Principles and Practice of Application Performance Measurement and Analysis on Parallel Systems

Lecture 2: Practical Performance Analysis and Tuning

1. July 2011 | Bernd Mohr
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

CONTENT

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems
Fall-back: Home-grown Performance Tools

- If performance analysis and tuning tools are:
  - not available
  - too complex and complicated
  it is still possible to do some simple measurements

- Time Measurements
  - gettimeofday()
  - clock_gettime()
  - ...

- Hardware Performance Counter Measurements
  - PAPI

Timer: gettimeofday()

- UNIX function
- Returns wall-clock time in seconds and microseconds
- Actual resolution is hardware-dependent
- Base value is 00:00 UTC, January 1, 1970
- Some implementations also return the timezone

```c
#include <sys/time.h>

struct timeval tv;
double walltime; /* seconds */

gmtimeofday(&tv, NULL);
walltime = tv.tv_sec + tv.tv_usec * 1.0e-6;
```
Timer: clock_gettime()

- POSIX function
- For clock_id CLOCK_REALTIME returns wall-clock time in seconds and nanoseconds
- More clocks may be implemented but are not standardized
- Actual resolution is hardware-dependent

```
#include <time.h>

struct timespec tv;
double walltime; /* seconds */

clock_gettime(CLOCK_REALTIME, &tv);
walltime = tv.tv_sec + tv.tv_nsec * 1.0e-9;
```

Timer: getrusage()

- UNIX function
- Provides a variety of different information
  - Including user time, system time, memory usage, page faults, etc.
  - Information provided system-dependent!

```
#include <sys/resource.h>

struct rusage ru;
double usrtime; /* seconds */
int memused;

getrusage(RUSAGE_SELF, &ru);
usrtime = ru.ru_utime.tv_sec +
          ru.ru_utime.tv_usec * 1.0e-6;
memused = ru.ru_maxrss;
```
Timer: Others

- MPI provides portable MPI wall-clock timer
  ```
  #include <mpi.h>
  double walltime; /* seconds */
  walltime = MPI_Wtime();
  ```
  - Not required to be consistent/synchronized across ranks!
- Same for OpenMP 2.0 (!) programming
  ```
  #include <omp.h>
  double walltime; /* seconds */
  walltime = omp_get_wtime();
  ```
- Hybrid MPI/OpenMP programming?
  - Interactions between both standards (yet) undefined

Timer: Others

- Fortran 90 intrinsic subroutines
  - cpu_time()
  - system_clock()
- Hardware Counter Libraries
  - Vendor APIs (PMAPI, HWPC, libhpm, libpfm, libperf, ...)
  - PAPI
What Are Performance Counters

- Extra logic inserted in the processor to count specific events
- Updated at every cycle

**Strengths**
- Non-intrusive
- Very accurate
- Low overhead

**Weaknesses**
- Provides only hard counts
- Specific for each processor
- Access is not appropriate for the end user nor well documented
- Lack of standard on what is counted

Hardware Counters Interface Issues

- **Kernel level issues**
  - Handling of overflows
  - Thread accumulation
  - Thread migration
  - State inheritance
  - Multiplexing
  - Overhead
  - Atomicity

- **Multi-platform interfaces**
  - The Performance API - PAPI
    - University of Tennessee, USA
  - LIKWID
    - University of Erlangen, Germany
Hardware Measurement

- Typical measured events account for:
  - Functional units status
    - Float point operations
    - Fixed point operations
    - Load/stores
  - Access to memory hierarchy
  - Cache coherence protocol events
  - Cycles and instructions counts
  - Speculative execution information
    - Instructions dispatched
    - Branches mispredicted

Hardware Metrics

- Typical Hardware Counter
  - Cycles / Instructions
  - Floating point instructions
  - Integer instructions
  - Load/stores
  - Cache misses
  - TLB misses

- Useful derived metrics
  - IPC - instructions per cycle
  - Float point rate
  - Computation intensity
  - Instructions per load/store
  - Load/stores per cache miss
  - Cache hit rate
  - Loads per load miss
  - Loads per TLB miss

- Derived metrics allow users to correlate the behavior of the application to one or more of the hardware components
- One can define threshold values acceptable for metrics and take actions regarding program optimization when values are below/above the threshold
### Accuracy Issues

- **Granularity of the measured code**
  - If not sufficiently large enough, overhead of the counter interfaces may dominate

- **Pay attention to what is not measured:**
  - Out-of-order processors
  - Sometimes speculation is included
  - Lack of standard on what is counted
    - Microbenchmarks can help determine accuracy of the hardware counters

### Hardware Counters Access on Linux

- **Linux** had not defined an out-of-the-box interface to access the hardware counters!

  - **Linux Performance Monitoring Counters Driver (PerfCtr)** by Mikael Pettersson from Uppsala X86 + X86-64
    - Needs kernel patching!
    - [http://user.it.uu.se/~mikpe/linux/perfctr/](http://user.it.uu.se/~mikpe/linux/perfctr/)

  - **Perfmon** by Stephane Eranian from HP – IA64
    - It was being evaluated to be added to Linux

- **Linux 2.6.31**
  - Performance Counter subsystem provides an abstraction of special performance counter hardware registers
Utilities to Count Hardware Events

- There are utilities that start a program and at the end of the execution provide **overall** event counts
  - hpmcount (IBM)
  - CrayPat (Cray)
  - pfmon from HP (part of Perfmon for Al64)
  - psrun (NCSA)
  - cputrack, sstrun (SGI)
  - perf (Linux 2.6.31)

Hardware Counters: PAPI

- Parallel Tools Consortium (PTools) sponsored project
- **Performance Application Programming Interface**
- Two interfaces to the underlying counter hardware:
  - The **high-level interface** simply provides the ability to start, stop and read the counters for a specified list of events
  - The **low-level interface** manages hardware events in user defined groups called **EventSets**
- Timers and system information
- C and Fortran bindings
- Experimental PAPI interface to performance counters support in the Linux 2.6.31 kernel
PAPI Architecture

Portable Layer
- PAPI Low Level
- PAPI High Level

Machine Specific Layer
- PAPI Machine Dependent Substrate
- Kernel Extensions
- Operating System
- Hardware Performance Counters

PAPI Predefined Events

- Common set of events deemed relevant and useful for application performance tuning (wish list)
  - papiStdEventDefs.h
  - Accesses to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit and pipeline status
  - Run PAPI papi_avail utility to determine which predefined events are available on a given platform
  - Semantics may differ on different platforms!

- PAPI also provides access to native events on all supported platforms through the low-level interface
  - Run PAPI papi_native_avail utility to determine which predefined events are available on a given platform
PAPI avail Utility

% papi_avail -h
This is the PAPI avail program.
It provides availability and detail information
for PAPI preset and native events. Usage:

papi_avail [options] [event name]
papi_avail TESTS_QUIET

options:
--a         display only available PAPI preset events
--d         display PAPI preset event info in detailed format
--e EVENTNAME display full detail for named preset or native event
--h         print this help message
--t         display PAPI preset event info in tabular format (default)

PAPI Preset Listing

(derose@jaguar1) 184% papi_avail
LibLustre: NAL NID: 0005dc02 (2)
Lustre: OBD class driver Build Version: 1, info@clusterfs.com
-------------------------------------------------------------------------
Vendor string and code   : AuthenticAMD (2)
Model string and code    : AMD K8 (13)
CPU Revision             : 1.000000
CPU Megahertz            : 2400.000000
CPU's in this Node       : 1
Nodes in this System     : 1
Total CPU's              : 1
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------------------------------------
Name            Code            Available Derivative Description (Note)
PAPI_L1_DCM     0x80000000      Yes     Yes     Level 1 data cache misses (O)
PAPI_L1_ICM     0x80000001      Yes     Yes     Level 1 instruction cache misses (O)
PAPI_L2_DCM     0x80000002      Yes     No      Level 2 data cache misses (O)
PAPI_L2_ICM     0x80000003      Yes     No      Level 2 instruction cache misses (O)
PAPI_L3_DCM     0x80000004      No      No      Level 3 data cache misses (O)
PAPI_L3_ICM     0x80000005      No      No      Level 3 instruction cache misses (O)
PAPI_L1_TCM     0x80000006      Yes     Yes     Level 1 cache misses (O)
PAPI_L2_TCM     0x80000007      Yes     Yes     Level 2 cache misses (O)
PAPI_L3_TCM     0x80000008      No      No      Level 3 cache misses (O)
...
Example: papi_avail –e PAPI_L1_TCM (AMD Opteron)

- Event name: PAPI_L1_TCM
- Event Code: 0x80000006
- Number of Native Events: 4
- Short Description: L1 cache misses
- Long Description: Level 1 cache misses
- Developer's Notes: 

  Derived Type: DERIVED_ADD

  Postfix Processing String: DC_SYS_REFILL_NOES

  Native Code[0]: 0x40001e1c
  Register[0]: 0x20f
  Register[1]: 0x1e43
  Native Event Description: Refill from system. Cache bits: Modified Owner Exclusive Shared

  Native Code[1]: 0x40000037
  Register[0]: 0xf
  Register[1]: 0x83
  Native Event Description: Refill from system

  Native Code[2]: 0x40000036
  Register[0]: 0xf
  Register[1]: 0x82
  Native Event Description: Refill from L2

PAPI papi_native_avail Utility (AMD Opteron)

- Test case NATIVE_AVAIL: Available native events and hardware information.

  Vendor string and code: AuthenticAMD (2)
  Model string and code: AMD K8 Revision C (15)
  CPU Revision: 10.000000
  CPU Megahertz: 2193.406982
  CPU's in this Node: 2
  Nodes in this System: 1
  Total CPU's: 2
  Number Hardware Counters: 4
  Max Multiplex Counters: 32

  The following correspond to fields in the PAPI_event_info_t structure.

  Symbol     Event Code     Count
  Short Description
  Long Description
  Derived
  PostFix

  The count field indicates whether it is a) available (count >= 1) and b) derived (count > 1)

  FP_ADD_PIPE 0x40000000
  | Dispatched FPU ops - Revision B and later revisions - Speculative add pipe ops excluding junk ops|
  | Register Value[0]: 0xf      P3 Ctrl Mask|
  | Register Value[1]: 0x100    P3 Ctrl Code|

  FP_MULT_PIPE 0x40000001
  | Dispatched FPU ops - Revision B and later revisions - Speculative multiply pipe ops excluding junk ops|
  | Register Value[0]: 0xf      P3 Ctrl Mask|
  | Register Value[1]: 0x200    P3 Ctrl Code|

© 2011 Bernd Mohr
**High Level API**

- Meant for application programmers wanting simple but accurate measurements
- Calls the lower level API
- Allows only PAPI preset events
- Eight functions:
  - `PAPI_num_counters`
  - `PAPI_start_counters`, `PAPI_stop_counters`
  - `PAPI_read_counters`
  - `PAPI_accum_counters`
  - `PAPI_flops`
  - `PAPI_flips`, `PAPI_ipc` (New in Version 3.x)
- Not thread-safe (Version 2.x)

---

**Example: Quick and Easy Mflop/s**

```fortran
program papiMflops
  parameter (N=1024)
  include "f77papi.h"
  integer*8 fpins
  real*4 realtm, cputime, mflops
  integer ierr
  real*4 a(N,N)

  call random_number(a)
  call PAPIF_flops(realtm, cputime, fpins, mflops, ierr)
  do j=1,N
    do i=1,N
      a(i,j)=a(i,j)*a(i,j)
    end do
  end do
  call PAPIF_flops(realtm, cputime, fpins, mflops, ierr)
  print *, 'realtime: ', realtm, ' cputime: ', cputime
  print *, 'papi_flops: ', mflops, ' MFlops'
end
```

% ./papiMflops
realtime: 3.640159 cputime: 3.630502
papi_flops: 29.67809  MFlops
**General Events**

```fortran
program papicount
    parameter (N=1024)
    include "f77papi.h"
    integer*8 values(2)
    integer events(2), ierr
    real*4 a(N,N)
    call random_number(a)
    events(1) = PAPI_L1_DCM
    events(2) = PAPI_L1_DCA
    call PAPIF_start_counters(events, 2, ierr)
    do j=1,N
        do i=1,N
            a(i,j)=a(i,j)*a(i,j)
        end do
    end do
    call PAPIF_read_counters(values, 2, ierr)
    print *,' L1 data misses  : ', values(1)
    print *,' L1 data accesses: ', values(2)
end
```

```
% ./papicount
L1 data misses  :                  13140168
L1 data accesses:                 500877001
```

**Low Level API**

- Increased efficiency and functionality over the high level PAPI interface
- 54 functions
- Access to native events
- Obtain information about the executable, the hardware, and memory
- Set options for multiplexing and overflow handling
- System V style sampling (profil())
- Thread safe
CONTENT

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems

MPI Profiling: mpiP

- Scalable, light-weight MPI profiling library
- Generates detailed text summary of MPI behavior
  - Time spent at each MPI function callsite
  - Bytes sent by each MPI function callsite (where applicable)
  - MPI I/O statistics
  - Configurable traceback depth for function callsites
- Controllable from program using MPI_Pcontrol
  - Allows you to profile just one code module or cycle
  - Allows mpiP profile dumps mid-run
- Uses PMPI interface ⇒ only re-link of application necessary

http://mpip.sourceforge.net/
mpiP Text Output Example

@ mpiP
@ Version: 3.1.1
// 10 lines of mpiP and experiment configuration options
// 8192 lines of task assignment to BlueGene topology information
@--- MPI Time (seconds) -----------------------------------------------
Task | AppTime | MPITime | MPI%
0    | 37.7    | 25.2    | 66.89
8191 | 37.6    | 26      | 69.21
*    | 3.09e+05| 2.04e+05| 65.88
@--- Callsites: 26 --------------------------------------------------
ID Lev File/Address       Line Parent_Funct            MPI_Call
1   0 coarsen.c         542 hypre_StructCoarsen      Waitall
// 25 similar lines
@--- Aggregate Time (top twenty, descending, milliseconds)---------
Call | Site | Time | App% | MPI% | COV
Waitall | 21   | 1.03e+08 | 33.27 | 50.49 | 0.11
Waitall | 1    | 2.88e+07  | 9.34  | 14.17 | 0.01
// 18 similar lines
@--- Aggregate Sent Message Size (top twenty, descending, bytes)---
Call | Site | Count | Total       | Avrg | Sent%
Isend | 11   | 845594460 | 7.71e+11  | 912  | 59.92
Allreduce | 10   | 49152  | 3.93e+05   | 8    | 0.00
// 6 similar lines
@--- Callsite Time statistics (all, milliseconds): 212992 -------
Name | Site Rank | Count | Max | Mean | Min | App% | MPI%
Waitall | 21    | 111096 | 275 | 0.1 | 0.000707 | 29.61 | 44.27
Waitall | 21    | 65799  | 882 | 0.24 | 0.000707 | 41.98 | 60.66
// 213,042 similar lines
@--- Callsite Message Sent statistics (all, sent bytes) ----------
Name | Site Rank | Count | Max | Mean | Min | Sum
Isend | 11    | 0 | 72917  | 2.621e+05 | 851.1  | 8 | 6.206e+07
Isend | 11    | 8191 | 46651  | 2.621e+05 | 1029    | 8 | 4.801e+07
Isend | * 84594460 | 2.621e+05 | 911.5  | 8 | 7.708e+11
// 65,550 similar lines

mpiP Text Output Example (cont.)
“Swiss Army Knife” of Performance Analysis: TAU

- Very portable tool set for instrumentation, measurement and analysis of parallel multi-threaded applications
- Instrumentation API supports choice
  - between profiling and tracing
  - of metrics (i.e., time, HW Counter (PAPI))
- Uses Program Database Toolkit (PDT) for C, C++, Fortran source code instrumentation
- Supports
  - Languages: C, C++, Fortran 77/90, HPF, HPC++, Java, Python
  - Threads: pthreads, Tulip, SMARTS, Java, Win32, OpenMP
  - Systems: same as KOJAK + Windows + MacOS + …

http://tau.uoregon.edu/
http://www.cs.uoregon.edu/research/pdt/

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (PDT))
      - OpenMP (directive rewriting with Opari)
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPJ)
    - statically-linked and dynamically-loaded (e.g., Python)
  - Executable code
    - dynamic instrumentation (pre-execution) (DynInst)
    - virtual machine instrumentation (e.g., Java using JVMPJ)
- Support for performance mapping
- Support for object-oriented and generic programming
**TAU Usage**

- **Usage:**
  - Specify programming model by setting `TAU_MAKEFILE` to one of `$<TAU_ROOT>/<arch>/lib/Makefile.tau-*`.  
  - Examples from Linux cluster with Intel compiler and PAPI:
    - **MPI:** `Makefile.tau-icpc-papi-mpi-pdt`
    - **OMP:** `Makefile.tau-icpc-papi-pdt-openmp-opari`
    - **OMPI:** `Makefile.tau-icpc-papi-mpi-pdt-openmp-opari`
  - Compile and link with
    - `tau_cc.sh file.c ...`
    - `tau_cxx.sh file.cxx...`
    - `tau_f90.sh file.f90 ...`
  - Execute with real input data
    - Environment variables control measurement mode
      - `TAU_PROFILE`, `TAU_TRACE`, `TAU_CALLPATH`, ...
  - Examine results with `paraprof`

---

**Vampirtrace MPI Tracing Tool**

- **Library for Tracing of MPI and Application Events**
  - Records MPI-1 point-to-point and collective communication
  - Records MPI-2 I/O operations and RMA operations
  - Records user subroutines

- **Uses the standard MPI profiling interface**

- **Usage:**
  - Compile and link with
    - `vtcc -vt:cc mpircc file.c ...
    - `vtcxx -vt:cxx mpircc file.cxx...
    - `vtf90 -vt:f90 mpirf90 file.f90 ...
  - Execute with real input data (generates `<exe>.otf`)
Vampirtrace MPI Tracing Tool

- Versions up to 4.0 until 2003 were commercially distributed by PALLAS as VampirTrace.
- Current status
  - **Commercially** distributed by Intel
    - Version 5 and up distributed as Intel Trace Collector
    - For Intel based platforms only
  - **New open-source** VampirTrace version 5 distributed by Technical University Dresden
    - Based on KOJAK’s measurement system with OTF backend
    - http://www.tu-dresden.de/zh/vampirtrace/
    - Is also distributed as part of Open MPI

Vampir Event Trace Visualizer

- Visualization and Analysis of MPI Programs
- Commercial product
- Originally developed by Forschungszentrum Jülich
- Now all development by Technical University Dresden
Vampir Event Trace Visualizer

- Versions up to 4.0 until 2003 were commercially distributed by PALLAS as Vampir
- Current status
  - Commercially distributed by Intel
    - Version 4 distributed as Intel Trace Analyzer
    - For Intel based platforms only
    - Intel meanwhile released own new version (ITA V6 and up)
  - Original Vampir (and new VNG) commercially distributed by Technical University Dresden
    - http://www.vampir.eu

Vampir: Time Line Diagram

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective and I/O operations available through clicking on the representation
Vampir: Process and Counter Timelines

- Process timeline show call stack nesting
- Counter timelines for hardware or software counters

Vampir: Execution Statistics

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive
- Available for all / any group (activity) or all routines (symbols)
- Available for any part of the trace ⇒ selectable through time line diagram
Vampir: Process Summary

- Execution statistics over all processes for comparison
- Clustering mode available for large process counts

Vampir: Communication Statistics

- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics
- Available for any part of the trace
### Other Profiling Tools

- **gprof**
  - Available on many systems
  - Compiler instrumentation (invoked via `–g –pg`)

- **FPMPI-2 (ANL)**
  - MPI profiler (invoked via re-linking)
  - **special**: Optionally identifies synchronization time
  - Single output file: count, sum, avg, min, max over ranks

- **ThreadSpotter (Rogue Wave) [commercial product]**
  - Sampling-based memory and thread performance analyzer
  - Works on un-modified, optimized executables

### Other Profiling Tools II

- **ompP (UC Berkeley)**
  - [http://www.ompp-tool.com](http://www.ompp-tool.com)
  - OpenMP profiler (invoked via OPARI source instrumentation)

- **HPCToolkit (Rice University)**
  - [http://www.hpctoolkit.org](http://www.hpctoolkit.org)
  - Multi-platform sampling-based callpath profiler
  - Works on un-modified, optimized executables

- **Open|SpeedShop (Krell Insitute with support of LANL, SNL, LLNL)**
  - [http://www.openspeedshop.org](http://www.openspeedshop.org)
  - Comprehensive performance analysis environment
  - Uses binary instrumentation (via DynInst)
Other Tracing Tools

- **MPE / Jumpshot** (ANL)
  - Part of MPICH2
  - Invoked via re-linking
  - Only supports MPI P2P and collectives; SLOG2 trace format

- **Extrae / Paraver** (BSC/UPC)
  - [http://www.bsc.es/paraver](http://www.bsc.es/paraver)
  - Measurement system (Extrae) and visualizer (Paraver)
  - Powerful filter and summarization features
  - Very configurable visualization

CONTENT

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- **Practical Performance Analysis and Tuning**
  - Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems
Practical Performance Analysis and Tuning

- Successful tuning is combination of
  - Right algorithm and libraries
  - Compiler flags and pragmas / directives
    (Learn and use them)
  - THINKING

- Measurement is better than reasoning / intuition (= guessing)
  - To determine performance problems
  - To validate tuning decisions / optimizations
    (after each step!)

Practical Performance Analysis and Tuning

- It is easier to optimize a slow correct program than to debug a fast incorrect one
  - Debugging before Tuning
  - Nobody really cares how fast you can compute the wrong answer

- The 80/20 rule
  - Program spends 80% time in 20% of code
  - Programmer spends 20% effort to get 80% of the total speedup possible in the code
  - Know when to stop!

- Don’t optimize what doesn’t matter
  - Make the common case fast
Typical Performance Analysis Procedure

1. **Do I have a performance problem at all?**
   - Time / hardware counter measurements
   - Speedup and scalability measurements
2. **What is the main bottleneck (computation/communication...)?**
   - Flat profiling (sampling / prof)
3. **Where is the main bottleneck?**
   - Call graph profiling (gprof)
   - Detailed (basic block) profiling
4. **Why is it there?**
   - Hardware counters analysis
   - Trace selected parts to keep trace files manageable
5. **Does my code have scalability problems?**
   - Load Imbalance analysis
   - Profile code for typical small and large processor count
   - Compare profiles function-by-function

**CONTENT**

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems
Heterogenous Systems

- Current trend to use hardware acceleration to speedup calculations
  - IBM Cell (e.g. LANL Roadrunner)
  - Clearsped
  - FPGA-based acceleration
  - GPU-based acceleration

- In the long run: new programming models needed

- Very little to not existing tool support
  ⇒ Tool research opportunities!

GPU Performance Tools

- CUDA profiler

- TAU
  - CUDA + OpenCL profiling (host side)

- VampirTrace
  - CUDA tracing (host side)
Current Related Projects

- Hybrid Programming for Heterogeneous Architectures
  - EU ITEA2 funded project 2010 – 2013
  - Successor of successful ParMA project
  - 25 partners for France, Germany, Spain, Sweden including Bull, TUD, BSC, USQV, JSC, Acumem
  - Develop programming models and tools
  - Evaluation with large set of industrial codes

CONTENT

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems
“A picture is worth 1000 words…”

- MPI class example: Send messages in a ring program

“What about 1000’s of pictures?”
(with 100’s of menu options)
Example Automatic Analysis: Late Sender

Example Automatic Analysis (2): Wait at NxN
Basic Idea Automatic Performance Analysis

- “Traditional” Tool
  - Huge amount of Measurement data
  - For non-standard / tricky cases (10%)
  - For expert users
  → More productivity for performance analysis process!

- Automatic Tool
  - Simple: 1 screen + 2 commands + 3 panes
  - For standard cases (90% ?!) For "normal" users
  - Starting point for experts

The KOJAK Project

- Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks
- Forschungszentrum Jülich
- Innovative Computing Laboratory, TN
- Started 1998

- Approach
  - Instrument C, C++, and Fortran parallel applications
    - Based on MPI, OpenMP, SHMEM, or hybrid
  - Collect event traces
  - Search trace for event patterns representing inefficiencies
  - Categorize and rank inefficiencies found

- http://www.fz-juelich.de/jsc/kojak/
Principles and Practice of Application Performance Measurement and Analysis on Parallel Systems

Moscow
July 2011

© 2011 Bernd Mohr
CONTENT

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems

Beyond Execution Time and Flops

- High performance on today’s architectures requires extremely effective use of the cache and memory hierarchy of the system
  - Very complex designs
  - Will get “worse” with advanced multi- and many core chips
- tools for memory performance analysis needed
  - Current approach:
    - Measure cache, memory, TLB events (per execution unit)
    - Rarely tell what data object(s) are cause of the problem
  - Main problem:
    - if cause known, what is the fix?
    - Or worse, what is the portable fix?
  - Should be really better left to the compiler!
Beyond Execution Time and Flops II

- Do we need tools for I/O performance analysis?
  - Currently only very few tools available (CrayPat, Pablo I/O, …)
  - Perhaps scientific programmers only need training in parallel I/O facilities already available today?

- Tools for new programming paradigms?
  - Obvious next candidate: one-sided communication
    - SHMEM, MPI-2 RMA, ARMCI, …
      - “Easy” to monitor; intercept calls e.g., using wrapperfuncs
    - CAF, UPC
      - “Harder”; probably need compiler support
  - Other candidates?

Beyond Execution Time and Flops III

- Tool (sets and environments) must be able to handle “hybrid” cases!
  - Message passing (MPI)
  - Multi-threading (OpenMP, pthreads, …)
  - One-sided communication
  - (Parallel) I/O
  - …

- BIGGEST challenge:
  - Many tools for instrumentation, measurement, analysis, and visualization
  - What about (automatic) optimization tools?
CONTENTS

- Fall-back
  - Simple timers
  - Hardware counter measurements
- Overview of some performance tools
  - mpiP, TAU, Vampir + Vampirtrace
- Practical Performance Analysis and Tuning
- Challenges and open problems in performance optimization
  - Heterogeneous systems
  - Automatic performance analysis: KOJAK/Scalasca
  - Beyond execution time and flops
  - Extremely large HPC systems

Increasing Importance of Scaling

Number of Processors share for TOP 500 Jun 2011

<table>
<thead>
<tr>
<th>NProc</th>
<th>Count</th>
<th>Share</th>
<th>( \Sigma \text{Rmax} )</th>
<th>Share</th>
<th>( \Sigma \text{NProc} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025-2048</td>
<td>2</td>
<td>0.4%</td>
<td>168 TF</td>
<td>0.3%</td>
<td>2,632</td>
</tr>
<tr>
<td>2049-4096</td>
<td>20</td>
<td>4.0%</td>
<td>1,177 TF</td>
<td>2.0%</td>
<td>71,734</td>
</tr>
<tr>
<td>4097-8192</td>
<td>195</td>
<td>39.0%</td>
<td>9,759 TF</td>
<td>16.6%</td>
<td>1,262,738</td>
</tr>
<tr>
<td>8193-16384</td>
<td>224</td>
<td>44.8%</td>
<td>15,216 TF</td>
<td>25.8%</td>
<td>2,337,998</td>
</tr>
<tr>
<td>&gt; 16384</td>
<td>59</td>
<td>11.8%</td>
<td>32,556 TF</td>
<td>55.3%</td>
<td>4,100,234</td>
</tr>
<tr>
<td>Total</td>
<td>500</td>
<td>100%</td>
<td>58,876 TF</td>
<td>100%</td>
<td>7,775,336</td>
</tr>
</tbody>
</table>

- Average system size: 15,551 cores
- Median system size: 8,556 cores
Increasing Importance of Scaling II

- Number of Processors share for TOP 500 Jun 2001 – Jun 2011

Projection for a Exascale System*

<table>
<thead>
<tr>
<th>System attributes</th>
<th>2010</th>
<th>“2015”</th>
<th>“2018”</th>
<th>Difference 2010 &amp; 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>System peak</td>
<td>2 Pflop/s</td>
<td>200 Pflop/s</td>
<td>1 Eflop/sec</td>
<td>O(1000)</td>
</tr>
<tr>
<td>Power</td>
<td>6 MW</td>
<td>15 MW</td>
<td>~20 MW</td>
<td></td>
</tr>
<tr>
<td>System memory</td>
<td>0.3 PB</td>
<td>5 PB</td>
<td>32-64 PB</td>
<td>O(100)</td>
</tr>
<tr>
<td>Node performance</td>
<td>125 GF</td>
<td>0.5 TF</td>
<td>7 TF</td>
<td>10 TF</td>
</tr>
<tr>
<td>Node memory BW</td>
<td>25 GB/s</td>
<td>0.1 TB/sec</td>
<td>1 TB/sec</td>
<td>0.4 TB/sec</td>
</tr>
<tr>
<td>Node concurrency</td>
<td>12</td>
<td>O(100)</td>
<td>O(1,000)</td>
<td>O(1,000)</td>
</tr>
<tr>
<td>Total Concurrency</td>
<td>225,000</td>
<td>O(10⁹)</td>
<td>O(10⁹)</td>
<td>O(10,000)</td>
</tr>
<tr>
<td>Total Node Interconnect BW</td>
<td>1.5 GB/s</td>
<td>20 GB/sec</td>
<td>200 GB/sec</td>
<td>O(100)</td>
</tr>
<tr>
<td>MTTI</td>
<td>days</td>
<td>O(1 day)</td>
<td>O(1 day)</td>
<td>- O(10)</td>
</tr>
</tbody>
</table>

* From http://www.exascale.org
Extremely large HPC systems

- Many HPC computing centers have systems with some 1000s of PEs
  - Today’s tools (e.g., gprof, Vampir, TAU) can handle 500–1000 PEs
  - Measurements give enough insight to optimize for 2000–4000 PEs

- Now: IBM BlueGene, Cray XT5 ⇒ some 100,000s PEs
  - Small customer base
  - Very unique performance problems:
    - Why does it work on 16,000 PEs but not for 32,000 PEs?
  - Load balance problems amplify performance degradation
  - Interesting challenging problem, but given scarce resources in tools research? …
    - Lots of data ⇒ need better data mgmt, parallel analysis!? 
    - Bigger problem: scalable result displays / visualizations

VampirServer Architecture

- Large parallel application
- File system
- Vampir analysis server
- Vampir visualization client

- Monitoring system
- Event streams
- Parallel I/O
- Message passing

- Timeline window
- Current window
- Full trace outline
VampirServer:
PEPC, 16384 PEs, Global Timeline

VampirServer:
PEPC, 16384 PEs, Global Timeline (zoomed)
VampirServer: PEPC, 16384 PEs, Message Statistics

VampirServer: PEPC, 16384 PEs, Cluster Analysis
Vampirserver

- Parallel client/server version of Vampir
  - Can handle much larger trace files
  - Remote visualization possible

Usage
- Start VampirServer daemon
  - `vngd -n <t>` # on frontend, uses `<t>` threads
  - `mpiexec -n <p> vngd` # cluster, uses `<p>` processes
- Start the Vampir visualizer (`vampir`)
  - on local system if available
  - on frontend in 2nd shell
  - then connect client to server, load and analyze trace file `<exe>.otf`
- finally: `vngd -shutdown`

The Scalasca Project

- Scalable Analysis of Large Scale Applications
- Follow-up project to KOJAK [http://www.scalasca.org/](http://www.scalasca.org/)

Approach
- **Instrument** C, C++, and Fortran parallel applications
  - Based on MPI, OpenMP, SHMEM, or hybrid
- **Option 1: scalable call-path profiling**
- **Option 2: scalable event trace analysis**
  - **Collect** event traces
  - **Search** trace for event patterns representing inefficiencies
  - **Categorize and rank** inefficiencies found
- Supports MPI 2.2 and basic OpenMP
Sequential Analysis Process (KOJAK)

- Measurement library
- Instrumented process
- Instrumented executable
- Multi-level instrumenter
- Source modules
- Local event traces
- Unification+ Merge
- Global trace
- Sequential pattern search
- Pattern trace
- Conversion
- Exported trace
- Report manipulation
- TAU paraprof
- CUBE Report explorer
- TAU paraprof
- CUBE Report explorer
- Vampir or Paraver

New Parallel Analysis Process I (Scalasca)

- New enhanced measurement library
- Instrumented process
- Instrumented executable
- Multi-level instrumenter
- Source modules
- Local event traces
- Merge
- Global trace
- Sequential pattern search
- Pattern trace
- Conversion
- Exported trace
- Summary report
- Optimized measurement configuration
- TAU paraprof
- CUBE Report explorer
- Vampir or Paraver

= Third-party component
Scalasca Summary Analysis sweep3D@294,912

New scalable machine topology display

New Parallel Analysis Process II (Scalasca)

- New enhanced measurement library
- Instrumented process
- Instrumented executable
- Multi-level instrumenter
- Source modules

- Optimized measurement configuration
- Local event traces
- Parallel pattern search
- Pattern report
- Summary report
- Report manipulation

- TAU paraprof
- CUBE Report explorer
- Vampir or Paraver

= Third-party component

© 2011 Bernd Mohr
Scalable Automatic Trace Analysis

- Parallel pattern search to address wide traces
  - As many processes / threads as used to run the application
    - Can run in same batch job!!
  - Each process / thread responsible for its "own" local trace data
- Idea: "parallel replay" of application
  - Analysis uses communication mechanism that is being analyzed
  - Use MPI P2P operation to analyze MPI P2P communication, use MPI collective operation to analyze MPI collectives, ...
  - Communication requirements not significantly higher and (often lower) than requirements of target application
- In-memory trace analysis
  - Available memory scales with number of processors used
  - Local memory usually large enough to hold local trace

Example: Late Sender

- Sequential approach (EXPERT)
  - Scan the trace in sequential order
  - Watch out for receive event
  - Use links to access other constituents
  - Calculate waiting time

- New parallel approach (SCOUT)
  - Each process identifies local constituents
  - Sender sends local constituents to receiver
  - Receiver calculates waiting time
Scalasca Trace Analysis sweep3D@294,912

- 10 min sweep3D runtime
- 11 sec replay
- 4 min trace data write/read (576 files)
- 7.6 TB buffered trace data
- 510 billion events

TAU ParaProf: 3D Profile, Miranda, 16K PEs

Height and color can indicate different metrics
Performance Tuning: Still a Problem?

Further Documentation

- http://www.vi-hps.org/training/material/
  - Performance Tools LiveDVD image
  - Links to tool websites and documentation
  - Tutorial slides