -> uncheck Stack Bars Together
Paraprof main window

Unselect this to expand each routine in its own space
Paraprof main window

Each routine occupies its own space. Can see the extent of imbalance across all threads.
Paraprof node window (function barchart window)

Exclusive time spent in each code region (OpenMP loop) is shown here for MPI rank 0 thread 1.
Paraprof 3D visualization window

- Click Bar Plot
- Move Function and Thread Sliders
- Windows -> 3D visualization
Paraprof Thread Statistics Table with TAU_SAMPLING=1
Statement Level Profiling with TAU

Source location where samples are taken. Compute intensive region.
## Paraprof Thread Statistics Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Include Time</th>
<th>Include Time</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP_PARALLEL_REGION: L_z_solve_43_par_region0_2_44 [/scratch/saemir/NP83.3-MZ-MP/HT-MZ/L_z_solve.f] [43,0]</td>
<td>0.061</td>
<td>8.692</td>
<td>6,432</td>
<td>12,864</td>
</tr>
<tr>
<td>OpenMP_IMPLICTIC_TASK: L_z_solve_43_par_region0_2_44 [/scratch/saemir/NP83.3-MZ-MP/HT-MZ/L_z_solve.f] [43,0]</td>
<td>0.04</td>
<td>8.568</td>
<td>6,432</td>
<td>6,432</td>
</tr>
<tr>
<td>OpenMP_LOOP: L_z_solve_43_par_region0_2_44 [/scratch/saemir/NP83.3-MZ-MP/HT-MZ/L_z_solve.f] [43,0]</td>
<td>8.528</td>
<td>8.528</td>
<td>6,432</td>
<td>6,432</td>
</tr>
<tr>
<td>[CONTEXT] OpenMP_LOOP: L_z_solve_43_par_region0_2_44 [/scratch/saemir/NP83.3-MZ-MP/HT-MZ/L_z_solve.f] [43,0]</td>
<td>0</td>
<td>9.23</td>
<td>847</td>
<td>0</td>
</tr>
<tr>
<td>[SAMPLE] L_z_solve_43_par_region0_2_44 [/scratch/saemir/NP83.3-MZ-MP/HT-MZ/L_z_solve.f]</td>
<td>3.67</td>
<td>3.67</td>
<td>340</td>
<td>0</td>
</tr>
</tbody>
</table>

Right click here and choose “Show Source Code” for a sample
Frequently executing lightweight routines are automatically throttled at runtime. Reduces runtime dilation.
ParaProf Derived Metric Window: Intel MIC Vectorization Intensity

% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
% export TAU_METRICS=TIME,PAPI_NATIVE_VPU_ELEMENTS_ACTIVE,PAPI_NATIVE_VPU_INSTRUCTIONS_EXECUTED
% export TAU_PROFILE_FORMAT="merged"
% scp stampede:tauprofile.xml . ; paraprof tauprofile.xml [Options -> Show Derived Metric Panel]
ParaProf Comparison Window
ParaProf Manager Widow: scout.cubex

Metrics in the profile
ParaProf: Main Window
ParaProf: Thread Statistics Table

<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>omp do @y_solve.f:52</td>
<td>5.817</td>
<td>5.817</td>
<td>3,216</td>
<td>0</td>
</tr>
<tr>
<td>omp do @z_solve.f:52</td>
<td>5.657</td>
<td>5.657</td>
<td>3,216</td>
<td>0</td>
</tr>
<tr>
<td>omp do @x_solve.f:54</td>
<td>5.609</td>
<td>5.609</td>
<td>3,216</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:191</td>
<td>0.609</td>
<td>0.609</td>
<td>3,232</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:80</td>
<td>0.583</td>
<td>0.583</td>
<td>3,232</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>0.402</td>
<td>0.402</td>
<td>203</td>
<td>0</td>
</tr>
<tr>
<td>omp implicit barrier</td>
<td>0.402</td>
<td>0.402</td>
<td>203</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:301</td>
<td>0.36</td>
<td>0.36</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>omp implicit barrier</td>
<td>0.026</td>
<td>0.026</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp implicit barrier</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:37</td>
<td>0.343</td>
<td>0.343</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:62</td>
<td>0.225</td>
<td>0.225</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp implicit barrier</td>
<td>0.004</td>
<td>0.004</td>
<td>3,216</td>
<td>0</td>
</tr>
<tr>
<td>omp implicit barrier</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>MPI_init_thread</td>
<td>0.019</td>
<td>0.019</td>
<td>3,232</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:384</td>
<td>0.099</td>
<td>0.111</td>
<td>3,216</td>
<td>3,216</td>
</tr>
<tr>
<td>omp parallel do @add.f:22</td>
<td>0.069</td>
<td>0.069</td>
<td>3,232</td>
<td>0</td>
</tr>
<tr>
<td>omp do @rhs.f:428</td>
<td>0.043</td>
<td>0.043</td>
<td>603</td>
<td>0</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>0.04</td>
<td>0.04</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>omp do @initialize.f:50</td>
<td>0.03</td>
<td>0.03</td>
<td>51,712</td>
<td>0</td>
</tr>
<tr>
<td>omp parallel do @rhs.f:28</td>
<td>2.536</td>
<td>3.232</td>
<td>6,432</td>
<td>6,432</td>
</tr>
<tr>
<td>omp parallel do @exch_qbc.f:215</td>
<td>0.021</td>
<td>0.029</td>
<td>6,432</td>
<td>6,432</td>
</tr>
<tr>
<td>omp parallel do @exch_qbc.f:255</td>
<td>0.02</td>
<td>0.033</td>
<td>6,432</td>
<td>6,432</td>
</tr>
<tr>
<td>omp parallel do @exch_qbc.f:244</td>
<td>0.02</td>
<td>0.053</td>
<td>6,432</td>
<td>6,432</td>
</tr>
</tbody>
</table>
### ParaProf: Callpath Thread Relations Window

<table>
<thead>
<tr>
<th>Metric Name: Time</th>
<th>Sorted By: Exclusive</th>
<th>Units: seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>0.04</td>
<td>32/32</td>
</tr>
<tr>
<td>0.04</td>
<td>0.04</td>
<td>32</td>
</tr>
<tr>
<td>0.03</td>
<td>2.590</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.03</td>
<td>2.590</td>
<td>3232</td>
</tr>
<tr>
<td>9.0E-4</td>
<td>9.0E-4</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.225</td>
<td>0.220</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.002</td>
<td>0.002</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.002</td>
<td>0.002</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.199</td>
<td>0.199</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.002</td>
<td>0.002</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.243</td>
<td>0.243</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.016</td>
<td>0.016</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.016</td>
<td>0.016</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.016</td>
<td>0.016</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.016</td>
<td>0.016</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.009</td>
<td>0.009</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.006</td>
<td>0.006</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.006</td>
<td>0.006</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.015</td>
<td>0.015</td>
<td>3232/3232</td>
</tr>
<tr>
<td>0.211</td>
<td>0.200</td>
<td>6432/6432</td>
</tr>
<tr>
<td>0.021</td>
<td>0.020</td>
<td>6432/6432</td>
</tr>
<tr>
<td>0.007</td>
<td>0.007</td>
<td>6432/51600</td>
</tr>
<tr>
<td>0.008</td>
<td>0.008</td>
<td>6432/51600</td>
</tr>
<tr>
<td>0.006</td>
<td>0.006</td>
<td>6432/51600</td>
</tr>
<tr>
<td>0.007</td>
<td>0.007</td>
<td>6432/51600</td>
</tr>
<tr>
<td>0.02</td>
<td>0.033</td>
<td>6432/6432</td>
</tr>
<tr>
<td>0.02</td>
<td>0.033</td>
<td>6432/6432</td>
</tr>
<tr>
<td>0.013</td>
<td>0.012</td>
<td>6432/51600</td>
</tr>
</tbody>
</table>

TAU: ParaProf: Call Path Data n,c,t 0,0,0 - scout.cubex
ParaProf: 3D Visualization Window Showing Entire Profile
ParaProf: 3D Visualization Window MIC Vectorization Intensity
ParaProf: 3D Scatter Plot
ParaProf: Node View
ParaProf: Add Thread to Comparison Window
ParaProf: Score-P Profile Files, Database
ParaProf: File Preferences Window

[Image of ParaProf preferences window with options for setting font, size, and window defaults.]

- **Font**: Sans Serif, Bold, Italic
- **Size**: 0 to 40
- **Units**: Seconds
- **Settings**:
  - Show Path Title in Reverse
  - Reverse Call Paths
  - Interpret threads that do not call a given function as a 0 value for statistics computation
  - Generate data for reverse call tree
  - Show Source Locations
  - Auto label node/context/threads

[Buttons: Restore Defaults, Apply, Cancel]
ParaProf: Group Changer Window
ParaProf: Derived Metric Panel in Manager Window
Sorting Derived FLOPS metric by Exclusive Time
Download TAU from U. Oregon

http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD, OVA]

Free download, open source, BSD license
Automatic trace analysis with the Scalasca Trace Tools

Markus Geimer
Jülich Supercomputing Centre
Automatic trace analysis

- Idea
  - Automatic search for patterns of inefficient behavior
  - Classification of behavior & quantification of significance
  - Identification of delays as root causes of inefficiencies

- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability
Scalasca Trace Tools: Objective

- Development of a **scalable trace-based** performance analysis toolset for the most popular parallel programming paradigms
  - Current focus: MPI, OpenMP, and POSIX threads

- Specifically targeting large-scale parallel applications
  - Such as those running on IBM Blue Gene or Cray systems with one million or more processes/threads

- Latest release:
  - Scalasca v2.3.1 coordinated with Score-P v2.0.2 (May 2016)
Scalasca Trace Tools features

- Open source, 3-clause BSD license
- Fairly portable
  - IBM Blue Gene, Cray XT/XE/XK/XC, SGI Altix, Fujitsu FX10/100 & K computer, Linux clusters (x86, Power, ARM), Intel Xeon Phi, ...
- Uses Score-P instrumenter & measurement libraries
  - Scalasca v2 core package focuses on trace-based analyses
  - Supports common data formats
    - Reads event traces in OTF2 format
    - Writes analysis reports in CUBE4 format
- Current limitations:
  - Unable to handle traces
    - With MPI thread level exceeding MPI_THREAD_FUNNELED
    - Containing CUDA or SHMEM events, or OpenMP nested parallelism
  - PAPI/rusage metrics for trace events are ignored
Scalasca workflow

Optimized measurement configuration

Measurement library
- Measurement library
- Instr. target application
- HWC
- Instrumented executable
- Instrumenter compiler / linker
- Source modules

Local event traces
- Local event traces
- Parallel wait-state search
- Wait-state report

Scalasca trace analysis

Summary report
- Summary report
- Report manipulation

Which problem?
- Which problem?
- Where in the program?
- Which process?
Example: “Late Sender” wait state

- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication
Example: Critical path

- Shows call paths and processes/threads that are responsible for the program’s wall-clock runtime
- Identifies good optimization candidates and parallelization bottlenecks
Example: Root-cause analysis

- Classifies wait states into direct and indirect (i.e., caused by other wait states)
- Identifies *delays* (excess computation/communication) as root causes of wait states
- Attributes wait states as *delay costs*
Hands-on:
NPB-MZ-MPI / BT

scalasca
Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
  - 1.1 Summary measurement collection
  - 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
  - 2.1 Summary measurement collection with filtering
  - 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
  - 3.1 Event trace examination & analysis
**Scalasca command – One command for (almost) everything**

```bash
% scalasca
Scalasca 2.3.1
Toolset for scalable performance analysis of large-scale parallel applications
usage: scalasca [OPTION]... ACTION <argument>...
   1. prepare application objects and executable for measurement:
      scalasca -instrument <compile-or-link-command> # skin (using scorep)
   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

Options:
   -c, --show-config show configuration summary and exit
   -h, --help show this help and exit
   -n, --dry-run show actions without taking them
   --quickref show quick reference guide and exit
   --remap-specfile show path to remapper specification file and exit
   -v, --verbose enable verbose commentary
   -V, --version show version information and exit
```

- The `scalasca -instrument` command is deprecated and only provided for backwards compatibility with Scalasca 1.x., recommended: use Score-P instrumenter directly
Scalasca compatibility command: skin / scalasca -instrument

% skin
Scalasca 2.3.1: application instrumenter (using Score-P instrumenter)
   -comp={all|none|...}: routines to be instrumented by compiler [default: all]
     (... custom instrumentation specification depends on compiler)
   -pdt: process source files with PDT/TAU instrumenter
   -pomp: process source files for POMP directives
   -user: enable EPIK user instrumentation API macros in source code
   -v: enable verbose commentary when instrumenting
   --*: options to pass to Score-P instrumenter

- Scalasca application instrumenter
  - Provides compatibility with Scalasca 1.x
  - Deprecated! Use Score-P instrumenter directly.
Scalasca convenience command: `scan / scalasca -analyze`

```bash
% scan
Scalasca 2.3.1: measurement collection & analysis nexus
usage: scan {options} [launchcmd [launchargs]] target [targetargs]
    where {options} may include:
    -h    Help: show this brief usage message and exit.
    -v    Verbose: increase verbosity.
    -n    Preview: show command(s) to be launched but don't execute.
    -q    Quiescent: execution with neither summarization nor tracing.
    -s    Summary: enable runtime summarization. [Default]
    -t    Tracing: enable trace collection and analysis.
    -a    Analyze: skip measurement to (re-)analyze an existing trace.
    -e    exptdir: Experiment archive to generate and/or analyze.
             (overrides default experiment archive title)
    -f    filtfile: File specifying measurement filter.
    -l    lockfile: File that blocks start of measurement.
    -m    metrics: Metric specification for measurement.
```

- Scalasca measurement collection & analysis nexus
Scalasca advanced command:
scout - Scalasca automatic trace analyzer

% scout.hyb --help
SCOUT Copyright (c) 1998-2016 Forschungszentrum Juelich GmbH
Copyright (c) 2009-2014 German Research School for Simulation
Sciences GmbH

Usage: <launchcmd> scout.hyb [OPTION]... <ANCHORFILE | EPIK DIRECTORY>
Options:
    --statistics          Enables instance tracking and statistics [default]
    --no-statistics       Disables instance tracking and statistics
    --critical-path       Enables critical-path analysis [default]
    --no-critical-path    Disables critical-path analysis
    --rootcause           Enables root-cause analysis [default]
    --no-rootcause        Disables root-cause analysis
    --single-pass         Single-pass forward analysis only
    --time-correct        Enables enhanced timestamp correction
    --no-time-correct     Disables enhanced timestamp correction [default]
    --verbose, -v         Increase verbosity
    --help                Display this information and exit

- Provided in serial (.ser), OpenMP (.omp), MPI (.mpi) and MPI+OpenMP (.hyb) variants
**Scalasca advanced command: clc_synchronize**

- Scalasca trace event timestamp consistency correction

  Usage: `<launchcmd> clc_synchronize.hyb <ANCHORFILE | EPIK_DIRECTORY>`

- Provided in MPI (.mpi) and MPI+OpenMP (.hyb) variants
- Takes as input a trace experiment archive where the events may have timestamp inconsistencies
  - E.g., multi-node measurements on systems without adequately synchronized clocks on each compute node
- Generates a new experiment archive (always called ./clc_sync) containing a trace with event timestamp inconsistencies resolved
  - E.g., suitable for detailed examination with a time-line visualizer
Scalasca convenience command: square / scalasca -examine

```
% square
Scalasca 2.3.1: analysis report explorer
    -c <none | quick | full> : Level of sanity checks for newly created reports
    -F                       : Force remapping of already existing reports
    -f filtfile              : Use specified filter file when doing scoring
    -s                       : Skip display and output textual score report
    -v                       : Enable verbose mode
    -n                       : Do not include idle thread metric
```

- Scalasca analysis report explorer (Cube)
Automatic measurement configuration

- scan configures Score-P measurement by automatically setting some environment variables and exporting them
  - E.g., experiment title, profiling/tracing mode, filter file, ...
  - Precedence order:
    - Command-line arguments
    - Environment variables already set
    - Automatically determined values
- Also, scan includes consistency checks and prevents corrupting existing experiment directories
- For tracing experiments, after trace collection completes then automatic parallel trace analysis is initiated
  - Uses identical launch configuration to that used for measurement (i.e., the same allocated compute resources)
Setup environment

- Remember to source provided shell code snippet to add local tool installations to $PATH

  ```bash
  % source ~tg828282/Tutorial/vihps.sh
  ```

- Change to directory containing NPB3.3-MZ-MPI sources
- Existing instrumented executable in bin.scorep/ directory can be reused

  ```bash
  % cd $SCRATCH/NPB3.3-MZ-MPI
  ```
BT-MZ summary measurement collection...

% cd bin.scorep
% cp ..:/jobscript/stampede/scalasca_mic.sbatch .
% vi scalasca_mic.sbatch

[...]

export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=32M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="—time-correct"

scalasca -analyze ibrun.symm -m ./bt-mz_${CLASS}.${PROCS}

% sbatch scalasca_mic.sbatch

- Change to directory with the executable and edit the job script
- Submit the job
BT-MZ summary measurement

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command

- Creates experiment directory: 
  ./scorep_bt-mz_B_mic15p30x4_sum

S=C=A=N: Scalasca 2.3.1 runtime summarization
S=C=A=N: ./scorep_bt-mz_B_mic15p30x4_sum experiment archive
S=C=A=N: Fri Jun 10 08:55:37 2016: Collect start
ibrun.symm -m ./bt-mz_B.30

NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark

Number of zones: 8 x 8
Iterations: 200 dt: 0.000300
Number of active processes: 30

[... More application output ...]

S=C=A=N: Fri Jun 10 08:56:17 2016: Collect done (status=0) 40s
S=C=A=N: ./scorep_bt-mz_B_mic15p30x4_sum complete.
BT-MZ summary analysis report examination

- Score summary analysis report

```bash
% square -s scorep_bt-mz_B_mic15p30x4_sum
INFO: Post-processing runtime summarization result...
INFO: Score report written to ./scorep_bt-mz_B_mic15p30x4_sum/scorep.score
```

- Post-processing and interactive exploration with Cube

```bash
% square scorep_bt-mz_B_mic15p30x4_sum
INFO: Displaying ./scorep_bt-mz_B_mic15p30x4_sum/summary.cubex...
```

[GUI showing summary analysis report]

- The post-processing derives additional metrics and generates a structured metric hierarchy

Hint: Copy ‘summary.cubex’ to Live-DVD environment using ‘scp’ to improve responsiveness of GUI
### Post-processed summary analysis report

#### Split base metrics into more specific metrics

<table>
<thead>
<tr>
<th>Metric Tree</th>
<th>Call Tree</th>
<th>System Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td>0.02 exact_rhs_</td>
<td>- generic cluster</td>
</tr>
<tr>
<td>0.00 Execution</td>
<td>0.42 exch_qbc_</td>
<td>- i06r01c20</td>
</tr>
<tr>
<td>0.00 MPI</td>
<td>0.00 adi_</td>
<td>- 0.05 CPU thread 0</td>
</tr>
<tr>
<td>0.01 Synchronization</td>
<td>4.99 compute_rhs_</td>
<td>- 1.43 CPU thread 1</td>
</tr>
<tr>
<td>11.28 Communication</td>
<td>0.00 x_solve_</td>
<td>- 0.02 CPU thread 2</td>
</tr>
<tr>
<td>0.00 File I/O</td>
<td>0.00 !$omp parallel @x_solve_</td>
<td>- 0.01 CPU thread 3</td>
</tr>
<tr>
<td>0.84 Init/Exit</td>
<td>0.00 !$omp do @x_solve_</td>
<td></td>
</tr>
<tr>
<td>0.00 OMP</td>
<td>9.09 !$omp implicit barrier</td>
<td></td>
</tr>
<tr>
<td>0.00 Flush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.48 Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.6B Synchronization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38.64 idle threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.93e6 Visits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Synchronizations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Communications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Bytes transferred</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Sent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.27e8 Point-to-point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>608 Collective</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Received</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.27e8 Point-to-point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>608 Collective</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **File Display**
  - Topology
  - Help
- **Topology**
  - File Display
  - Topology
  - Help
Performance analysis steps

- 0.0 Reference preparation for validation
- 1.0 Program instrumentation
  - 1.1 Summary measurement collection
  - 1.2 Summary analysis report examination
- 2.0 Summary experiment scoring
  - 2.1 Summary measurement collection with filtering
  - 2.2 Filtered summary analysis report examination
- 3.0 Event trace collection
- 3.1 Event trace examination & analysis
BT-MZ trace measurement collection...

% cd bin.scorep
% cp ../jobscript/stampede/scalasca_mic.sbatch .
% vi scalasca_mic.sbatch

[...]

export SCOREP_FILTERING_FILE=../config/scorep.filt
export SCOREP_TOTAL_MEMORY=32M
export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="--time-correct"

scalasca -analyze -t ibrun.symm -m ./bt-mz_${CLASS}.${PROCS}

% sbatch scalasca_mic.sbatch

- Change to directory with the executable and edit the job script
- Add "-t" to the scalasca -analyze command
- Submit the job
BT-MZ trace measurement ... collection

- Starts measurement with collection of trace files ...

S=C=A=N: Scalasca 2.3.1 trace collection and analysis
S=C=A=N: Fri Jun 10 09:31:03 2016: Collect start
ibrun.symm -m ./bt-mz_B.30

NAS Parallel Benchmarks (NPB3.3-MZ-MPI) – BT-MZ MPI+OpenMP \>Benchmark

Number of zones: 8 x 8
Iterations: 200 dt: 0.000300
Number of active processes: 30

[... More application output ...]

S=C=A=N: Fri Jun 10 09:31:49 2016: Collect done (status=0) 46s
BT-MZ trace measurement ... analysis

- Continues with automatic (parallel) analysis of trace files

S=C=A=N: Fri Jun 10 09:31:03 2016: Analyze start
ibrun.symm -m scout.hyb ./scorep_bt-mz_B_mic15p30x4_trace/traces.otf2

Analyzing experiment archive
./scorep_bt-mz_B_mic15p30x4_trace/traces.otf2

Opening experiment archive ... done (0.099s).
Reading definition data ... done (0.069s).
Reading event trace data ... done (1.008s).
Preprocessing ... done (0.253s).
Timestamp correction ... done (0.880s).
Analyzing trace data ... done (9.906s).
Writing analysis report ... done (2.705s).

Total processing time: 14.681s
S=C=A=N: Fri Jun 10 09:32:06 2016: Analyze done (status=0) 17s
BT-MZ trace analysis report exploration

- Produces trace analysis report in the experiment directory containing trace-based wait-state metrics

```bash
% square scorep_bt-mz_B_mic15p30x4_trace
INFO: Post-processing runtime summarization result...
INFO: Post-processing trace analysis report...
INFO: Displaying ./scorep_bt-mz_B_mic15p30x4_trace/trace.cubex...
```

[GUI showing trace analysis report]

**Hint:**

Run ‘square -s’ first and then copy ‘trace.cubex’ to Live-DVD environment using ‘scp’ to improve responsiveness of GUI
Post-processed trace analysis report

Additional trace-based metrics in metric hierarchy
Online metric description

Access online metric description via context menu
Online metric description

**Late Sender Time**

**Description:**
Refers to the time lost waiting caused by a blocking receive operation (e.g., `MPI_Recv` or `MPI_Wait`) that is posted earlier than the corresponding send operation.

If the receiving process is waiting for multiple messages to arrive (e.g., in an call to `MPI_Waitall`), the maximum waiting time is accounted, i.e., the waiting time due to the latest sender.

**Unit:**
Seconds

**Diagnosis:**
- Try to replace `MPI_Recv` with a non-blocking receive `MPI_1recv` that can be posted earlier, proceed concurrently with computation, and complete with a wait operation after the message is expected to have been sent. Try to post sends earlier, such that they are available when receivers need them. Note that outstanding messages (i.e., sent before the receiver is ready) will occupy internal message buffers, and that large numbers of posted receive buffers will also introduce message management overhead. Therefore moderation is advisable.

**Parent:**
`MPI Point-to-point Communication Time`

**Children:**
Critical-path analysis

Critical-path profile shows wall-clock time impact
Critical-path analysis

Critical-path imbalance highlights inefficient parallelism
Pattern instance statistics

Access pattern instance statistics via context menu

Click to get statistics details
To investigate most severe pattern instances, connect to a trace browser...

...and select trace file from the experiment directory.
Show most severe pattern instances

Select “Max severity in trace browser” from context menu of call paths marked with a red frame.
Investigate most severe instance in Vampir

Vampir will automatically zoom to the worst instance in multiple steps (i.e., undo zoom provides more context)
Scalasca Trace Tools: Further information

- Collection of trace-based performance tools
  - Specifically designed for large-scale systems
  - Features an automatic trace analyzer providing wait-state, critical-path, and delay analysis
  - Supports MPI, OpenMP, POSIX threads, and hybrid MPI+OpenMP/Pthreads
- Available under 3-clause BSD open-source license
- Documentation & sources:
  - http://www.scalasca.org
- Contact:
  - mailto: scalasca@fz-juelich.de
Performance Analysis with Vampir

Bert Wesarg
Technische Universität Dresden
Outline

- **Part I: Welcome to the Vampir Tool Suite**
  - Mission
  - Event Trace Visualization
  - Vampir & VampirServer
  - The Vampir Displays

- **Part II: Vampir Hands-On**
  - Visualizing and analyzing NPB-MZ-MPI / BT
Event Trace Visualization with Vampir

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics

**Timeline charts**
- Show application activities and communication along a time axis

**Summary charts**
- Provide quantitative results for the currently selected time interval
Visualization Modes (1)
Directly on front end or local machine

% vampir

Multi-Core Program

Score-P

Trace File

Vampir

Small/Medium sized trace

Thread parallel
Visualization Modes (2)
On local machine with remote VampirServer

% vampirserver start
% vampir

LAN/WAN
Large Trace File (stays on remote machine)
Parallel application
The main displays of Vampir

- Timeline Charts:
  - Master Timeline
  - Process Timeline
  - Counter Data Timeline
  - Performance Radar

- Summary Charts:
  - Function Summary
  - Message Summary
  - Process Summary
  - Communication Matrix View
Hands-on: Visualizing and analyzing NPB-MZ-MPI / BT
Help! Where is my trace file?

% ls $SCRATCH/NPB3.3-MZ-MPI/bin.scorep/
> scorep_bt-mz_B_mic15p30x4_trace
profile.cubex scorep.cfg traces/ traces.def traces.otf2

% ls ~tg828282/Tutorial/traces/scorep_bt-mz_B_mic15p30x4_trace
profile.cubex scorep.cfg traces/ traces.def traces.otf2

- If you followed the Score-P hands-on up to the trace experiment
- If you did not follow to that point, take a prepared trace
Starting VampirServer on Stampede

% vampirserver start
Launching VampirServer...
Submitting batch job (this might take a while)...

- Start VampirServer on Stampede
Starting VampirServer on Stampede

% vampirserver start
Launching VampirServer...
Submitting batch job (this might take a while)...

VampirServer 9.0.0 (r9950)
Licensed to ZIH, TU Dresden (@ISC`16)
Running 4 analysis processes... (abort with 
  vampirserver stop 28974)
VampirServer <28974> listens on: 
  c401-602.stampede.tacc.utexas.edu:30019

- Start VampirServer on Stampede
  Copy host:port
Open a port forwarding to Stampede to be able to access the VampirServer.
Start Vampir on local computer
Use the “Open Other” option
Select “Remote File”
Server is "localhost"

Port is "30000"

Connection type "Socket"
Visualization of the NPB-MZ-MPI / BT trace

- Navigation Toolbar
- Function Summary
- Master Timeline
- Function Legend
Visualization of the NPB-MZ-MPI / BT trace
Master Timeline

Detailed information about functions, communication and synchronization events for collection of processes.
Visualization of the NPB-MZ-MPI / BT trace

Process Timeline

Detailed information about different levels of function calls in a stacked bar chart for an individual process.
Visualization of the NPB-MZ-MPI / BT trace

Typical program phases

- Initialisation Phase
- Computation Phase
Visualization of the NPB-MZ-MPI / BT trace
Counter Data Timeline

Detailed counter information over time for an individual process.
Visualization of the NPB-MZ-MPI / BT trace
Performance Radar

Detailed counter information over time for a collection of processes.
Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Initialisation Phase

Context View: Detailed information about function “initialize_”.
Visualization of the NPB-MZ-MPI / BT trace

Find Function

Execution of function “initialize_” results in higher page fault rates.
Visualization of the NPB-MZ-MPI / BT trace

Computation Phase

Computation phase results in higher floating point operations.
MPI communication results in lower floating point operations.
Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Finalisation Phase

“Early reduce” bottleneck.
Visualization of the NPB-MZ-MPI / BT trace

Process Summary:
Overview of the accumulated information across all functions and for every process independently.

Function Summary:
Overview of the accumulated information across all functions and for a collection of processes.
Find groups of similar processes and threads by using summarized function information.
Summary and Conclusion
Summary

- **Vampir & VampirServer**
  - Interactive trace visualization and analysis
  - Intuitive browsing and zooming
  - Scalable to large trace data sizes (20 TiByte)
  - Scalable to high parallelism (200,000 processes)

- Vampir for Linux, Windows, and Mac OS X
Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team
Score-P:
Specialized Measurements and Analyses
Mastering build systems

- Hooking up the Score-P instrumenter `scorep` into complex build environments like `Autotools` or `CMake` was always challenging.
- Score-P provides new convenience wrapper scripts to simplify this (since Score-P 2.0).
- `Autotools` and `CMake` need the used compiler already in the `configure step`, but instrumentation should not happen in this step, only in the `build step`.

```bash
% SCOREP_WRAPPER=off \
> cmake .. \
> -DCMAKE_C_COMPILER=scorep-icc \
> -DCMAKE_CXX_COMPILER=scorep-icpc
```

- Allows to pass addition options to the Score-P instrumenter and the compiler via environment variables without modifying the `Makefiles`.
- Run `scorep-wraper --help` for a detailed description and the available wrapper scripts of the Score-P installation.
Mastering C++ applications

- Automatic compiler instrumentation greatly disturbs C++ applications because of frequent/short function calls => Use sampling instead
- Novel combination of sampling events and instrumentation of MPI, OpenMP, ...
  - Sampling replaces compiler instrumentation (instrument with --nocompiler to further reduce overhead) => Filtering not needed anymore
  - Instrumentation is used to get accurate times for parallel activities to still be able to identify patterns of inefficiencies
- Supports profile and trace generation

```bash
% export SCOREP_ENABLE_UNWINDING=true
% # use the default sampling frequency
% #export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000
% OMP_NUM_THREADS=4 mpiexec –np 4 ./bt-mz_W.4
```

- Set new configuration variable to enable sampling

- Available since Score-P 2.0, only x86-64 supported currently
Mastering C++ applications

Less disturbed measurement
Mastering application memory usage

- Determine the maximum heap usage per process
- Find high frequent small allocation patterns
- Find memory leaks
- Support for:
  - C, C++, MPI, and SHMEM (Fortran only for GNU Compilers)
  - Profile and trace generation (profile recommended)
    - Memory leaks are recorded only in the profile
    - Resulting traces are not supported by Scalasca yet

% export SCOREP_MEMORY_RECORDING=true
% export SCOREP_MPI_MEMORY_RECORDING=true

% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4

- Set new configuration variable to enable memory recording

- Available since Score-P 2.0
Mastering application memory usage

Different maximum heap usages per ranks
Mastering application memory usage

Memory leaks
Mastering heterogeneous applications

- Record CUDA applications and device activities
  % export SCOREP_CUDA_ENABLE=gpu,kernel,idle

- Record OpenCL applications and device activities
  % export SCOREP_OPENCL_ENABLE=api,kernel

- Record OpenACC applications
  % export SCOREP_OPENACC_ENABLE=yes

- Can be combined with CUDA if it is a NVIDIA device
  % export SCOREP_CUDA_ENABLE=kernel
Mastering heterogeneous applications
Enriching measurements with performance counters

- Available PAPI metrics
  - Preset events: common set of events deemed relevant and useful for application performance tuning
    - Abstraction from specific hardware performance counters, mapping onto available events done by PAPI internally
  - Native events: set of all events that are available on the CPU (platform dependent)

Note: Due to hardware restrictions:
- Number of concurrently recorded events is limited
- There may be invalid combinations of concurrently recorded events

- Use the metric names in one of the SCOREP_METRIC_PAPI or SCOREP_METRIC_PAPI_PER_PROCESS measurement configuration variables separated by comma
Enriching measurements with performance counters

% man getrusage
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_miflmt; /* page reclaims (soft page faults) */
    long ru_majflt; /* page faults (hard page faults) */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* IPC messages sent */
    long ru_msgrcv; /* IPC messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};

- Available resource usage metrics
- Note:
  1. Not all fields are maintained on each platform.
  2. Check scope of metrics (per process vs. per thread)
- Use the member names in one of these measurement configuration variables separated by comma:
  - SCOREP_METRIC_RUSAGE
  - SCOREP_METRIC_RUSAGE_PER_PROCESS
Score-P user instrumentation API

- No replacement for automatic compiler instrumentation

- Can be used to further subdivide functions
  - E.g., multiple loops inside a function

- Can be used to partition application into coarse grain phases
  - E.g., initialization, solver, & finalization

- Enabled with `--user` flag to Score-P instrumenter

- Available for Fortran / C / C++
Score-P user instrumentation API (Fortran)

```fortran
#include "scorep/SCOREP_User.inc"

subroutine foo(...)  
! Declarations
  SCOREP_USER_REGION_DEFINE( solve )

! Some code...
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \  
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100  
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
! Some more code...
end subroutine
```

- Requires processing by the C preprocessor
- For most compilers, this can be automatically achieved by having an uppercase file extension, e.g., main.F or main.F90
Score-P user instrumentation API (C/C++)

```c
#include "scorep/SCOREP_User.h"

void foo()
{
    /* Declarations */
    SCOREP_USER_REGION_DEFINE( solve )

    /* Some code... */
    SCOREP_USER_REGION_BEGIN( solve, "<solver>",
                                SCOREP_USER_REGION_TYPE_LOOP )
    for (i = 0; i < 100; i++)
    {
        [...]
    }
    SCOREP_USER_REGION_END( solve )
    /* Some more code... */
}
```
Score-P user instrumentation API (C++)

```cpp
#include "scorep/SCOREP_User.h"

void foo()
{
    // Declarations

    // Some code...
    {
        SCOREP_USER_REGION( "<solver>",
            SCOREP_USER_REGION_TYPE_LOOP
        )
        for (i = 0; i < 100; i++)
        {
            [...]
        }
    }
    // Some more code...
}
```
Score-P measurement control API

- Can be used to temporarily disable measurement for certain intervals
  - Annotation macros ignored by default
  - Enabled with `--user` flag

```fortran
#include "scorep/SCOREP_User.inc"

subroutine foo(...)  
  ! Some code...  
  SCOREP_RECORDING_OFF()  
  ! Loop will not be measured  
  do i=1,100  
    [...]  
  end do  
  SCOREP_RECORDING_ON()  
  ! Some more code...  
end subroutine
```

```c
#include "scorep/SCOREP_User.h"

void foo(...) {  
  /* Some code... */  
  SCOREP_RECORDING_OFF()  
  /* Loop will not be measured */  
  for (i = 0; i < 100; i++) {  
    [...]  
  }  
  SCOREP_RECORDING_ON()  
  /* Some more code... */  
}
```

Fortran (requires C preprocessor)  C / C++
Score-P: Conclusion and Outlook
Project management

- Ensure a single official release version at all times which will always work with the tools

- Allow experimental versions for new features or research

- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
Future features

- Scalability to maximum available CPU core count
- Support for emerging architectures and new programming models

Features currently worked on:
- User provided wrappers to 3rd party libraries
- Hardware and MPI topologies
- Basic support of measurements without re-compiling/-linking
- I/O recording
- Java recording
- Persistent memory recording (e.g., PMEM, NVRAM, ...)

VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING
Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods) and sampling
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under New BSD open-source license
- Documentation & Sources:
  - http://www.score-p.org
- User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date
Performance data management with TAU
PerfExplorer

Sameer Shende
sameer@cs.uoregon.edu
University of Oregon
http://tau.uoregon.edu
TAU Analysis

Instrumentation

Profiles

Profiles

Profiles

Traces

Symbol table

Profile Data Management (PerfDMF)

Profile Analysis (ParaProf)

Profile Data Mining (PerfExplorer)

Trace Data Management

Trace Visualizers

Trace Analyzers

Profile Translators

Metadata (XML)

Profile Database

Trace Translators

Trace Storage

Vampir

Expert

JumpShot

ProfileGen

Paraver

Vampir Server
TAUdb: Performance Data Management Framework

TAU Performance System

Performance Analysis Process

Query and Analysis Toolkit

Data Mining (Weka)

Statistics (R / Omega)

Java PerfDMF API

SQL (PostgreSQL, MySQL, DB2, Oracle)

Profile metadata

Raw profiles

* gprof
* mpiP
* psrun
* HPMtoolkit
* ...

XML document

Formatted profile data
Using TAUdb

- **Configure TAUdb (Done by each user)**
  % taudb_configure --create-default
  - Choose derby, PostgreSQL, MySQL, Oracle or DB2
  - Hostname
  - Username
  - Password
  - Say yes to downloading required drivers (we are not allowed to distribute these)
  - Stores parameters in your ~/.ParaProf/taudb.cfg file

- **Configure PerfExplorer (Done by each user)**
  % perfexplorer_configure

- **Execute PerfExplorer**
  % perfexplorer
% wget http://tau.uoregon.edu/data.tgz (Contains CUBE profiles from Score-P)
% tar zxf data.tgz; cd data; cat README; cd tau; ./upload.sh; perfexplorer
Or manually:
% taudb_configure --create-default
(Chooses derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK) OR use taudb_loadtrial -
 a “app” -x “experiment” -n “name” file.ppk
Then,
% tar zxf $TAU/data.tgz; cd data/tau;
% taudb_loadtrial -a BT_MZ -x “Class_B” bt-mz_B.*.ppk
% perfexplorer
(Select experiment, Menu: Charts -> Speedup)
Performance Data Mining (PerfExplorer)

- Performance knowledge discovery framework
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, …
  - Use the existing TAU infrastructure
    - TAU performance profiles, taudb
  - Client-server based system architecture

- Technology integration
  - Java API and toolkit for portability
  - taudb
  - R-project/Omegahat, Octave/Matlab statistical analysis
  - WEKA data mining package
  - JFreeChart for visualization, vector output (EPS, SVG)
PerfExplorer: Using Cluster Analysis

- Performance data represented as vectors - each dimension is the cumulative time for an event
- \( k \)-means: \( k \) random centers are selected and instances are grouped with the "closest" (Euclidean) center
- New centers are calculated and the process repeated until stabilization or max iterations
- Dimension reduction necessary for meaningful results
- Virtual topology, summaries constructed
PerfExplorer - Cluster Analysis (sPPM)
PerfExplorer - Correlation Analysis (Flash)

- Describes strength and direction of a linear relationship between two variables (events) in the data
PerfExplorer - Correlation Analysis (Flash)

-0.995 indicates strong, negative relationship.

As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases.
PerfExplorer - Comparative Analysis

- Relative speedup, efficiency
  - total runtime, by event, one event, by phase
- Breakdown of total runtime
- Group fraction of total runtime
- Correlating events to total runtime
- Timesteps per second
PerfExplorer - Interface

Select experiments and trials of interest

Data organized in application, experiment, trial structure (will allow arbitrary in future)

Experiment metadata
PerfExplorer - Interface
PerfExplorer - Relative Efficiency Plots

![Relative Efficiency Plot](image-url)
PerfExplorer - Relative Efficiency by Routine
PerfExplorer - Relative Speedup

![Graph showing Relative Speedup - GYRO:Time](image_url)
PerfExplorer - Timesteps Per Second
Evaluate Scalability

- **Goal**: How does my application scale? What bottlenecks occur at what core counts?
- **Load profiles in taudb database and examine with PerfExplorer**
Evaluate Scalability
PerfExplorer

![Total TIME Bar Chart for IRMHD:Scaling_BGP](image)

- ADVANCE_DIFFUSION
- CCHEBYB
- CCHEBYF
- CHEBDIFF
- DERIVE
- ENERGY
- MPIFFT::CRFFT2D_MPI
- MPIFFT::HC2R
- MPIFFT::R2HC
- MPIFFT::RCFFT2D_MPI
- MPIFFT::REORDER_COMPLEX
- MPI_Allreduce0
- MPI_Alltoall0
- MPI_BARRIER0
- MPI_Init0
- MPI_Waitall0
- RK3NL::ERK3_STAGE
- WDERIVS
- other
PerfExplorer
Performance Regression Testing

FACETS Bassi Regression: 32 Procs (events above 2%)

Date

Exclusive Time (seconds)

0 10 20 30 40 50 60 70 80


- int main(int, char **)
- std::vector<double, std::allocator<double>> > FcCoreCellUpdate...
- void FcTmCoreFluxCalc::computeFluxes() 
- MPI_RecvQ
- double FcDataAssimilator::getValue(const std::string &, cons... MPI_Init() 
- FcHdfSTmpl<DATATYPE>::writeDataSet
- void FcDataAssimilatorUfiles::parseUfiles(const std::vector<...
- void FcUpdaterComponent::dumpToFile(const std::string &) con... other
Download TAU from U. Oregon

http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD, OVA]

Free download, open source, BSD license
Typical performance bottlenecks and how they can be identified

The VI-HPS Team
Outline

• Case I:
  • Load imbalances in OpenMP codes

• Case II:
  • Communication and computation overlapping in MPI codes

• Note: We won’t do the complete performance engineering cycle here.
Case I: Sparse matrix vector multiplication

\[
\begin{pmatrix}
  y_1 \\
  \vdots \\
  y_m
\end{pmatrix} =
\begin{pmatrix}
  a_{11} & \cdots & a_{n1} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \cdots & a_{mn}
\end{pmatrix} \cdot
\begin{pmatrix}
  x_1 \\
  \vdots \\
  x_n
\end{pmatrix}
\]

- A sparse matrix is a matrix populated primarily with zeros
- Only non-zero elements of \( a_{ij} \) are saved efficiently in memory
- Algorithm

```c
foreach row r in A
    y[r.x] = 0
    foreach non-zero element e in row
        y[r.x] += e.value * x[e.y]
```
Case I: Sparse matrix vector multiplication

- Naive OpenMP algorithm

```c
#pragma omp parallel for
define row r in A
    y[r.x] = 0
    foreach non-zero element e in row
        y[r.x] += e.value * x[e.y]
```

- Distributes the rows of A evenly across the threads in the parallel region
- The distribution of the non-zero elements may influence the load balance in the parallel application
Case I: Load imbalances in OpenMP codes

- Measuring the static OpenMP application

```bash
% cd ~/Bottlenecks/smxv
% make PREP=scorep
scorep gcc -fopenmp -DLITTLE_ENDIAN \
  -DFUNCTION_INC='"y_Ax-omp.inc.c"' -DFUNCTION=y_Ax_omp \
  -o smxv-omp smxv.c -lm
scorep gcc -fopenmp -DLITTLE_ENDIAN \
  -DFUNCTION_INC='"y_Ax-omp-dynamic.inc.c"' \
  -DFUNCTION=y_Ax_omp_dynamic -o smxv-omp-dynamic smxv.c -lm
% OMP_NUM_THREADS=8 scan -t ./smxv-omp yax_large.bin
```
Case I: Load imbalances in OpenMP codes: Profile

- Two metrics which indicate load imbalances:
  - Time spent in OpenMP barriers
  - Computational imbalance

- Open prepared measurement on the LiveDVD with Cube

```
% cube ~/Bottlenecks/smxv/scorep_smxv-omp_large/trace.cubex
```

[CUBE GUI showing trace analysis report]
Case I: Time spent in OpenMP barriers

Great variation in the distribution of the time spent in the barrier.
Case I: Computational imbalance

Master thread does 24% of the work (2.00/8.30) and has 66% of the computational overload.
Case I: Sparse matrix vector multiplication

- Improved OpenMP algorithm

```c
#pragma omp parallel for schedule(dynamic,1000)
foreach row r in A
    y[r.x] = 0
    foreach non-zero element e in row
        y[r.x] += e.value * x[e.y]
```

- Distributes the rows of A dynamically across the threads in the parallel region
Case I: Profile Analysis

- Two metrics which indicate load imbalances
  - Time spent in OpenMP barriers
  - Computational imbalance

- Open prepared measurement on the LiveDVD with Cube

```
cube ~/Bottlenecks/smxv/scorep_smxv-omp-dynamic_large/trace.cubex
```

[CUBE GUI showing trace analysis report]
Case I: Time spent in OpenMP barriers

Variation is now much smaller
Case I: Computational imbalance

Computational imbalance can still be improved
Case I: Trace comparison

- Open prepared measurement on the LiveDVD with Vampir

```
% vampir ~/Bottlenecks/smxv/scorep_smxv-omp_large/traces.otf2 \
~/Bottlenecks/smxv/scorep_smxv-omp-dynamic_large/traces.otf2
```

[Vampir GUI showing trace]
Case I: Time spent in OpenMP barriers

- Improved runtime
- Less time in OpenMP barrier
Case I: Computational imbalance

Great imbalance for time spent in computational code.
Outline

- Case I:
  - Load imbalances in OpenMP codes

- Case II:
  - Communication and computation overlapping in MPI codes
Case II: Heat conduction simulation

- Calculating the heat conduction at each time step
- Discretized formula for space $dx, dy$ and time $dt$

\[
\theta_{i,j}^{t+1} = \theta_{i,j}^t + \left( \frac{\theta_{i+1,j}^t - 2\theta_{i,j}^t + \theta_{i-1,j}^t}{dx^2} + \frac{\theta_{i,j+1}^t - 2\theta_{i,j}^t + \theta_{i,j-1}^t}{dy^2} \right) \cdot k \cdot dt
\]
Case II: Heat conduction simulation

- Application uses MPI for boundary exchange
- Simulation grid is distributed across MPI ranks
Case II: Heat conduction simulation

- Ranks need to exchange boundaries before next iteration step
**Case II: Profile analysis**

- **MPI algorithm**

  ```
  foreach step in [1:nsteps]
  exchangeBoundaries
  computeHeatConduction
  ```

- **Building and measuring the heat conduction application**

  ```
  % cd ~/Bottlenecks/heat
  % make PREP='scorep --user'
  [...] make output [...] 
  % scan mpirun -np 16 ./heat-MPI 3072 32
  ```

- **Open prepared measurement on the LiveDVD with Cube**

  ```
  % cube ~/Bottlenecks/heat/scorep_heat-MPI_small/profile.cubex
  ```
  
  ![CUBE GUI showing trace analysis report]
Case II: Time spent in boundary exchange

Each process spent 8 seconds in boundary exchange.
Case II: Time spent in boundary exchange

... that’s ~10% of the computation
Case II: Hide MPI communication with computation

- Step 1: Compute heat in the area which is communicated to your neighbors

Compute heat conduction in the boundaries of $P_i$
Case II: Hide MPI communication with computation

- Step 2: Start communicating boundaries with your neighbors
Case II: Hide MPI communication with computation

- Step 3: Compute heat in the interior area

Compute heat conduction in the interior of $P_i$
Case II: Profile analysis

- Improved MPI algorithm

```c
foreach step in [1:nsteps]
    computeHeatConductionInBoundaries
    startBoundaryExchange
    computeHeatConductionInInterior
    waitForCompletionOfBoundaryExchange
```

- Note: As not all MPI implementations support overlapping, it is here done with the help of OpenMP tasks.

- Measuring the improved heat conduction application

  ```
  % scan mpirun -np 16 ./heat-MPI-overlap 3072 32
  ```

- Open prepared measurement on the LiveDVD with Cube

  ```
  % cube ~/Bottlenecks/heat/scorep_heat-MPI-overlap_small/profile.cubex
  ```

  [CUBE GUI showing trace analysis report]
Case II: Time spent in boundary exchange

Still ~8 seconds in boundary exchange
Case II: Time spent in boundary exchange

... still ~10% of the computation
Case II: Profile comparison

- Calculate differences between profiles

```bash
% cube_diff ~/Bottlenecks/heat/scorep_heat-MPI_small/profile.cubex \\ ~/Bottlenecks/heat/scorep_heat-MPI-overlap_small/profile.cubex
```

- Open prepared profile diff on the LiveDVD with Cube

```bash
% cube ~/Bottlenecks/heat/diff.cubex
```

[CUBE GUI showing trace analysis report]
Case II: Profile comparison

But all threads spend ~8 seconds less in the main loop.
Case II: Trace comparison

- Open prepared measurement on the LiveDVD with Vampir

```
% vampir ~/Bottlenecks/heat/scorep_heat-MPI_small/traces.otf2 \
~/Bottlenecks/heat/scorep_heat-MPI-overlap_small/traces.otf2
```

[Vampir GUI showing trace]
Case II: Trace comparison

Improved runtime
Case II: Trace comparison

Communication completely hidden by computation
Acknowledgments

- Thanks to Dirk Schmidl, RWTH Aachen, for providing the sparse matrix vector multiplication code
Review

Brian Wylie
Jülich Supercomputing Centre
Summary

You’ve been introduced to a variety of tools
- with hints to apply and use the tools effectively

Tools provide complementary capabilities
- computational kernel & processor analyses
- communication/synchronization analyses
- load-balance, scheduling, scaling, ...

Tools are designed with various trade-offs
- general-purpose versus specialized
- platform-specific versus agnostic
- simple/basic versus complex/powerful
Tool selection

Which tools you use and when you use them likely to depend on the situation
- which are available on (or for) your computer system
- which support your programming paradigms and languages
- which you are familiar (comfortable) with using
- which type of issue you suspect
- which question you want to have answered

Being aware of (potentially) available tools and their capabilities can help finding the most appropriate tools
Workflow (getting started)

First ensure that the parallel application runs correctly
- no-one will care how quickly you can get invalid answers or produce a set of corefiles
- parallel debuggers help isolate known problems
- correctness checking tools can identify other issues
- (that might not cause problems right now, but will eventually)
  - e.g., race conditions, invalid/non-compliant usage

Best to start with an overview of execution performance
- fraction of time spent in computation vs comm/synch vs I/O
- which sections of the application/library code are most costly
- Example profilers: Score-P + Cube/ParaProf, TAU

and how it changes with scale or different configurations
- processes vs threads, mappings, bindings
Communication issues generally apply to every computer system (to different extents) and typically grow with the number of processes/threads

- **Weak scaling**: fixed computation per thread, and perhaps fixed localities, but increasingly distributed
- **Strong scaling**: constant total computation, increasingly divided amongst threads, while communication grows
- **Collective communication** (particularly of type “all-to-all”) result in increasing data movement
- Synchronizations of larger groups are increasingly costly
- Load-balancing becomes increasingly challenging, and imbalances more expensive
  - generally manifests as waiting time at following collective ops
Workflow (wasted waiting time)

Waiting times are difficult to determine in basic profiles

- Part of the time each process/thread spends in communication & synchronization operations may be wasted waiting time
- Need to correlate event times between processes/threads
  - **Periscope** uses augmented messages to transfer timestamps plus on-line analysis processes
  - Post-mortem event trace analysis avoids interference and provides a complete history
  - **Scalasca** automates trace analysis and ensures waiting times are completely quantified
  - **Vampir** allows interactive exploration and detailed examination of reasons for inefficiencies
Workflow (core computation)

Effective computation within processors/cores is also vital
- Optimized libraries may already be available
- Optimizing compilers can also do a lot
  - provided the code is clearly written and not too complex
  - appropriate directives and other hints can also help
- Processor hardware counters can also provide insight
  - although hardware-specific interpretation required
- Tools available from processor and system vendors help navigate and interpret processor-specific performance issues
Technologies and their integration

- KCACHEGRIND
- PAPI
- MUST / ARCHER
- DDT
- STAT
- SYSMON / SPINDLE / SIONLIB / OPENMPI
- LWM2 / MAP / MPIP / O|SS / MAQAO / PR
- TAU
- SCORE-P
- PERISCOPE
- SCALASCA
- VAMPIR
- PARAVER
- PTF / RUBIK / MAQAO

Execution → Optimization

Debugging, error & anomaly detection

Hardware monitoring

Automatic profile & trace analysis

Visual trace analysis
Further information

Website

- Introductory information about the VI-HPS portfolio of tools for high-productivity parallel application development
  - VI-HPS Tools Guide
  - links to individual tools sites for details and download

- Training material
  - tutorial slides
  - latest ISO image of VI-HPS Linux DVD with productivity tools
  - user guides and reference manuals for tools

- News of upcoming events
  - tutorials and workshops
  - mailing-list sign-up for announcements

http://www.vi-hps.org
Acknowledgement

Many thanks to the

Texas Advanced Computing Center (TACC)

for sponsoring this tutorial by providing training accounts and compute time!