Performance Measurement and Analysis of Parallel Programs

Bernd Mohr
Forschungszentrum Jülich (FZJ)
John von Neumann Institute for Computing (NIC)
Central Institute for Applied Mathematics (ZAM)
52425 Jülich, Germany
b.mohr@fz-juelich.de

Performance Properties of Parallel Programs

- Factors which influence performance of parallel programs
  - "Sequential" factors
    - Computation
      - Choose right algorithm, use optimizing compiler
    - Cache and memory
      - Tough! Not many tools yet, hope compiler gets it right
    - Input / output
      - Not given enough attention
  - "Parallel" factors
    - Communication (Message passing)
    - Threading
    - Synchronization
      - More or less understood, tool support
Performance Measurement Cycle

- Instrumentation
  - Insertion of extra code (probes, hooks) into application
- Measurement
  - Collection of data relevant to performance analysis
- Analysis
  - Calculation of metrics, identification of performance problems
- Presentation
  - Transformation of the results into a representation that can be easily understood by a human user
- Optimization
  - Elimination of performance problems

Outline

- Metrics
  - Instrumentation techniques
    - Source code instrumentation
    - Binary instrumentation
  - Instrumentation of parallel programs
    - MPI
    - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
- Simple timers and hardware counter measurements
Metrics of Performance

- What can be measured?
  - A count of how many times an event occurs
    - E.g., Number of input / output requests
  - The duration of some time interval
    - E.g., duration of these requests
  - The size of some parameter
    - Number of bytes transmitted or stored

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

Example Metrics

- Clock rate
- MIPS
  - Millions of instructions executed per second
- MFLOPS
  - Millions of floating-point operations per second
- Benchmarks
  - Standard test program(s)
  - Standardized methodology
  - E.g., SPEC, Linpack
- QUIPS / HINT [Gustafson and Snell, 95]
  - Quality improvements per second
  - Quality of solution instead of effort to reach it
- Execution time

"math" Operations?
HW Operations?
HW Instructions?
32-/64-bit? ...
**Execution Time**

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

- **CPU time**
  - Time spent by the CPU to execute the program
  - Execution time on behalf of the program
  - Does not include time the program was context-switched out
    - Problem: does not include inherent waiting time (e.g., IO)
    - Problem: portability? What is user, what is system time?

- Problem: execution time is non-deterministic
  - Use mean or minimum of several runs

**Other Metrics**

- **Response time**
  - Time from request submission to return of results

- **Throughput**
  - Number of jobs or operations completed per time unit

- **Bandwidth**
  - Bytes transferred by elapsed time

- **Utilization**

- **Speedup**
  - Factor by which an application/system runs faster
  - Parallel speedup
    \[ s(n) = \frac{t_{\text{sequential}}}{t_{\text{parallel}}(n)} \]
  - Parallel efficiency
    \[ e(n) = \frac{s(n)}{n} \]
Outline

- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
- Simple timers and hardware counter measurements

Semantic Performance Mapping

- Instrumentation levels
  - Source code
  - Library
  - Runtime system
  - Object code
  - Operating system
  - Runtime image
  - Virtual machine

- Problem
  - Every level provides different information
  - Often instrumentation on multiple levels required

- Challenge
  - Mapping performance data onto application-level abstraction
Instrumentation Techniques

- Static instrumentation
  - Program is instrumented prior to execution
- Dynamic instrumentation
  - Program is instrumented at runtime

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

  ⇒ e.g., "printf" ⇒ manual static source-code instrumentation

Source Code Instrumentation (I)

- For large complex applications, manual instrumentation is too tedious and error-prone ⇒ Tool support needed
- Automatic performance Instrumentation typically requires full source code parsers, e.g.,
  - Fortran, C: find 1st executable line and all exit points
  - C: executable code inside return statements

```
int func(...) {
    double d;
    return (foo() * bar());
}
```

```
int func(...) {
    double d;
    trace_enter();
    { int t1_ = (foo() * bar());
    trace_exit();
    return t1_; }
}
```
Source Code Instrumentation (II)

- Example C++ issues:
  - Template instrumentation?
  - Executing code before main
- C++ instrumentation trick
  - Define instrumentation object
    ```cpp
class Tracer { public:
    Tracer(...) { trace_enter(); }
    ~Tracer() { trace_exit(); }
};
```
  - Declare instrumentation object as 1st statement in every function and method to be instrumented
    ```cpp
    int func(...) { Tracer trc_1;
    double d;
    return (foo()*bar());
    }
    ```
- Function overloading
- Operator overloading

Binary Instrumentation

- Static binary rewrite
  - Instrumentation code is inserted into the binary before it starts to execute
  - Creates modified executable

- Dynamic binary instrumentation
  - On-the-fly: Insert, remove, and change instrumentation in the application program while it is running
  - Most flexible (but most complex) technique
  - Parallel programs
    - Coordinated instrumentation of all images needed
### Dyninst

- Dyninst is a C++ library for machine-independent
  - process control and manipulation
  - runtime code generation
  - and binary patching

- University of Wisconsin and University of Maryland
- Basis for Paradyn and DPCL
- Open source

- Supports
  - Mips (IRIX)
  - Alpha (Tru64)
  - Power/PowerPC (AIX)
  - IA64 (Linux)
  - Sparc (Solaris)
  - IA32 (Linux and NT)

- [http://www.dyninst.org](http://www.dyninst.org)

### Dynamic Probe Class Library

- C++ Based Class Library
  - IBM Poughkeepsie Unix Development Lab
  - 11 Classes, Plus Additional API’s

- DPCL provides a general purpose infrastructure that flexibly supports the generation of arbitrary instrumentation for:
  - Serial
  - Shared memory
  - Message passing

- Based on Paradyn and Dyninst
- AIX (+ Linux)

- Now: open source
  - [http://dpcl.sourceforge.net](http://dpcl.sourceforge.net)
Comparison of Techniques (I)

- Source code instrumentation
  - 🙆 Portable
  - 🙆 Link back to source code easy
  - 😞 Only way to capture "high-level" user abstractions

  - 🙆 Recompilation necessary for (change in) instrumentation
  - 😞 Requires source code to be available
  - 😞 Hard to use for mixed-language applications
  - 😞 Source-to-source translation tool hard to implement for C++ and Fortran90

Comparison of Techniques (II)

- Binary code instrumentation
  - 😥 / 😃 The other way round compared to source instrumentation

- Pre-instrumented library / runtime
  - 😃 Easy to use: only re-linking necessary
  - 😞 Can only record information about library / runtime entities

- No single technique is sufficient!
- Typically, combinations of techniques needed!
Outline

- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
- Simple timers and hardware counter measurements

Instrumentation of Parallel Programs

- User-level constructs
  - Functions
  - Loops
  - ...

- Constructs of the parallel programming models
  - Message passing
    - MPI, PVM, ...
  - Threading and synchronization
    - OpenMP, POSIX, Win32, or Java threads, ...
**Instrumentation of User Functions**

- Ideally: instrumentation by compiler or tool
  - Hidden, unsupported compiler options
    - (GNU, Intel, IBM, NEC, Sun Fortran, PGI, Hitachi, ???)
  - TAU Source Code Instrumentor
  - TAU Binary Instrumentor (Dyninst)
  - TAU Virtual Machine Instrumentor (Java, Python)

- Always works: manually
  - TAU Instrumentation API
  - KOJAK’s POMP Directives
  - More details later …

- Main problem: selection of relevant constructs

---

**PMPI: The MPI Profiling Interface**

- Every MPI function has two names: `MPI_xxx` and `PMPI_xxx`
- This allows selective replacement of MPI routines at link time → no re-compilation necessary
- Also called: wrapper function library
- Used by basically every MPI performance tools
  - Intel Trace Collector, MPICH MPE, KOJAK, TAU, …

![Diagram](diagram.png)
**PMPI Example (C/C++)**

```c
#include <stdio.h>
#include "mpi.h"

static int numsend = 0;

int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm) {
    numsend++;
    return PMPI_Send(buf, count, type, dest, tag, comm);
}

int MPI_Finalize() {
    int me;
    PMPI_Comm_rank(MPI_COMM_WORLD, &me);
    printf("%d sent %d messages.\n", me, numsend);
    return PMPI_Finalize();
}
```

**PMPI Wrapper Development**

- MPI has many functions! [MPI-1: 130  MPI-2: 320]
  - use wrapper generator (e.g., from MPICH MPE)
  - needed for Fortran and C/C++

- Message analysis / recording
  - Location recording  ⇒ use ranks in MPI_COMM_WORLD?
  - Data volume  ⇒ #elements * sizeof(type)
  - No message ID  ⇒ need complete recording of traffic
  - Wildcard source and tag  ⇒ record real values
  - Collective communication  ⇒ communicator tracking
  - Non-blocking, persistent communication  ⇒ track requests
  - Non-blocking  ⇒ record recv at Wait*, Test*, Irecv ?
  - One-sided communication?
**OpenMP Monitoring?**

**Problem:**
- OpenMP does not define standard monitoring interface
- OpenMP is defined mainly by directives/pragmas

**Solution:**
- POMP: OpenMP Monitoring Interface
- Joint Development
  - Forschungszentrum Jülich
  - University of Oregon
- Presented at EWOMP'01, LACSI'01 and SC01


---

**Example:**

```fortran
!$OMP PARALLEL DO POMP Instrumentation

call pomp_parallel_fork(d1)
!$OMP PARALLEL other-clauses...
call pomp_parallel_begin(d1)
call pomp_do_enter(d2)
!$OMP DO schedule-clauses, ordered-clauses,
   lastprivate-clauses
do loop
!$OMP END DO NOWAIT
call pomp_barrier_enter(d3)
!$OMP BARRIER
call pomp_barrier_exit(d3)
call pomp_do_exit(d2)
call pomp_parallel_end(d1)
!$OMP END PARALLEL DO
call pomp_parallel_join(d1)
```

© 2006 Bernd Mohr, Forschungszentrum Jülich
Prototype POMP Instrumentation Tool

- OpenMP Pragma And Region Instrumentor
- Source-to-source translator to insert POMP calls around OpenMP constructs and API functions
- Implemented in C++

- Supports:
  - Fortran77 und Fortran90, OpenMP 2.0
  - C und C++, OpenMP 1.0
  - Additional POMP directives for control and region definition
  - KOJAK and TAU POMP measurement libraries
  - Preserves source code information (#line line file)
- Does not support: Instrumentation of user functions

http://www.fz-juelich.de/zam/kojak/opari/

Outline

- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
- Simple timers and hardware counter measurements
### Performance Measurement

- **Two dimensions**
  - When performance measurement is triggered
    - External agent (asynchronous)
      - Sampling
        - Timer interrupt
        - Hardware counters overflow
        - Can measure unmodified executables, very low overhead
    - Internal agent (synchronous)
      - Code instrumentation:
        - Automatic or manual instrumentation
  - How performance data is recorded
    - Profile ::= Summation of events over time
    - run time summarization (functions, call sites, loops, ...)
    - Trace file ::= Sequence of events over time

### Measurement

- Typical performance data include
  - Counts
  - Durations
    - **inclusive** duration
    - **exclusive** duration
  - Communication cost
  - Synchronization cost
  - IO accesses
  - System calls
  - Hardware events

```c
int foo()
{
    int a;
a = a + 1;
bar();
a = a + 1;
return a;
}
```
Critical issues

- Accuracy
  - Perturbation
    - Measurement alters program behavior
    - E.g., memory access pattern
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Accuracy of timers, counters
- Granularity
  - How many measurements
  - How much information / work during each measurement
- Tradeoff
  - Accuracy ⇔ expressiveness of data

Profiling

- Recording of aggregated information
  - Time
  - Counts
    - Calls
      - Hardware counters
  - about program and system entities
    - Functions, loops, basic blocks, ...
    - Processes, threads
- Methods to create a profile
  - PC sampling (statistical approach)
  - Interval timer / direct measurement (deterministic approach)
Profiling (2)

- Sampling
  - General statistical measurement technique based on the assumption that a subset of a population being examined is representative for the whole population
  - Periodic operating system signal interrupts the running program
  - Interrupt service routine examines return-address stack to find address of instruction being executed when interrupt occurred
  - Using symbol-table information this address is mapped onto specific subroutine
  - Requires long-running programs

- Interval timing
  - Time measurement at the beginning and at the end of a code region
  - Requires instrumentation + high-resolution / low-overhead clock

Measurement Methods: Tracing

- Recording information about significant points (events) during execution of the program
  - Enter/leave a code region (function, loop, ...)
  - Send/receive a message ...

- Save information in event record
  - Timestamp, location ID, event type
  - plus event specific information

- Event trace := stream of event records sorted by time

- Can be used to reconstruct the dynamic behavior
  - Abstract execution model on level of defined events
Tracing: Instrumentation, Monitoring, Trace

Process A:

```c
void master {
    ... trace(ENTER, 1);
    send(B, tag, buf);
    ... trace(EXIT, 1);
}
```

Process B:

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

Tracing: Time-line visualization

```
1 master
2 slave
3 ...
```

```
<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th>slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>ENTER 1</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>ENTER 2</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>SEND B</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>EXIT 1</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>RECV A</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>EXIT 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

© 2006 Bernd Mohr, Forschungszentrum Jülich
### Tracing vs. Profiling

- **Tracing Advantages**
  - Event traces preserve the temporal and spatial relationships among individual events (context!
  - Allows reconstruction of dynamic behavior of application on any required abstraction level
    - Automatic analysis
    - Visualization
  - Most general measurement technique
    - Profile data can be constructed from event traces
- **Disadvantages**
  - Traces can become very large
  - Writing events to a file at runtime can cause perturbation
  - Writing tracing software is complicated
    - Event buffering, clock synchronization, ...

### Trace File Formats: VTF3 + OTF

- Used by Vampir trace analyzer
- Supported event types
  - Region enter and exit
  - Collective region enter and exit
  - Message send and receive
- Stores source code and HW counter information
- Current Vampir trace formats
  - VTF: family of historical ASCII and binary formats
  - OTF: new Open Trace Format
- VTF3 reading/writing API available at http://www.cs.uoregon.edu/research/paracompp/tau/vtf3-1.43.tar.gz
- OTF beta available to developers and researchers
Trace File Formats: TAU

- Used by TAU performance analysis toolset
- Supported event types
  - Region enter and exit
  - Message send and receive
- Stores HW counter information
- TAU format converter for all open formats available
  - VTF3 + OTF
  - ALOG + SLOG2
  - EPILOG
  - Paraver

Trace File Formats: EPILOG

- Event Processing, Investigation, and LOGging
- Open-source trace format of KOJAK project
- MPI and OpenMP support (i.e., thread-safe)
- Current event types
  - Region enter and exit
  - Collective region enter and exit
  - Message send and receive
  - Parallel region fork and join
  - Lock acquire and release
  - RMA get and put
- Stores source code and HW counter information
- Hierarchical location ID (machine, node, process, thread)
- EPILOG ⇔ VTF3 + OTF + Paraver converters available
Trace File Formats: MPICH MPE

- MPICH Multi-Processing Environment
  - Unfortunately, uses the term "logging" (= tracing)
- Supports only
  - Region enter and exit
  - Message send and receive events
- Long history of trace formats, see
  - and associated visualization tools, see
- Current formats and corresponding viewers
  - CLOG \rightarrow Jumpshot-2
  - SLOG \rightarrow Jumpshot-3
  - SLOG-2 \rightarrow Jumpshot-4

Outline

- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
- Simple timers and hardware counter measurements
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
  □ clock()
  □ times()
  □ ...

- Hardware Performance Counter Measurements
  □ PAPI
  □ PCL
  □ ...

---

Timer: clock()

- ANSI C
- Returns an approximation of processor time used by the program
- To get value in seconds divide by CLOCKS_PER_SEC

```c
#include <time.h>

clock_t starttime, endtime;
double elapsed; /* seconds */

starttime = clock(); /* -- long running code -- */
endtime = clock();

elapsed = (endtime - starttime)
            / (double) CLOCKS_PER_SEC;
```

© 2006 Bernd Mohr, Forschungszentrum Jülich
**Timer: times()**

- UNIX function
- Returns wall-clock time; system and user time in output parameter

```c
#include <sys/times.h>
#include <unistd.h>

struct tms s, e;
clock_t startime, endtime;
double wtime, utime, stime; /* seconds */
long clk_tck = sysconf(_SC_CLK_TCK);

startime = times(&s);
/* -- long running code -- */
endtime = times(&e);

utime = (e.tms_utime - s.tms_utime) / (double) clk_tck;
stime = (e.tms_stime - s.tms_stime) / (double) clk_tck;
wtime = (endtime - startime) / (double) clk_tck;
```

**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 */

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

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

```c
#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
  - Not required to be consistent/synchronized across ranks!
- Same for OpenMP 2.0 (!) programming

```c
#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, HPMLIB, libpfm, libperf, ...)
  - PAPI
  - PCL

---

**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
  - The Performance Counter Library - PCL
    - Research Centre Juelich, 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 does not define 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
  - Provides per-process 64-bit memory-mapped virtual counters
  - Provides per-process virtual Time Stamp Counter (TSC)
  - Needs kernel patching!
    - http://user.it.uu.se/~mikpe/linux/perfctr/

- Perfmon by Stephane Eranian from HP - IA64
  - Provides functions to control monitoring from user level
  - Handles EAR events

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)
  - pat_hwpc (Cray)
  - pfmon from HP (part of Perfmon for IA64)
  - psrun (NCSA)
  - cputrack, har (Sun)
  - perfex, ssrun (SGI)
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
- http://icl.cs.utk.edu/papi/

PAPI Architecture

- Tools
- Portable Layer
  - PAPI Low Level
  - PAPI High Level
- Machine Specific Layer
  - PAPI Machine Dependent Substrate
  - Kernel Extensions
  - Operating System
  - Hardware Performance Counters

© 2006 Bernd Mohr, Forschungszentrum Jülich
Software Tools to Support Programming and Optimization on HPC Systems

ISC Tutorial
June 2006

© 2006 Bernd Mohr, Forschungszentrum Jülich

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 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 native_avail utility to determine which predefined events are available on a given platform

PAPI Preset Listing

Test case 8: Available events and hardware information

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCM</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>No</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>0x80000003</td>
<td>No</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>No</td>
<td>No</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>0x80000005</td>
<td>No</td>
<td>No</td>
<td>Level 3 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_CA_SNP</td>
<td>0x80000006</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 cache misses</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>0x80000007</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>0x80000008</td>
<td>No</td>
<td>No</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>0x80000009</td>
<td>No</td>
<td>No</td>
<td>Requests for a snoop</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>0x80000010</td>
<td>No</td>
<td>No</td>
<td>Requests for shared cache line</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>0x80000011</td>
<td>No</td>
<td>No</td>
<td>Requests for clean cache line</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>0x80000012</td>
<td>No</td>
<td>No</td>
<td>Requests for cache line inv</td>
</tr>
</tbody>
</table>
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    realtime, cputime, mflops
  integer   ierr
  real*4    a(N,N)
  call random_number(a)
  call PAPIF_flops(realtime, 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(realtime, cputime, fpins, mflops, ierr)
  print *, realtime, cputime, mflops
  print *, papi_flops, 'MFlops'
end
```

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

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

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
Hardware Counter: PCL

- Performance Counter Library
  - Portable API for accessing
    - Hardware performance counters
    - Elapsed time (version 2.0)
  - Supports counting in
    - User mode
    - system mode
    - User+system mode
  - Provides C, C++, Fortran, and Java interfaces
  - Nested counters
  - Thread-safe for Linux, IBM (possibly more) version 2.0

- http://www.fz-juelich.de/zam/PCL/

PCL: API Example

```c
#include <stdio.h>
#include "pcl.h"

PCL_DESCR_TYPE d;
PCL_CNT_TYPE    ires[2];
PCL_FP_CNT_TYPE fpres[2];
int ctr[2]         = { PCL_FP_INSTR, PCL_CYCLES };
unsigned int flags = PCL_MODE_USER;

if (PCLinit(&d) != PCL_SUCCESS) error();
if (PCLstart(d, ctr, 2, flags) != PCL_SUCCESS) error();
/* -- some other code -- */
if (PCLstop(d, ires, fpres, 2) != PCL_SUCCESS) error();
printf("%15.0f flops in %15.0f cycles\n", (double) ires[0], (double) ires[1]);
if (PCLexit(d) != PCL_SUCCESS) error();
```

© 2006 Bernd Mohr, Forschungszentrum Jülich