CS 594: Scientific Computing for Engineers

Topic: Performance Analysis Tools

Heike Jagode
jagode@eeecs.utk.edu

With slides from Karl Fuerlinger, Andreas Knuepfer, Shirley Moore, Sameer Shende, Felix Wolf and others.

Outline

1. Motivation
   - What is Performance?
   - Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

3. Performance Analysis Tools
   - PAPI: Access to hardware performance counters
   - Vampir Suite: Instrumentation and Trace visualization
   - KOJAK / Scalasca: automatic performance analysis tool
   - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
Think carefully about the prevalent idea that only delivering results counts as acceptable performance. If you don’t reach the objectives, maybe you have not performed well enough?

The idea that the design of software consists of is NOT simply implementing it instead, it consists of designing its functional aspects first, then implementing, and finally trying to improve the low-quality product of this procedure.

As long as we don’t know what Performance is, it will produce confusion and frustration.

☐ Simply, getting a job done?
☐ Producing results that you aimed at?
☐ Nothing else matters?
☐ Basically, solving a problem themed: “hack now, fix later”?

☑ Think carefully about the prevalent idea that only delivering results counts as acceptable performance
☑ If you don’t reach the objectives, maybe you have not performed well enough?

☑ The idea that the design of software consists of is NOT simply implementing it
☑ Instead, it consists of designing its functional aspects first, then implementing, and finally trying to improve the low-quality product of this procedure.
Why Performance Analysis?

1/2

- Domenico Ferrari (1986):
  “The study of performance evaluation as an independent subject has sometimes caused researchers in the area to lose contact with reality.”

- Why is it that performance evaluation is by no means an integrated and natural part of software development?
  - The primary duty of software developers is to create functionally correct programs!
  - Performance evaluation tends to be optional
    - Some people compare it to the freestyle event in ice-skating
  - Raj Jain (1991)
    “Contrary to common belief, performance evaluation is an art. ... Like artist, each analyst has a unique style. Given the sample problem, two analysts may choose different performance metrics and evaluation methodologies.” … but even they need tools!

Performance Analysis is important:

- Large investments in HPC systems
  - Procurement costs: ~$40 Mio
  - Operational costs: ~$5 Mio per year
  - Electricity costs: 1 MW / year ~$1 Mio

- Efficient usage is important because of expensive and limited resources
- Scalability is important to achieve next bigger simulation

- Performance analysis: Get highest performance for a given cost
- „Performance Analyst“: Anyone who is associated with computer systems,
- i.e. system engineers, computer scientists and of course users

2/2
1. Motivation
   - What is Performance?
   - Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

3. Performance Analysis Tools
   - PAPI: Access to hardware performance counters
   - Vampir Suite: Instrumentation and Trace visualization
   - KOJAK / Scalasca: automatic performance analysis tool
   - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications

Performance Optimization cycle:

Measure & Analyze:
   - Have an optimization phase
   - just like testing & debugging phase
   - Do profiling and tracing
   - Use tools!
   - avoid do-it-yourself with printf solutions
   - ... seriously!
**Measurement: Profiling vs Tracing**

**Profiling**
- Records aggregated information of performance metrics
- Number of times a routine was invoked
- Exclusive, inclusive time/counts spent executing it
- Number of instrumented child routines invoked, etc.
- Structure of invocations (call-trees/call-graphs)
- Memory, message communication sizes

**Tracing**
- When and where events took place along a global timeline
- Time-stamped events (points of interest)
- Message communication events (sends/receives) are tracked
- Shows when and from/to where messages were sent
- Event Trace: collection of all events of a process/program sorted by time

**Measurement: Profiling**

- Recording of summary information during execution
  - inclusive, exclusive time, # calls, hardware counter statistics, ...
- Reflects performance behavior of program entities
  - functions, loops, basic blocks
  - user-defined “semantic” entities
- Very good for low-cost performance assessment
- Helps to expose performance bottlenecks and hotspots
- Implemented through either
  - **sampling**: periodic OS interrupts or hardware counter traps
  - **measurement**: direct insertion of measurement code
### Profiling: Inclusive vs Exclusive

```c
int main( )
{ /* takes 100 secs */
    f1(); /* takes 20 secs */
    /* other work */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */
    /* other work */
}
/* similar for other metrics, such as hardware performance counters, etc. */
```

- **Inclusive time for main:**
  100 secs

- **Exclusive time for main:**
  100
  - 20
  - 50
  - 20
  =10 secs

- Exclusive time sometimes called “self”

---

### Tracing: Advantages & Disadvantages

- Recording of information about significant points (events) during program execution
  - entering/exiting code region (function, loop, block, …)
  - thread/process interactions (e.g., send/receive message)

- Save information in event record
  - Timestamp
  - CPU identifier, thread identifier
  - Event type and event-specific information

- **Event Trace:** Collection of all events of a process/program sorted by time
  Can be used to reconstruct dynamic program behavior
  - Profiles can be calculated from traces

**Tracing Disadvantages**
- traces can become very large
- instrumentation and tracing is complicated
- event buffering, clock synchronization, …
**COMMON EVENT TYPES**

- enter/leave of function/routine/region
  - time stamp, process/thread, function ID
- send/receive of P2P message (MPI)
  - time stamp, sender, receiver, length, tag, communicator
- collective communication (MPI)
  - time stamp, process, root, communicator, # bytes
- hardware performance counter values
  - time stamp, process, counter ID, value
- etc.

**PARALLEL TRACE FILE**

```
10010 P 1 ENTER 5
10090 P 1 ENTER 6
10110 P 1 ENTER 12
10110 P 1 SEND TO 3 LEN 1024 ...
10330 P 1 LEAVE 12
10400 P 1 LEAVE 6
10520 P 1 ENTER 9
10550 P 1 LEAVE 9
...
```

```
DEF TIMERRES 1000000000
DEF PROCESS 1 `Master`
DEF PROCESS 2 `Slave`
...
DEF FUNCTION 5 `main`
DEF FUNCTION 6 `foo`
DEF FUNCTION 9 `bar`
DEF FUNCTION 12 `MPI_Send`
DEF FUNCTION 13 `MPI_Recv`
...
```

```
10020 P 2 ENTER 5
10095 P 2 ENTER 6
10120 P 2 ENTER 13
10300 P 2 RECV FROM 3 LEN 1024 ...
10350 P 2 LEAVE 13
10450 P 2 LEAVE 6
10620 P 2 ENTER 9
10650 P 2 LEAVE 9
...
```
**Tracing Example:**
**Instrumentation, Monitor, Trace**

CPU A:
```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:
```c
void slave {
    trace(EXIT, 1);
    ...
    recv(A, tag, buf);
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

**Event definition**

<table>
<thead>
<tr>
<th>Event</th>
<th>CPU A</th>
<th>CPU B</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>SEND</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>EXIT</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>RECV</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

**Tracing: Timeline Visualization**

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

- **Timeline Representation:**
  - **A:** Orange
  - **B:** Green
  - **Time:** 58, 60, 62, 64, 66, 68, 70
**Performance Data Analysis**

- Draw conclusions from measured performance data

- Manual analysis
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling
  - Examples: TAU, Vampir Suite, Paraver, Intel Trace Collector & Analyzer

- Automated analysis
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca

**Automated Performance Analysis**

- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - ORNL’s Jaguar Cray XT5: 150,152 cores (2009)
  - LLNL Sequoia (IBM, based on future Blue Gene arch.): ~1.6 million cores (2011-2012)
  - Trend to multi-core

- Large amounts of performance data
  - Several gigabytes or even terabytes
  - Overwhelms user

- Not all programmers are performance experts
  - Scientists want to focus on their domain
  - Need to keep up with new machines

- Automation can solve some of these issues
This is a situation that can be detected automatically by analyzing the trace file

-> late sender pattern

Outline

1. Motivation
   - What is Performance?
   - Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

3. Performance Analysis Tools
   - PAPI: Access to hardware performance counters
   - Vampir Suite: Instrumentation and Trace visualization
   - KOJAK / Scalasca: automatic performance analysis tool
   - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
**What are Hardware Performance Counters?**

For many years, hardware engineers have designed in specialized registers to measure the performance of various aspects of a microprocessor.

HW performance counters provide application developers with valuable information about code sections that can be improved.

Hardware performance counters can provide insight into:
- Whole program timing
- Cache behaviors
- Branch behaviors
- Memory and resource contention and access patterns
- Pipeline stalls
- Floating point efficiency
- Instructions per cycle
- Subroutine resolution
- Process or thread attribution

**What’s PAPI?**

- **Middleware** that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.

- Started as a Parallel Tools Consortium project in 1998
  - Goal was to produce a specification for a **portable interface** to the hardware performance counters.

- Countable events are defined in two ways:
  - Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
  - Platform-dependent **Native Events** (e.g., L3_CACHE_MISS)

- Preset Events can be **derived** from multiple Native Events (e.g. PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform)
PAPI Hardware Events

Preset Events

- Standard set of over 100 events for application performance tuning
- No standardization of the exact definition
- Mapped to either single or linear combinations of native events on each platform
- Use `papi_avail` to see what preset events are available on a given platform

Native Events

- Any event countable by the CPU
- Same interface as for preset events
- Use `papi_native_avail` utility to see all available native events

Use `papi_event_chooser` utility to select a compatible set of events

Where’s PAPI?

PAPI runs on most modern processors and Operating Systems of interest to HPC:

- IBM POWER{4, 5, 5+, 6} / AIX or Linux
- PowerPC{-32, -64, 970} / Linux
- Cell
- Blue Gene / {L, P}
- Intel Pentium II, III, 4, M, Core, etc. / Linux
- Intel Itanium{1, 2, Montecito, Montvale}
- AMD Athlon, Opteron / Linux
- Cray XT{3, 4} Catamount, CNL
- Altix, Sparc, SiCortex…
- …and even Windows {XP, 2003 Server; PIII, Athlon, Opteron}!
- …but not Mac 😞
**PAPI COUNTER INTERFACES**

PAPI provides 3 interfaces to the underlying counter hardware:

1. A Low Level API manages hardware events (preset and native) in user defined groups called *EventSets*. Meant for experienced application programmers wanting fine-grained measurements.

2. A High Level API provides the ability to start, stop and read the counters for a specified list of events (preset only). Meant for programmers wanting simple event measurements.

3. Graphical and end-user tools provide facile data collection and visualization.

---

**PAPI PORTABLE LAYER**

**PAPI HARDWARE SPECIFIC LAYER**

- Kernel Extension
- Operating System
- Perf Counter Hardware

---

**PAPI HIGH LEVEL CALLS**

- **PAPI_num_counters()**
  - get the number of hardware counters available on the system

- **PAPI_flips (float *rtime, float *ptime, long long *flpins, float *mflips)**
  - simplified call to get Mflips/s (floating point instruction rate), real and processor time

- **PAPI_flops (float *rtime, float *ptime, long long *flpops, float *mflops)**
  - simplified call to get Mflops/s (floating point operation rate), real and processor time

- **PAPI_ipc (float *rtime, float *ptime, long long *ins, float *ipc)**
  - gets instructions per cycle, real and processor time

- **PAPI_accum_counters (long long *values, int array_len)**
  - add current counts to array and reset counters

- **PAPI_read_counters (long long *values, int array_len)**
  - copy current counts to array and reset counters

- **PAPI_start_counters (int *events, int array_len)**
  - start counting hardware events

- **PAPI_stop_counters (long long *values, int array_len)**
  - stop counters and return current counts
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];

/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);

/* Start the counters */
retval = PAPI_start (EventSet);
do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);

jump01lm: ~>papi_cost -h
This is the PAPI cost program.
It computes min / max / mean / std. deviation for PAPI start/stop
pairs and for PAPI reads. Usage:

cost [options] [parameters]
cost TESTS QUIET

Options:

-b BINS set the number of bins for the graphical
distribution of costs. Default: 100
-d show a graphical distribution of costs
-h print this help message
-s show number of iterations above the first 10 std
deviations
-t THRESHOLD set the threshold for the number of iterations.
Default: 100,000
Usage: papi_avail [options]

Options:

General command options:
-a, --avail Display only available preset events
-d, --detail Display detailed information about all preset events
-e EVENTNAME Display detail information about specified preset or native event
-h, --help Print this help message

This program provides information about PAPI preset and native events.

Available events and hardware information.

Vendor string and code : IBM (-1)
Model string and code    : POWER6 (16384)
CPU Revision             : 1081344.000000
CPU Megahertz            : 4704.000000
CPU Clock Megahertz      : 0
CPU's in this Node       : 16
Nodes in this System     : 1
Total CPU's              : 16
Number Hardware Counters : 6
Max Multiplex Counters   : 32

The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>0x80000005</td>
<td>Yes</td>
<td>No</td>
<td>Level 3 instruction cache misses</td>
</tr>
</tbody>
</table>

Of 103 possible events, 34 are available, of which 7 are derived.
avail.c PASSED
**PAPI UTILITIES: papi_avail**

```
jump01lm: ~/papi_avail -a
Available events and hardware information.
-------------------------------------------------------------------------------------------------
Vendor string and code   : IBM (-1)
Model string and code    : POWER6 (16384)
CPU Revision             : 1081344.000000
CPU Megahertz            : 4704.000000
CPU Clock Megahertz      : 0
CPU's in this Node       : 16
Nodes in this System     : 1
Total CPU's              : 16
Number Hardware Counters : 6
Max Multiplex Counters   : 32
-------------------------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>Yes</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAPI_FP_OPS</td>
<td>0x80000066</td>
<td>Yes</td>
<td>Floating point operations</td>
</tr>
</tbody>
</table>

Of 34 available events, 7 are derived.
```

```
jump01lm: ~/papi_avail -e PAPI_L1_DCM
Available events and hardware information.
---------------------------------------------------------------------------
[...]
Event name:                   PAPI_L1_DCM
Event Code:                   0x80000000
Number of Native Events:      2
Short Description:           |L1D cache misses|
Long Description:            |Level 1 data cache misses|
Developer's Notes:           ||
Derived Type:                |DERIVED_ADD|
Postfix Processing String:   ||
Native Code[0]: 0x400000c7  |PM_LD_MISS_L1|
Number of Register Values:   2
Register[ 0]: 0x00000000   |PowerPC64 event code|
Register[ 1]: 0xffffb9c4   |PowerPC64 counter_cmd code|
Native Event Description:    |Load references that miss the Level 1 Data cache. Combined unit 0 + 1.|
Native Code[1]: 0x40000134  |PM_ST_MISS_L1|
Number of Register Values:   2
Register[ 0]: 0x00000000   |PowerPC64 event code|
Register[ 1]: 0xffffb9c4   |PowerPC64 counter_cmd code|
Native Event Description:    |A store missed the dcache. Combined Unit 0 + 1.|
```
Available native events and hardware information.

Event Code   Symbol  | Long Description |
------------------------------------------------------------
0x40000000   PM_INST_FETCH  | No instructions were fetched this cycles (due to IFU hold, redirect, or icache miss) |
Groups:   55 188
0x40000001   PM_PLUS_PPC_CMPL  | A group containing at least one PPC instruction completed. For microcoded instructions that span multiple groups, this will only occur once. |
Groups:   54

0x40000227   PM_RUN_INST_CMPL  | Number of run instructions completed. |
Groups:   37  46

Total events reported: 552
native_avail.c                              PASSED
PAPI UTILITIES: papi_event_chooser

Jump01lm: \$ papi_event_chooser PRESET PAPI_L1_DCM

Event Chooser: Available events which can be added with given events.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_LDM</td>
<td>0x80000017</td>
<td>No</td>
<td>Level 1 load misses</td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_STM</td>
<td>0x80000018</td>
<td>No</td>
<td>Level 1 store misses</td>
<td></td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>0x80000032</td>
<td>No</td>
<td>Instructions completed</td>
<td></td>
</tr>
<tr>
<td>PAPI_LD_INS</td>
<td>0x80000035</td>
<td>No</td>
<td>Load instructions</td>
<td></td>
</tr>
<tr>
<td>PAPI_SR_INS</td>
<td>0x80000036</td>
<td>No</td>
<td>Store instructions</td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>0x80000040</td>
<td>Yes</td>
<td>Level 1 data cache accesses</td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_DCR</td>
<td>0x80000043</td>
<td>No</td>
<td>Level 1 data cache reads</td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_DCW</td>
<td>0x80000046</td>
<td>No</td>
<td>Level 1 data cache writes</td>
<td></td>
</tr>
</tbody>
</table>

Total events reported: 10

event_chooser.c                          PASSED

jump01lm: \$ papi_event_chooser PRESET PAPI_L1_DCM

PAPI_TOT_INS

Event Chooser: Available events which can be added with given events.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_LDM</td>
<td>0x80000017</td>
<td>No</td>
<td>Level 1 load misses</td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_STM</td>
<td>0x80000018</td>
<td>No</td>
<td>Level 1 store misses</td>
<td></td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>0x8000003b</td>
<td>No</td>
<td>Total cycles</td>
<td></td>
</tr>
</tbody>
</table>

Total events reported: 3

event_chooser.c                          PASSED
**PAPI UTILITIES: PAPI_COMMAND_LINE**

```
jump01lm: ~> papi_command_line PAPI_FP_OPS
Successfully added: PAPI_FP_OPS

PAPI_FP_OPS :  100000000

-----------------------------
Verification: None.
This utility lets you add events from the command line interface to see if they work.
command_line.c                           PASSED

jump01lm: ~> papi_command_line PAPI_FP_OPS PAPI_L1_DCA
Successfully added: PAPI_FP_OPS
Successfully added: PAPI_L1_DCA

PAPI_FP_OPS :  100000000
PAPI_L1_DCA :  120034404

-----------------------------
Verification: None.
This utility lets you add events from the command line interface to see if they work.
command_line.c                           PASSED
```

---

**Performance Measurement Categories**

- **Efficiency**
  - Instructions per cycle (IPC)
  - Memory bandwidth
- **Caches**
  - Data cache misses and miss ratio
  - Instruction cache misses and miss ratio
- **L2 cache misses and miss ratio**
- **Translation lookaside buffers (TLB)**
  - Data TLB misses and miss ratio
  - Instruction TLB misses and miss ratio
- **Control transfers**
  - Branch mispredictions
  - Near return mispredictions
#define ROWS 1000 // Number of rows in each matrix
#define COLUMNS 1000 // Number of columns in each matrix

void classic_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            float sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum +=
                matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }
}

void interchanged_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (k = 0; k < COLUMNS; k++) {
            for (j = 0; j < COLUMNS; j++) {
                matrix_c[i][j] +=
                matrix_a[i][k] * matrix_b[k][j];
            }
        }
    }
    // Note that the nesting of the innermost loops
    // has been changed. The index variables j and k
    // change the most frequently and the access
    // pattern through the operand matrices is
    // sequential using a small stride (one.) This
    // change improves access to memory data through
    // the data cache. Data translation lookaside
    // buffer (DTLB) behavior is also improved.
}

void classic_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            float sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum +=
                matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }
}

void interchanged_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (k = 0; k < COLUMNS; k++) {
            for (j = 0; j < COLUMNS; j++) {
                matrix_c[i][j] +=
                matrix_a[i][k] * matrix_b[k][j];
            }
        }
    }
    // Note that the nesting of the innermost loops
    // has been changed. The index variables j and k
    // change the most frequently and the access
    // pattern through the operand matrices is
    // sequential using a small stride (one.) This
    // change improves access to memory data through
    // the data cache. Data translation lookaside
    // buffer (DTLB) behavior is also improved.
}

• Measure instruction level parallelism
• An indicator of code efficiency

int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

realtime[0] = PAPI_get_real_usec();
retval = PAPI_start_counters(events, 2);
classic_matmul();
retval = PAPI_stop_counters(cvalues, 2);
realtime[1] = PAPI_get_real_usec();

int events[] = {PAPI_TOT_CYC, PAPI_TOT_INS};

retval = PAPI_library_init (PAPI_VER_CURRENT);
retval = PAPI_create_eventset(&EventSet);
retval = PAPI_add_events(EventSet, events, 2);
realtime[0] = PAPI_get_real_usec();
retval = PAPI_start(EventSet);
classic_matmul();
retval = PAPI_stop(EventSet, cvalues);
realtime[1] = PAPI_get_real_usec();
### IPC – Instructions Per Cycle

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High Level IPC Test (PAPI_{start,stop}_counters)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6106 sec</td>
<td>2.9762 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6939</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362605525</td>
<td>5318626915</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034503</td>
<td>9009011245</td>
</tr>
<tr>
<td>Low Level IPC Test (PAPI low level calls)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6113 sec</td>
<td>2.9772 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6933</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362750167</td>
<td>5320395138</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034381</td>
<td>9009011130</td>
</tr>
</tbody>
</table>

- Both PAPI methods are consistent
- Roughly 460% improvement in reordered code

### Data Cache Access

Cache miss: a failed attempt to read or write a piece of data in the cache
- Results in main memory access with much longer latency
- Important to keep data as close as possible to CPU

Data Cache Misses can be considered in 3 categories:

- **Compulsory misses**: Occurs on first reference to a data item
  - Prefetching can help

- **Capacity misses**: Occurs when the working set exceeds the cache capacity
  - **Spatial locality**: use all the data that is loaded into the cache
  - Smaller working set (blocking/tiling algorithms)

- **Conflict misses**: Occurs when a data item is referenced after the cache line containing the item was evicted earlier.
  - **Temporal locality**: reuse a word as long as possible
  - Data layout; memory access patterns
### L1 Data Cache Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PAPI NATIVE EVENTS:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA_CACHE_ACCESSES</td>
<td>2,002,807,841</td>
<td>3,008,528,961</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED</td>
<td>205,968,263</td>
<td>60,716,301</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED</td>
<td>61,970,925</td>
<td>1,950,282</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PAPI PRESET EVENTS:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>2,002,808,034</td>
<td>3,008,528,895</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>268,010,587</td>
<td>62,680,818</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Cache Request Rate</td>
<td>0.2224 req/inst</td>
<td>0.3339 req/inst</td>
</tr>
<tr>
<td>Data Cache Miss Rate</td>
<td>0.0298 miss/inst</td>
<td>0.0070 miss/inst</td>
</tr>
<tr>
<td>Data Cache Miss Ratio</td>
<td>0.1338 miss/req</td>
<td>0.0208 miss/req</td>
</tr>
</tbody>
</table>

- Two techniques
  - First uses native events
  - Second uses PAPI presets only
- ~50% more requests from reordered code
- 1/4 as many misses per instruction
- 1/6 as many misses per request

### Component PAPI (PAPI-C)

**Motivation:**
- Hardware counters aren't just for CPUs anymore
  - Network counters; thermal & power measurement...
- Often insightful to measure multiple counter domains at once

**Goals:**
- Support simultaneous access to on- and off-processor counters
- Isolate hardware dependent code in a separable component module
- Extend platform independent code to support multiple simultaneous components
- Add or modify API calls to support access to any of several components
- Modify build environment for easy selection and configuration of multiple available components
**COMPONENT PAPI**

![Diagram showing PAPI framework and components](image)

**PAPI FRAMEWORK**

Low Level User API

High Level User API

Developer API

PAPI COMPONENT (NETWORK)

Operating System

Counter Hardware

PAPI COMPONENT (CPU)

Operating System

Counter Hardware

PAPI COMPONENT (THERMAL)

Operating System

Counter Hardware

---

**SOME TOOLS THAT USE PAPI**

- **TAU** (U Oregon) [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)
- **PerfSuite** (NCSA) [http://persuite.ncsa.uiuc.edu/](http://persuite.ncsa.uiuc.edu/)
- **HPCToolkit** (Rice Univ) [http://hipersoft.cs.rice.edu/hpctoolkit/](http://hipersoft.cs.rice.edu/hpctoolkit/)
- **KOJAK** and **SCALASCA** (FZ Juelich, UTK) [http://icl.cs.utk.edu/kojak/](http://icl.cs.utk.edu/kojak/)
- **VampirTrace** and **Vampir** (TU Dresden) [http://www.vamir.eu](http://www.vamir.eu)
- **SvPablo** (UNC Renaissance Computing Institute) [http://www.renci.unc.edu/Software/Pablo/pablo.htm](http://www.renci.unc.edu/Software/Pablo/pablo.htm)
- **ompP** (UTK) [http://www.ompp-tool.com](http://www.ompp-tool.com)
1. Motivation
   • What is Performance?
   • Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   • The performance analysis cycle
   • Measurement: profiling vs. tracing
   • Analysis: manual vs. automated

3. Performance Analysis Tools
   • PAPI: Access to hardware performance counters
   • Vampir Suite: Instrumentation and Trace visualization
   • KOJAK / Scalasca: automatic performance analysis tool
   • TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications

Vampir: Performance Analysis Suite

Consists of:
• VampirTrace part for instrumentation, monitoring and recording
• VampirServer part for visualization and analysis

VampirTrace
• Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

VampirServer
• Implements client / server model with distributed server
• Allows very scalable interactive visualization for traces
Vampir: Performance Analysis Suite

Consists of:
- VampirTrace part for instrumentation, monitoring and recording
- VampirServer part for visualization and analysis

VampirTrace
- Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

VampirServer
- Implements client / server model with distributed server
- Allows very scalable interactive visualization for traces

Instrumentation in General

- Instrumentation: Process of modifying programs to detect and report events by calling *instrumentation* functions.

- instrumentation functions provided by trace library

- call == notification about run-time event

- there are various ways of instrumentation:
  - Source Code instrumentation (manual or automatic)
  - Wrapper function instrumentation (e.g. MPI profiling interface)
  - Compiler instrumentation (e.g. gcc -finstrument-functions -c foo.c) supported by many compilers (GCC, Intel, IBM, PGI, NEC, Hitachi, Sun Fortran, …)
  - Dynamic instrumentation (modify binary executable in memory) very platform/machine dependent, expensive (see www.dyninst.org)
**Trace Libraries**

- provide instrumentation functions
- receive events of various types
- collect event properties
  - time stamp
  - location (thread, process, cluster node, MPI rank)
  - event specific properties
    - perhaps hardware performance counter values
- record to memory buffer, flush eventually
- try to be fast, minimize overhead

**Trace Files and Formats**

- TAU Trace Format (University of Oregon)
- Epilog (ZAM, FZ Jülich, Germany)
- STF (Pallas, now Intel)
- Open Trace Format (OTF)
  - ZIH, TU Dresden, Germany in cooperation with Oregon & Jülich
  - single/multiple files per trace with
  - fast sequential and random access
  - including API for writing/reading
  - supports auxiliary information
  - see [http://www.tu-dresden.de/zh/otf/](http://www.tu-dresden.de/zh/otf/)
**Instrumentation with VampirTrace**

**Edit – Compile – Run Cycle**

- Source Code → Compiler → Binary → Run → Results

**Edit – Compile – Run Cycle with VampirTrace**

- Source Code → Compiler → VT Wrapper → Binary → Run → Results

**Instrumentation Types**

- Automatic instrumentation using compiler wrappers
- Manual instrumentation
- Binary instrumentation
**Compiler Wrappers**

- Easiest way of using VampirTrace
- No source code modifications
- In the build system of your application: Swap calls to the regular compiler with calls to the VampirTrace compiler wrappers
  - For compiling and linking
  - *e.g.* in the `makefile` change:

  ```
  CC=mpicc
  CC=vtcc
  ```

- Rebuild the application
- Run the application to produce trace data

Captured events:
- All user function entries and exits
  - If supported by the compiler (Intel, GNU, PGI, NEC, IBM)
- MPI calls and messages
  - If the application is MPI parallel
- OMP regions
  - If the application is OpenMP parallel

```c
icc  hello.c -o hello
vtcc hello.c -o hello
icpc hello_parallel.cpp -Impi -o hello_parallel
vtcxx hello_parallel.cpp -Impi -o hello_parallel
```
**Manual Instrumentation**

- Allows detailed source code instrumentation, e.g. regions of functions such as loops.
- Can – and should – be combined with automatic instrumentation.
- Be sure to instrument all function exits!

- Add the following into our source code to instrument a region, e.g. C: (available for C++ and FORTRAN as well)

```c
#include "vt_user.h"
...
VT_USER_START("Region_1");
...
VT_USER_END("Region_1");
...
```

- Compile with "-DVTRACE"
  - otherwise, VampirTrace macros will expand to empty blocks, producing zero overhead

```
vtcc -vt:inst manual prog.c -DVTRACE -o prog
```

---

**Runtime Measurement Settings**

- Environment Variables
- PAPI hardware performance counters
- Memory allocation counters
- Application I/O calls
- Filtering
- Grouping
**Environment Variables**

- By default, trace data is written to the `pwd`
- About everything of this can be customized with environment variables
- Environment variables must be set prior to running the application, not prior to building the application

- `VT_PFORM_GDIR` Directory where final trace file is stored
- `VT_PFORM_LDIR` Directory for intermediate trace files
- `VT_FILE_PREFIX` Trace file name
- `VT_BUFFER_SIZE` Internal trace buffer size
- `VT_MAX_FLUSHES` Max number of buffer flushes
- `VT_MEMTRACE` Enable memory allocation tracing
- `VT_IOTRACE` Enable I/O tracing
- `VT_MPTTRACE` Enable MPI tracing
- `VT_FILTER_SPEC` Name of filter file
- `VT_GROUPS_SPEC` Name of function groups file
- `VT_COMPRESSION` Compress trace files
- `VT_METRICS` List of PAPI counters

**PAPI Counters in Traces**

- PAPI counters can be included in traces
  - If VampirTrace was build with PAPI support
  - If PAPI is available on the platform

- `VT_METRICS` can be used to specify a colon-separated list of PAPI counters

```sh
export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```
FILTERING

- Filtering is one of the ways to reduce trace file size
- Environment variable `VT_FILTER_SPEC`
  ```
  export VT_FILTER_SPEC=/home/user/filter.spec
  ```
- Filter file contains a list of filters for functions that are applied during the execution of the application
  ```
  my*;test -- 1000
  calculate -- -1    # -1: unlimited
  * -- 1000000
  ```
- The `vtfilter` tool
  - can create a filter file
  - can reduce the size of trace files

GROUPING

- Groups can be defined by the user to group related functions
  - Groups can be assigned different colors in Vampir and VampirServer, highlighting application behavior
- Environment variable `VT_GROUPS_SPEC`
  ```
  export VT_GROUPS_SPEC=/home/<user>/groups.spec
  ```
- Groups file contains a list of groups with associated functions
  ```
  CALC=calculate
  MISC=my*;test
  UNKNOWN=*
**Vampir: Performance Analysis Suite**

Consists of:
- VampirTrace part for instrumentation, monitoring and recording
- VampirServer part for visualization and analysis

**VampirTrace**
- Supports a variety of performance aspects: e.g. MPI comm events, subroutine calls from user code, HW perf counter, I/O events, etc.

**VampirServer**
- Implements client / server model with distributed server
- Allows very scalable interactive visualization for traces

---

**Trace Visualization**

- Vampir Architecture
- Various Displays
  - Global Timeline, Process Timeline + Counter Display
  - Summary Chart (aka. Profile)
  - Summary Timeline
  - Message Statistics
  - Collective Communication Statistics
  - Counter Timeline
  - I/O Event Display
  - Call Tree
**Vampir Architecture 1/2**

- Parallel Program
  - Monitor System
- File System
  - Trace 1
  - Trace 2
  - Trace 3
  - Trace N
- Analysis
  - Worker 1
  - Worker 2
  - Worker m
- Parallel I/O
- Event Streams
- Message Passing
- Master
- Visualization Client
  - Timeline with 16 Traces visible
  - Segment Indicator
  - 768 Processes Thumbnail View

**Vampir Architecture 2/2**

- parallel/distributed server
  - runs in (part of) production environment
  - no need to transfer huge traces
  - parallel I/O
- lightweight client on local workstation
  - receive visual content only
  - already adapted to display resolution
  - moderate network load
- scales to traces > 40 GB
**Timeline Display (Zoomed)**

1. Shows all selected processes
2. Shows state changes (activity color)
3. Shows messages, collective and MPI–IO operations

**Timeline Display – Message Details**

- Click on message line
- Message information
- Message send op
- Message receive op

4/22/09
**Message statistics** for each process/node pair:
- Byte and message count
- min/max/avg message length, bandwidth

**COLLECTIVE OPERATIONS**

- For each process: mark operation locally
  - Start of op
  - Data being sent
  - Stop of op
  - Data being received
- Connect start/stop points by lines
- E.g. MPI_Gather: gathers together values from a group of processes
**Trace Visualization in General**

**Trace Visualization**
- Vampir provides a number of display types
- each allows many different options

**Advice**
- make a hypothesis about performance problems
- consider application's internal workings if known
- select the appropriate display
- use statistic displays in conjunction with timelines

**Finding Performance Bottlenecks**

**Where?**
- Computation
- Communication
- Memory, I/O, …
- Tracing itself

**What?**
- unbalanced computation (e.g. single late comer)
- strictly serial parts of program (e.g. idle processes/threads)
- very frequent tiny function calls
- sparse loops
Example: Idle OpenMP Threads

- communication as such (domination over computation)
- late sender, late receiver
- point-to-point messages instead of collective communication
- unmatched messages
- overcharge of MPI's buffers
- bursts of large messages (bandwidth)
- frequent short messages (latency)
- unnecessary synchronization (barrier)
Example: Dominating Communication

- **Summary Timeline**: # of processes, actively involved in activity at certain point in time (vertical histogram)

Finding Performance Bottlenecks (cont.)

- memory bound computation
  - inefficient L1/L2/L3 cache usage
  - TLB misses
  - detectable via HW performance counters
- I/O bound computation
  - slow input/output
  - sequential I/O on single process
  - I/O load imbalance
- exception handling
**Example: Low Floating Point Rate Due to Heavy FP Exceptions**

PAPI counter:

![Graph showing PAPI counter data](image)

**Finding Performance Bottlenecks (cont.)**

- Measurement overhead
  - especially grave for tiny function calls
  - solve with selective instrumentation; filtering
- Long, asynchronous trace buffer flushes
- Too many concurrent counters
- Heisenbugs
  - software bug that disappears or alters its characteristics when an attempt is made to study it
  - considered exceptionally difficult to understand and repair
Example: Trace Buffer Flush

- Default: VT_BUFFER_SIZE = 32 MB ; VT_MAX_FLUSHES = 1
- Events that are to be recorded after the limit has been reached are no longer written into the trace file
- Remove limit to get complete trace of an application: VT_MAX_FLUSHES = 0
  Then VampirTrace always writes the buffer to disk when it is full
- Increasing size of VT buffer will decrease memory available to application
  parts of app. swapped to disk → leads to significant change in behavior of app.

Outline

1. Motivation
   - What is Performance?
   - Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

3. Performance Analysis Tools
   - PAPI: Access to hardware performance counters
   - Vampir Suite: Instrumentation and Trace visualization
   - KOJAK / Scalasca: automatic performance analysis tool
   - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
**Basic Idea**

- **For non-standard / tricky cases (10%)**
- **For expert users**

- **For standard cases (90% ?!)**
- **For “normal” users**
- **Starting point for experts**

Automatic searching of event traces of parallel applications for execution patterns that indicate inefficient behavior!

---

**MPI-1 Pattern: Wait at Barrier**

- Time spent in front of MPI synchronizing operation such as barriers
- This pattern covers the time spent waiting in front of an MPI barrier, which is the time inside the barrier call until the last process has reached the barrier. A large amount of waiting time spent in front of barriers can be an indication of load imbalance.
**MPI-1 Pattern: Late Sender / Receiver**

- **Late Sender:** Time lost waiting in a blocking receive operation posted earlier than the corresponding send operation.

- **Late Receiver:** Time lost waiting in a blocking send operation until the corresponding receive operation is called.

---

**Performance Property**

**What problem?**

**Region Tree**

**Where in source code?**

**In what context?**

**How is the problem distributed across the machine?**

**Color Coding**

**How severe is the problem?**
KOJAK: sPPM run on (8x16x14) 1792 PEs

- New topology display
- Shows distribution of pattern over HW topology
- Easily scales to even larger systems

Outline

1. Motivation
   - What is Performance?
   - Why being annoyed with Performance Analysis?

2. Concepts and Definitions
   - The performance analysis cycle
   - Measurement: profiling vs. tracing
   - Analysis: manual vs. automated

3. Performance Analysis Tools
   - PAPI: Access to hardware performance counters
   - Vampir Suite: Instrumentation and Trace visualization
   - KOJAK / Scalasca: automatic performance analysis tool
   - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
TAU Parallel Performance System

- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation

- Flexible and configurable performance measurement

- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers

- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid

- Integration in complex software, systems, applications

Using TAU: A Brief Introduction

- To instrument source code use TAU stub Makefiles:
  
  ```
  % setenv TAU_MAKEFILE /usr/pkgs/tau3/lib/Makefile.tau-mpi-pdt-pgi
  ```

  And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:

  ```
  % mpif90 foo.f90
  ```

  changes to

  ```
  % tau_f90.sh foo.f90
  ```

- Execute application and then run:

  ```
  % pprof  (flat profile for text based profile display)
  % paraprof (for GUI)
  ```
TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events (“user-defined timers”)
  - Atomic events (e.g., size of memory allocated/freed)
  - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization (eliminate instrumentation in lightweight routines)

Flexible instrumentation mechanisms at multiple levels:

- **Source code**
  - manual (TAU API, TAU Component API)
  - automatic
    - C, C++, F77/90/95 (Program Database Toolkit \(\text{PDT}\))
    - OpenMP (directive rewriting \(\text{Opari}\), \(\text{POMP spec}\))
- **Object code**
  - pre-instrumented libraries (e.g., MPI using \(\text{PMPI}\))
  - statically-linked and dynamically-linked
- **Executable code**
  - dynamic instrumentation (pre-execution) \(\text{DynInstAPI}\)
  - virtual machine instrumentation (e.g., Java using \(\text{JVMPI}\))
  - Python interpreter based instrumentation at runtime
**TAU Measurement Approach**

- Portable and scalable parallel profiling solution
  - Multiple profiling types and options
  - Event selection and control (enabling/disabling, throttling)
  - Online profile access and sampling
  - Online performance profile overhead compensation
- Portable and scalable parallel tracing solution
  - Trace translation to Open Trace Format (OTF)
  - Trace streams and hierarchical trace merging
- Robust timing and hardware performance support
- Multiple counters (hardware, user-defined, system)
- Performance measurement for CCA component software

---

**ParaProf Main Window**

Click left mouse button

Click right mouse button

Analyze workload characteristics
3D PERFORMANCE DATA EXPLORATION WITH TAU

Paraprof viewer
(from the TAU toolset)

ParaProf – 3D Scatterplot (Miranda)

- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library
  - JOGL

32k processors
PerfExplorer - Cluster Analysis

- Four significant events automatically selected (from 16K processors)
- Clusters and correlations are visible

ParaProf - Callgrap

- A simple one ....
Describes strength and direction of a linear relationship between two variables (events) in the data.
-0.995 indicates strong, negative relationship

As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases

**DOCUMENTATION, MANUALS, USER GUIDES**

- **PAPI:**
  PAPI documentation page available from the PAPI website:
  http://icl.cs.utk.edu/papi/

- **VAMPIR:**
  - VampirTrace download and documentation:
    http://www.tu-dresden.de/zih/vampirtrace
  - Vampir Suite
    http://www.vampir-ng.org/

- **Scalasca:**
  http://www.scalasca.org

- **TAU:**
  TAU Users Guide and papers available from the TAU website:
  http://www.cs.uoregon.edu/research/tau/
**Summary: Steps of Performance Evaluation**

- Collect basic routine-level timing profile to determine where most time is being spent.

- Collect routine-level hardware counter data to determine types of performance problems.

- Collect callpath profiles to determine sequence of events causing performance problems.

- Conduct finer-grained tracing to pinpoint performance bottlenecks.
  - E.g. Loop-level profiling with hardware counters.
  - E.g. Tracing of communication operations.

---

**Conclusion**

- Performance analysis very important in HPC.

- Use performance analysis tools for profiling and tracing.

- Do not spend effort in *Do-It-Yourself* solutions, e.g. like printf-debugging.

- Use tracing tools with some precautions.
  - Overhead.
  - Data volume.
CS 594: Scientific Computing for Engineers

Topic: Performance Analysis Tools

Heike Jagode
jagode@eecs.utk.edu

With slides from Karl Fuerlinger, Andreas Knuepfer, Shirley Moore, Sameer Shende, Felix Wolf and others.