Logistics

- Midterm a week from today
  - Open book and open notes
  - All lectures up to and including last Tuesday’s class

- Homework
  - Homework #2 answers posted
  - Homework #3 posted
Parallel Performance and Complexity

- To use a scalable parallel computer well, you must write high-performance parallel programs.
- To get high-performance parallel programs, you must understand and optimize performance for the combination of programming model, algorithm, language, platform, …
- Unfortunately, parallel performance measurement, analysis and optimization can be an easy process.
- Parallel performance is complex.
Parallel Performance Evaluation

- Study of performance in parallel systems
  - Models and behaviors
  - Evaluative techniques

- Evaluation methodologies
  - Analytical modeling and statistical modeling
  - Simulation-based modeling
  - Empirical measurement, analysis, and modeling

- Purposes
  - Planning
  - Diagnosis
  - Tuning
Scalable, optimized applications deliver HPC promise
Optimization through *performance engineering* process
- Understand performance complexity and inefficiencies
- Tune application to run optimally on high-end machines
How to make the process more effective and productive?
What performance technology should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
Process, performance technology, and its use will change as parallel systems evolve
Goal is to deliver effective performance with high productivity value now and in the future
Parallel / distributed systems are complex

- Four layers
  - Application
    - Algorithm, data structures
  - Parallel programming interface / middleware
    - Compiler, parallel libraries, communication, synchronization
  - Operating system
    - Process and memory management, IO
  - Hardware
    - CPU, memory, network

Mapping/interaction between different layers
Factors which determine a program's performance are complex, interrelated, and sometimes hidden.

Application related factors
- Algorithms, dataset sizes, task granularity, memory usage patterns, load balancing, I/O communication patterns.

Hardware related factors
- Processor architecture, memory hierarchy, I/O network.

Software related factors
- Operating system, compiler/preprocessor, communication protocols, libraries.
Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
  - Identifying these circumstances can give clues to where performance problems exist

- Resources may be “virtual”
  - Not actually a physical resource (e.g., thread, process)

- Performance analysis tools are essential to optimizing an application's performance
  - Can assist you in understanding what your program is "really doing”
  - May provide suggestions how program performance should be improved
Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
  - Iterative process to optimize efficiency
  - Efficiency is a relationship of execution time

- So, where does the time go?

- Find your program's hot spots and eliminate the bottlenecks in them
  - **Hot spot**: an area of code within the program that uses a disproportionately high amount of processor time
  - **Bottleneck**: an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays

- Understand *what, where, and how* time is being spent
Sequential Performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when

- “Sequential” factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output
Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - “Parallel” factors lead to overheads
    - concurrency (threading, processes)
    - interprocess communication (message passing)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances
Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
  - Changes in data storage and structure
  - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
  - Cache management is a big one
  - Locality, locality, locality!
  - Virtual memory management may also pay off
- May look for opportunities for better processor usage
Parallel Performance Tuning

- In contrast to sequential performance tuning, parallel performance tuning might be described as conflict-driven or interaction-driven.
- Find the points of parallel interactions and determine the overheads associated with them.
- Overheads can be the cost of performing the interactions:
  - Transfer of data
  - Extra operations to implement coordination
- Overheads also include time spent waiting:
  - Lack of work
  - Waiting for dependency to be satisfied
Parallel Performance Engineering Process

1. Implementation
2. Preparation
3. Performance Analysis
4. Program Tuning
5. Production
Parallel Performance Engineering Process

Implementation

Preparation

Performance Analysis

Program Tuning

Production

Measurement

Refinement

Analysis

Ranking
Performance Observability (Guiding Thesis)

- Performance evaluation problems define the requirements for performance analysis methods.
- Performance observability is the ability to “accurately” capture, analyze, and present (collectively observe) information about computer system/software performance.
- Tools for performance observability must balance the need for performance data against the cost of obtaining it (environment complexity, performance intrusion):
  - Too little performance data makes analysis difficult.
  - Too much data perturbs the measured system.
- Important to understand performance observability complexity and develop technology to address it.
Traditionally an empirically-based approach
- observation ↔ experimentation ↔ diagnosis ↔ tuning

Performance technology developed for each level

- Observation
  - Performance Observation
    - Instrumentation
    - Measurement
    - Analysis
    - Visualization

- Experimentation
  - Performance Experimentation
    - Data mining
    - Models
    - Expert systems

- Diagnosis
  - Performance Diagnosis
    - Data
    - Models
    - Expert systems

- Tuning
  - Performance Tuning
    - Hypotheses
    - Properties

Performance Technology
- Experiment management
- Performance storage
Performance Analysis and Optimization 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
Performance Metrics and Measurement

- Observability depends on measurement
- What is able to be observed and measured?
- A metric represents a type of measured data
  - **Count**: how often some thing occurred
    - calls to a routine, cache misses, messages sent, …
  - **Duration**: how long some thing took place
    - execution time of a routine, message communication time, …
  - **Size**: how big some thing is
    - message size, memory allocated, …

- A measurement records performance data

- Certain quantities can not be measured directly
  - **Derived metric**: calculated from metrics
    - rates of some thing (e.g., flops per second) are one example
Benchmarking typically involves the measurement of metrics for a particular type of evaluation

- Standardize on an experimentation methodology
- Standardize on a collection of benchmark programs
- Standardize on set of metrics

- High-Performance Linpack (HPL) for Top 500
- NAS Parallel Benchmarks
- SPEC

- Typically look at MIPS and FLOPS
How Is Time Measured?

- How do we determine where the time goes?
  
  "A person with one clock knows what time it is, a person with two clocks is never sure."

  Confucious (attributed)

- Clocks are not the same
  - Have different resolutions and overheads for access

- Time is an abstraction based on clock
  - Only as good (accurate) as the clock we use
  - Only as good as what we use it for
Execution Time

- There are different types of time

  - **Wall-clock time**
    - Based on realtime clock (continuously running)
    - Includes time spent in all activities

  - **Virtual process time** (aka CPU time)
    - Time when process is executing (CPU is active)
      - user time and system time (can mean different things)
    - Does not include time when process is inherently waiting

- **Parallel execution time**
  - Runs whenever any parallel part is executing
  - Need to define a global time basis
Observation Types

- There are two types of performance observation that determine different measurement methods
  - Direct performance observation
  - Indirect performance observation

- *Direct performance observation* is based on a scientific theory of measurement that considers the cost (overhead) with respect to accuracy

- *Indirect performance observation* is based on a sampling theory of measurement that assumes some degree of statistical stationarity
Direct Performance Observation

- Execution actions exposed as events
  - In general, actions reflect some execution state
    - presence at a code location or change in data
    - occurrence in parallelism context (thread of execution)
  - Events encode actions for observation

- Observation is direct
  - Direct instrumentation of program code (*probes*)
  - Instrumentation invokes performance measurement
  - Event measurement = performance data + context

- Performance experiment
  - Actual events + performance measurements
Indirect Performance Observation

- Program code instrumentation is not used
- Performance is observed indirectly
  - Execution is interrupted
    - can be triggered by different events
  - Execution state is queried (sampled)
    - different performance data measured
  - Event-based sampling (EBS)
- Performance attribution is inferred
  - Determined by execution context (state)
  - Observation resolution determined by interrupt period
  - Performance data associated with context for period
Direct Observation: Events

- Event types
  - Interval events (begin/end events)
    - measures performance between begin and end
    - metrics monotonically increase
  - Atomic events
    - used to capture performance data state
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops
- User-defined events
  - Specified by the user
- Abstract mapping events
Direct Observation: Instrumentation

- Events defined by instrumentation access
- Instrumentation levels
  - Source code
  - Library code
  - Object code
  - Executable code
  - Runtime system
  - Operating system
- Levels provide different information / semantics
- Different tools needed for each level
- Often instrumentation on multiple levels required
Direct Observation: Techniques

- Static instrumentation
  - Program instrumented prior to execution

- Dynamic instrumentation
  - Program instrumented at runtime

- Manual and automatic mechanisms

- Tool required for automatic support
  - Source time: preprocessor, translator, compiler
  - Link time: wrapper library, preload
  - Execution time: binary rewrite, dynamic

- Advantages / disadvantages
Indirect Observation: Events/Triggers

- Events are actions external to program code
  - Timer countdown, HW counter overflow, …
  - Consequence of program execution
- Event frequency determined by:
  - type, setup, number enabled (exposed)
- Triggers used to invoke measurement tool
  - Traps when events occur (interrupt)
  - Associated with events
  - May add differentiation to events
Indirect Observation: Context

- When events trigger, execution context determined at time of trap (interrupt)
  - Access to PC from interrupt frame
  - Access to information about process/thread
  - Possible access to call stack
    - requires call stack unwinder

- Assumption is that the context was the same during the preceding period
  - Between successive triggers
  - Statistical approximation valid for long running programs assuming repeated behavior
Direct / Indirect Comparison

- Direct performance observation
  - 😊 Measures performance data exactly
  - 😊 Links performance data with application events
  - 😞 Requires instrumentation of code
  - 😞 Measurement overhead can cause execution intrusion and possibly performance perturbation

- Indirect performance observation
  - 😊 Argued to have less overhead and intrusion
  - 😊 Can observe finer granularity
  - 😊 No code modification required (may need symbols)
  - 😞 Inexact measurement and attribution
Measurement Techniques

- When is measurement triggered?
  - External agent (indirect, asynchronous)
    - sampling via interrupts, hardware counter overflow, …
  - Internal agent (direct, synchronous)
    - through code modification (instrumentation)

- How are measurements made (data recorded)?
  - Profiling
    - summarizes performance data during execution
    - per process / thread and organized with respect to context
  - Tracing
    - trace record with performance data and timestamp
    - per process / thread
Critical Issues

- **Accuracy**
  - Timing and counting accuracy depends on resolution
  - Any performance measurement generates *overhead*
    - execution on performance measurement code
  - Measurement overhead can lead to *intrusion*
  - Intrusion can cause *perturbation*
    - alters program behavior

- **Granularity**
  - How many measurements are made
  - How much overhead per measurement

- **Tradeoff (general wisdom)**
  - Accuracy is inversely correlated with granularity
Measured Performance

- Counts
- Durations
- Communication costs
- Synchronization costs
- Memory use
- Hardware counts
- System calls
Profiling

- Recording of aggregated information
  - Counts, time, ...

- ... about program and system entities
  - Functions, loops, basic blocks, ...
  - Processes, threads

- Methods
  - Event-based sampling (indirect, statistical)
  - Direct measurement (deterministic)
Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions

```c
int f1()
{
    int a;
    a = a + 1;
    f2();
    a = a + 1;
    return a;
}
```
Flat and Callpath Profiles

- **Static call graph**
  - Shows all parent-child calling relationships in a program

- **Dynamic call graph**
  - Reflects actual execution time calling relationships

- **Flat profile**
  - Performance metrics for when event is active
  - Exclusive and inclusive

- **Callpath profile**
  - Performance metrics for calling path (event chain)
  - Differentiate performance with respect to program execution state
  - Exclusive and inclusive
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
  - Any event specific information

- An *event trace*
  - Stream of event records sorted by time

- Main advantage is that it can be used to reconstruct the dynamic behavior of the parallel execution
  - Abstract execution model on level of defined events
Event Tracing

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

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

Local trace A

Global trace

Local trace B

merge

unify

1 foo

2 bar
### Tracing: Time-line Visualization

<table>
<thead>
<tr>
<th>1</th>
<th>master</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>slave</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>

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

![Timeline Diagram]

- **Main** task
- **Master** task
- **Slave** task

**Timeline:**
- 58: Execution of `A` enters and sends
- 60: Execution of `B` enters
- 62: Execution of `A` sends
- 64: Execution of `A` exits
- 68: Execution of `B` receives
- 69: Execution of `B` exits
There have been a variety of tracing formats developed over the years and supported in different tools:

- **Vampir**
  - *VTF*: family of historical ASCII and binary formats

- **MPICH / JumpShot**
  - *ALOG, CLOG, SLOG, SLOG-2*

- **Scalasca**
  - *EPILOG* (Jülich open-source trace format)

- **Paraver (BSC, CEPBA)**

- **TAU Performance System**

- **Convergence on Open Trace Format (OTF)**
Profiling / Tracing Comparison

- **Profiling**
  - ☺ Finite, bounded performance data size
  - ☺ Applicable to both direct and indirect methods
  - ☹ Loses time dimension (not entirely)
  - ☹ Lacks ability to fully describe process interaction

- **Tracing**
  - ☻ Temporal and spatial dimension to performance data
  - ☻ Capture parallel dynamics and process interaction
  - ☻ Can derive parallel profiles for any time region
  - ☹ Some inconsistencies with indirect methods
  - ☹ Unbounded performance data size (large)
  - ☹ Complex event buffering and clock synchronization
Gathering performance data is not enough

Need to analyze the data to derive performance understanding

Need to present the performance information in meaningful ways for investigation and insight

Single-experiment performance analysis
  - Identifies performance behavior within an execution

Multi-experiment performance analysis
  - Compares and correlates across different runs to expose key factors and relationships
Performance Tools and Technologies

- It is never the case that performance tools are developed from scratch
- They depend on a range of technologies that can themselves be significant engineering efforts
  - Even simple conceptual things can be hard
- Most technologies deal with how to observe performance metrics or state

"If I have seen further it is by standing on the shoulders of giants."

- Sir Isaac Newton
Technologies

- Timers
- Counters
- Instrumentation
  - Source level
  - Library wrapping (PMPI)
  - Compiler instrumentation
  - Binary (Dyninst, PEBIL, MAQAO)
  - Runtime Interfaces
- Program address resolution
- Stack Walking
- Heterogeneous (accelerator) timers and counters
Time

- How is time measured in a computer system?
- How do we derive time from a clock?
- What clock/time technologies are available to a measurement system?
- How are clocks synchronized in a parallel computer in order to provide a “global time” common between nodes?
- Different technologies are available
  - Issues of resolution and accuracy
Timer: gettimeofday()

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

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

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

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

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

```c
#include <time.h>

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

Clock_gettime(CLOCK_REALTIME, &tv);
walltime = tv.tv_sec + tv.tv_nsec * 1.0e-9;
```
Timer: getrusage()

- UNIX function
- Provides a variety of different information
  - Including user time, system time, memory usage, page faults, and other *resource use* information
  - Information provided system-dependent!

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

struct rusage ru;
double usrt ime; /* seconds */
int memused;

getrusage(RUSAGE_SELF, &ru);
usrt ime = 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
  ```c
  #include <mpi.h>
  double walltime; /* seconds */
  walltime = MPI_Wtime();
  ```
  - Not required to be consistent/synchronized across ranks!

- OpenMP 2.0 also provides a library function
  ```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 typically provide “timers” because underlying them are cycle counters
  - Vendor APIs
    - PMAPI, HWPC, libhpm, libpfm, libperf, …
  - PAPI (Performance API)
What Are Performance Counters

- Extra processor logic inserted to count specific events
- Updated at every cycle (or when some event occurs)

**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 is it well documented
- Lack of standard on what is counted
Hardware Counter Issues

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

- Multi-platform interfaces
  - Performance API (*PAPI*)
    - University of Tennessee, USA
  - Lightweight Performance Tools (*LIKWID*)
    - University of Erlangen, Germany
Hardware Measurement

- Typical measured events account for:
  - Functional units status
    - float point operations
    - fixed point operations
    - load/stores
  - Access to memory hierarchy
  - Cache coherence protocol events
  - Cycles and instructions counts
  - Speculative execution information
    - instructions dispatched
    - branches mispredicted
## Hardware Metrics

- **Typical hardware counter**
  - Cycles / Instructions
  - Floating point instructions
  - Integer instructions
  - Load/stores
  - Cache misses
  - Cache misses
  - Cache misses
  - TLB misses

- **Useful derived metrics**
  - IPC
  - FLOPS
  - 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 hardware components.

- Define threshold values acceptable for metrics and take actions regarding optimization when below/above thresholds.
Accuracy Issues

- Granularity of the measured code
  - If not sufficiently large enough, overhead of the counter interfaces may dominate
  - Mainly applies to time

- 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

- Impact of measurement on counters themselves
  - Typically less of an issue
Hardware Counters Access on Linux

- Linux had not defined an out-of-the-box interface to access the hardware counters!
  - Linux Performance Monitoring Counters Driver (PerfCtr) by Mikael Pettersson from Uppsala X86 + X86-64
    - needs kernel patching!
    - [http://user.it.uu.se/~mikpe/linux/perfctr/](http://user.it.uu.se/~mikpe/linux/perfctr/)
  - Perfmon by Stephane Eranian from HP – IA64
    - it was being evaluated to be added to Linux

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

- There are utilities that start a program and at the end of the execution provide overall event counts
  - `hpmcount` (IBM)
  - `CrayPat` (Cray)
  - `pfmon` from HP (part of Perfmon for Al64)
  - `psrun` (NCSA)
  - `cputrack, har` (Sun)
  - `perfex, ssrun` (SGI)
  - `perf` (Linux)
PAPI – Performance API

- Middleware to provide a consistent and portable API for the performance counter hardware in microprocessors
- Countable events are defined in two ways:
  - Platform-neutral *preset* events
  - Platform-dependent *native* events
- Presets can be derived from multiple native events
- Two interfaces to the underlying counter hardware:
  - *High-level* interface simply provides the ability to start, stop and read the counters for a specified list of events
  - *Low-level* interface manages hardware events in user defined groups called *EventSets*
- Events can be multiplexed if counters are limited

http://icl.cs.utk.edu/papi/
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)
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
Component PAPI

- Developed for the purpose of extending counter sets while providing a common interface
Source Instrumentation with Timers

- Measuring performance using timers requires instrumentation
  - Have to uniquely identify code region (name)
  - Have to add code for timer start and stop
  - Have to compute delta and accumulate statistics

- Hand-instrumenting becomes tedious very quickly, even for small software projects

- Also a requirement for enabling instrumentation only when wanted
  - Avoids unnecessary overheads when not needed
Program Database Toolkit (PDT)

- Used to automate instrumentation of C/C++, Fortran source code
- Source code parser(s) identify blocks such as function boundaries, loop boundaries, ...
- Instrumentor uses parse results to insert API calls into source code files at block enter/exit, outputs an instrumented code file
- Instrumented source passed to compiler
- Linker links application with measurement library
- Free download: http://tau.uoregon.edu
PDT Architecture

Application / Library Source code

Fortran, C / C++

Commercial grade frontend parsers

Program database (PDB)

EDG, Mutek, GNU

DUCTAPE

Tools

TAU Instrumentor
The MPI (Message Passing Interface) standard defines a mechanism for instrumenting all API calls in an MPI implementation.

Each MPI_* function call is actually a weakly defined interface that can be re-defined by performance tools.

Each MPI_* function call eventually calls a corresponding PMPI_* function call which provides the expected MPI functionality.

Performance tools can redefine MPI_* calls.
PMPI Example

- Original MPI_Send() definition:

```c
int __attribute__((weak))
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm) {
    PMPI_Send(buf, count, datatype, dest, tag, comm);
}
```

- Possible Performance tool definition:

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm) {
    MYTOOL_Timer_Start("MPI_Send");
    PMPI_Send(buf, count, datatype, dest, tag, comm);
    MYTOOL_Timer_Stop("MPI_Send");
    MYTOOL_Message_Size("MPI_Send", count * sizeof(datatype));
}
```
Compiler Instrumentation

- Modern compilers provide the ability to instrument functions at compile time
- Can exclude files, functions
- GCC example:
  - `--finstrument-functions` parameter
  - Instruments function entry and exit(s)

```c
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit   (void *this_fn, void *call_site);
```
Measurement libraries have to implement those two functions:

```c
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

The function and call site pointers are instruction addresses.

How to resolve those addresses to source code locations?

- Binutils: libbfd, libiberty (discussed later)
Binary Instrumentation

- Source Instrumentation not possible in all cases
  - Exotic / Domain Specific Languages (no parser support)
  - Pre-compiled system libraries
  - Utility libraries without source available

- Binary instrumentation modifies the existing executable and all libraries, adding user-specified function entry/exit API calls

- Can be done once, or as first step of execution
Binary Instrumentation: Dyninst API

- University of Wisconsin, University of Maryland
- Provides binary instrumentation for runtime code patching:
  - Performance Measurement Tools
  - Correctness Debuggers (efficient data breakpoints)
  - Execution drive simulations
  - Computational Steering

http://www.dyninst.org
Binary Instrumentation: PEBIL

- San Diego Supercomputing Center / PMaC group
- Static binary instrumentation for x86_64 Linux
- PEBIL = PMaC’s Efficient Binary Instrumentation for Linux/x86
- Lightweight binary instrumentation tool that can be used to capture information about the behavior of a running executable

http://www.sdsc.edu/PMaC/projects/pebil.html
Binary Instrumentation: MAQAO

- Modular Assembly Quality Analyzer and Optimizer
- Tool for analyzing and optimizing binary code
- Intel64 and Xeon Phi architectures supported
- Binary release only (for now)

http://maqao.org