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

Part I: Terminology and Methodology

Luiz DeRose
Cray Inc.
Mendota Heights, MN 55120
USA
ldr@cray.com

Bernd Mohr
Forschungszentrum Jülich
Jülich Supercomputing Centre
52425 Jülich, Germany
b.mohr@fz-juelich.de

Outline

- Solution techniques
  - Metrics
  - Instrumentation techniques
    - Source code instrumentation
    - Binary instrumentation
  - Instrumentation of parallel programs
    - MPI
    - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
**Common Goals of Performance Analysis**

- Compare alternatives
  - Which configurations are best under which conditions
- Determine the impact of a feature
  - Before-and-after comparison
- System tuning
  - Find parameters that produce best overall performance
- Identify relative performance
  - Which program / algorithm is faster
- Performance debugging
  - Search for bottlenecks
- Set expectations
  - Provide information for users

**Performance Measurement Cycle**

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

- Solution techniques
- **Metrics**
  - Instrumentation techniques
    - Source code instrumentation
    - Binary instrumentation
  - Instrumentation of parallel programs
    - MPI
    - OpenMP
  - Measurement techniques
    - Profiling
    - Tracing

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

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
Speedup and Efficiency

For a given problem A, let

- \( \text{SerTime}(n) \) = Time of the best serial algorithm to solve A for input of size n
- \( \text{ParTime}(n,p) \) = Time of the parallel algorithm + architecture to solve A for input size n, using p processors
- Note that \( \text{SerTime}(n) \leq \text{ParTime}(n,1) \)

Then

- \( \text{Speedup}(p) = \frac{\text{SerTime}(n)}{\text{ParTime}(n,p)} \)
- \( \text{Work}(p) = p \cdot \text{ParTime}(n,p) \)
- \( \text{Efficiency}(p) = \frac{\text{SerTime}(n)}{p \cdot \text{ParTime}(n,p)} \)

Speedup and Efficiency II

In general, expect

- \( 0 \leq \text{Speedup}(p) \leq p \)
- \( \text{Serial work} \leq \text{Parallel work} < \infty \)
- \( 0 \leq \text{Efficiency} \leq 1 \)

- **Linear speedup**: if there is a constant \( c > 0 \) so that speedup is at least \( c \cdot p \). Many use this term to mean \( c = 1 \).
- **Perfect or ideal speedup**: \( \text{speedup}(p) = p \)
- **Superlinear speedup**: \( \text{speedup}(p) > p \) (efficiency > 1)
  - Typical reason: Parallel computer has p times more memory (cache), so higher fraction of program data fits in memory instead of disk (cache instead of memory)
  - Parallel version is solving slightly different, easier problem or provides slightly different answer
### Amdahl’s Law

- **Amdahl [1967] noted:**
  - Given a program, let $f$ be fraction of time spent on operations that must be performed serially (unparallelizable work). Then for $p$ processors:
    \[
    \text{Speedup}(p) \leq \frac{1}{f + (1 - f)/p}
    \]
  - Thus no matter how many processors are used
    \[
    \text{Speedup}(p) \leq \frac{1}{f}
    \]
  - Unfortunately, typical $f$ is 5 - 20%

### Maximal Possible Speedup / Efficiency

[Graph showing speedup and efficiency for different values of $f$.]

- $f=0.001$
- $f=0.01$
- $f=0.1$

© 2009 Luiz DeRose & Bernd Mohr
Amdahl’s Law II

- Amdahl was an optimist
  - Parallelization might require extra work, typically
    - Communication
    - Synchronization
    - Load balancing
  - Amdahl convinced many people that general-purpose parallel computing was not viable

- Amdahl was a pessimist
  - Fortunately, we can break the law!
    - Find better (parallel) algorithms with much smaller values of f
    - Superlinear speedup because of more data fits cache/memory
    - Scaling: time spent in serial portion is often a decreasing fraction of the total time as problem size increase

Scaling

- Sometimes the serial portion
  - is a fixed amount of time independent of problem size
  - or grows with problem size but slower than total time
  - Thus can often exploit large parallel machines by scaling the problem size with the number of processes

- Scaling approaches used for speedup reporting/measurements:
  - Fixed problem size (⇒ strong scaling)
  - Fixed problem size per processor (⇒ weak scaling)
  - Fixed time, find largest problem solvable [Gustafson 1988]
    - Commonly used in evaluating databases (transactions/s)
  - Fixed efficiency: find smallest problem to achieve it (⇒ isoefficiency analysis)
Other Metrics - Load Imbalance Time

- Imbalance Time
  - Metric time to identify code regions that need optimization
  - Two variations:
    - Computation Imbalance Time
      \[ \text{Computation Imbalance Time} = \text{Max Time} - \text{Avg time} \]
    - Synchronization Imbalance Time
      \[ \text{Synchronization Imbalance Time} = \text{Avg Time} - \text{Min time} \]
  - Provides an estimation to the user of how much time in the overall program would be saved if the corresponding section of the code had a perfect balance
  - Represents an upper bound on the "potential saving"

Load Imbalance Metrics Rationale

Between two barriers
User: \( \text{Imb} = \text{Max-Avg} = 99-40 = 59 \)
MPI Sync: \( \text{Avg} = 59 \)
MPI Sync+Comm: \( \text{Avg-Min} = 60-1 = 59 \)
Load Imbalance Metrics

- **MPI Sync Time**
  - Metric that indicates the amount of time a PE (MPI rank) waits on a collective call

- **Imbalance %**
  - Provide an idea of the "badness" of the imbalance
  - Corresponds to the % of the time that the rest of the team, excluding the slowest PE is not engaged in useful work on the given function
  - "Percentage of resources available for parallelism" that is wasted

\[
\text{Imbalance\%} = 100 \times \frac{\text{Imbalance time}}{\text{Max Time}} \times \frac{N}{N-1}
\]

Outline

- Solution techniques
- Metrics
- **Instrumentation techniques**
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
User's mental model of the program does not match the executed version
- Performance tools must be able to revert this semantic gap

Semantic Gap

- **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

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

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

- Example C++ issues:
  - Template instrumentation?
  - Function overloading
  - Executing code before main
  - Operator overloading
- 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());
    }
    ```

TAU Source Code Instrumentor

- Part of the TAU performance framework
- Supports
  - f77, f90
  - C, and C++
- Inserts calls to the TAU monitoring API
- Based on the Program Database Toolkit

- http://tau.uoregon.edu/
Program Database Toolkit

- Based on commercial parsers
  - C, C++: Edison Design Group (EDG)
    - Full ISO 1998 C++ and ISO 1999 C Support
  - Fortran 77, Fortran90: Mutek, [Cleanscape]

- Program Database Utilities and Conversion Tools APplication Environment (DUCTAPE)
  - Object-oriented Access to Static Information
  - Classes, Modules, Routines, Types, Templates, Files, Macros, Namespaces, Comments/Pragmas, Statements (C/C++ only)

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

PDT Architecture and Tools
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, DPCL, and OpenSpeedShop
- Open source
- Supports
  - Mips (IRIX)
  - Power/PowerPC (AIX)
  - Sparc (Solaris)
  - Alpha (Tru64)
  - IA64 (Linux)
  - IA32 (Linux and NT)
  - X86_64 (Linux)

- http://www.dyninst.org
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 around 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

- Solution techniques
- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing

Instrumentation of Parallel Programs

- User-level constructs
  - Modules / components / ...
  - Program phases
  - 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
  - Instrumentation APIs of tools: KOJAK, Vampirtrace, TAU, ...
  - 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
  - Vampirtrace, MPICH MPE, EPILOG, TAU, ...

```
User Program
  Call MPI_Send
  Call MPI_Bcast

MPI Library
  MPI_Send
  MPI_Bcast

Wrapper Library
  PMPI_Send
  MPI_Send
```

© 2009 Luiz DeRose & Bernd Mohr
**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 SC’01


Example:

```c
 !$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)
```

context descriptor
**POMP-like Hooks in Production Compilers**

- POMP was the base for the OpenMP instrumentation hooks provided in production compilers
  - Cray Compiling Environment
  - PGI
  - IBM XL compilers

- These instrumentation hooks are used for performance analysis of OpenMP in production tools
  - CrayPat
  - PGProf

- Also: New OpenMP ARB sanctioned low-level tool interface
  - [http://www.compunity.org/futures/omp-api.html](http://www.compunity.org/futures/omp-api.html)
  - Proof-of-concept implementations by Sun and Intel compilers

---

**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 and Fortran90, OpenMP 2.0
  - C and C++, OpenMP 1.0
  - Additional POMP directives for control and region definition
  - Used by KOJAK, TAU, and ompP
  - Preserves source code information (#line line file)

- Does not support: Instrumentation of user functions

- Download as part of KOJAK: [http://www.fz-juelich.de/jsc/kojak/](http://www.fz-juelich.de/jsc/kojak/)
Current Major OPARI Limitations

- Does not yet support
  - Varying number of threads in different parallel regions
  - Nested parallelism
  - Latest OpenMP 3.0 standard features like tasking

- Executed before compiler preprocessor
  - issues with macros, conditional compilation, includes!

- Needs special care if building ...
  - ... more than one application in one directory
  - ... applications spread over multiple directories

Outline

- Solution techniques
- Metrics
- Instrumentation techniques
  - Source code instrumentation
  - Binary instrumentation
- Instrumentation of parallel programs
  - MPI
  - OpenMP
- Measurement techniques
  - Profiling
  - Tracing
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
  - Communication cost
  - Synchronization cost
  - IO accesses
  - System calls
  - Hardware events

```c
int f1()
{
    int a;
    a = a + 1;
    f2();
    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 $\Leftrightarrow$ expressiveness of data

Measurement Methods: Profiling

- Recording of *aggregated information*
  - Time
  - Counts
    - Calls
    - Hardware counters
- about program and system entities
  - Functions, call sites, 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
  - Running program is interrupted periodically
    - Operating system signal or Hardware counter overflow
    - 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(EXIT, 1);
    ...
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

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

**MONITOR**

```
<table>
<thead>
<tr>
<th>...</th>
<th>1 master</th>
<th>2 slave</th>
<th>3 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A ENTER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>B ENTER</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>A SEND</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A EXIT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>B RECV</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>B EXIT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Tracing: Time-line Visualization**

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

![Timeline Visualization](image-url)
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

- **Current Vampir trace formats**
  - **VTF**: family of historical ASCII and binary formats
    - http://www.cs.uoregon.edu/research/paracomp/tau/vtf3-1.43.tar.gz
  - **OTF**: new Open Trace Format
    - http://www.tu-dresden.de/zih/otf/
- **TAU** performance analysis toolset
  - http://tau.uoregon.edu/docs.php#api
- **EPILOG**: Jülich open-source trace format
  - http://www.fz-juelich.de/jsc/kojak/
- **MPICH Multi-Processing Environment** (*ALOG*, *CLOG*, *SLOG*, *SLOG-2*)
- **Paraver** trace analyzer (BSC, CEPBA)
- For details see Part III
No Single Solution is Sufficient!

Combination of methods, techniques and tools needed

- Instrumentation
  - Source code / binary, static / dynamic, manual / automatic
- Measurement
  - Internal / external trigger, profiling / tracing
- Analysis
  - Statistics, Visualization, Automatic, Data mining, ...

© 2009 Luiz DeRose & Bernd Mohr