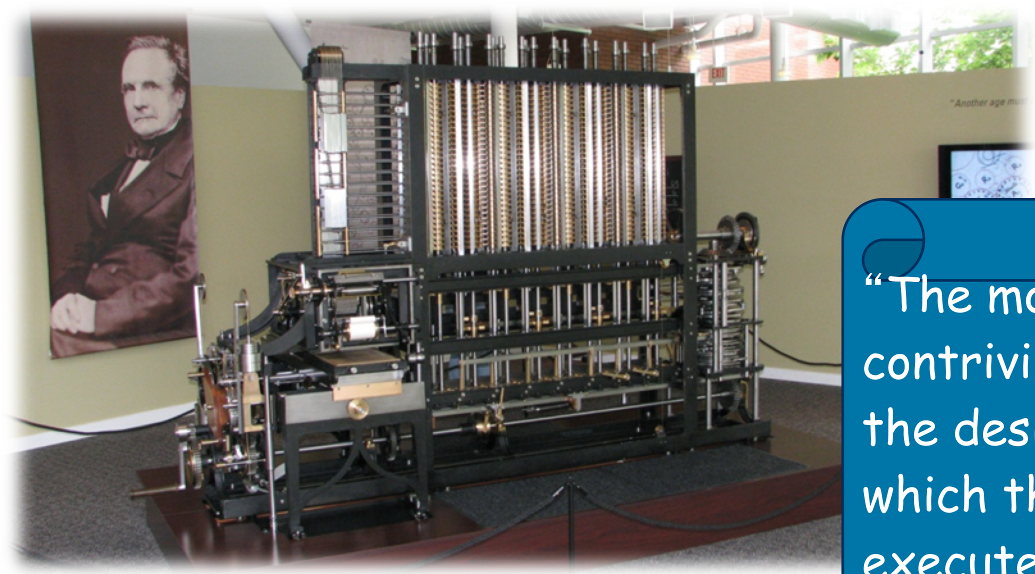
Center for Information Services and High Performance Computing (ZIH)

Introduction to Performance Engineering

OLCF – Tools Workshop

19-20 Jan 2022

Performance: an old problem



Difference Engine

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

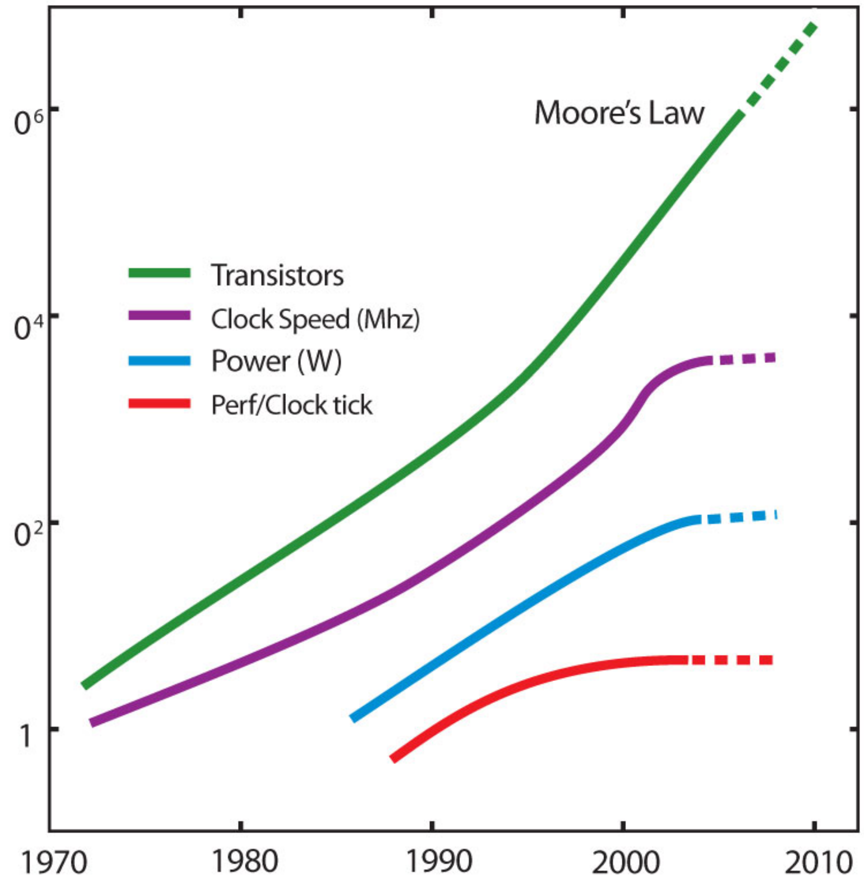
Charles Babbage
1791 – 1871

Today: the “free lunch” is over

Moore's law is still in charge, but
- Clock rates no longer increase
- Performance gains only through increased parallelism

Optimizations of applications more difficult
- Increasing application complexity
  - Multi-physics
  - Multi-scale
- Increasing machine complexity
  - Hierarchical networks / memory
  - More CPUs / multi-core

Every doubling of scale reveals a new bottleneck!



Performance factors of parallel applications

"Sequential" performance factors

- Computation
- Cache and memory
- Input / output

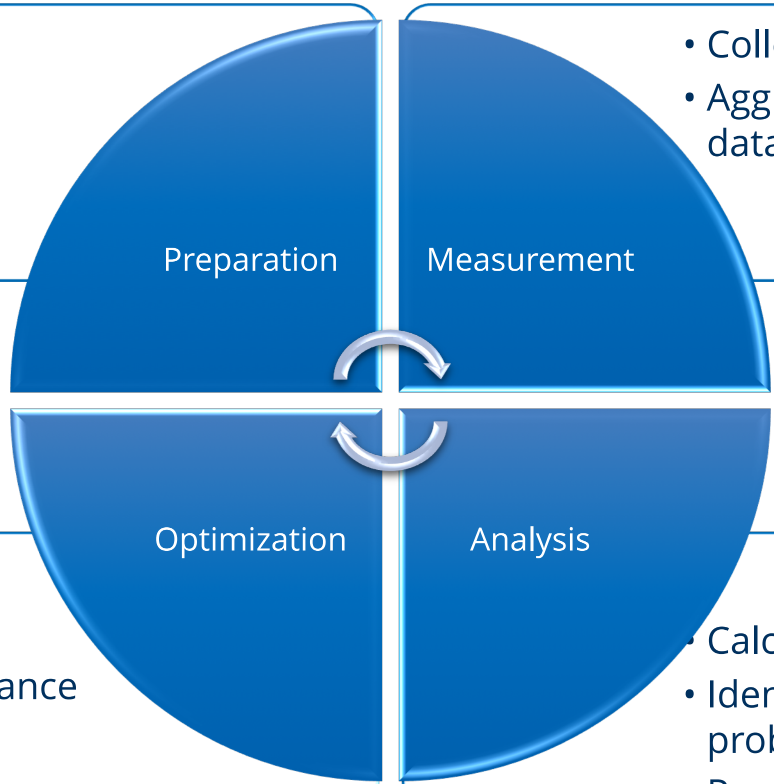
"Parallel" performance factors

- Partitioning / decomposition
- Communication (i.e., message passing)
- Multithreading
- Synchronization / locking

Performance engineering workflow

- Build model of predicted performance
- Select data to measure
- Prepare application with symbols

- Collection of performance data
- Aggregation of performance data



- Modifications intended to eliminate/reduce performance problem

- Calculation of metrics
- Identification of performance problems
- Presentation of results

Parallel Performance Engineering in Practice

Starting point: well-understood, well-optimized code at scale N

Goal: scale to M >> N

Predict behavior: what is the current bottleneck, what performance should we see?

Minimize perturbation

- May require multiple measurements!

Measure possible bottlenecks

- Idle resources
- Changes in profile

Performance Modeling: Predicting Behavior

Simplest models: scaling properties

- What parts of the code are serial and parallel?
- How much time is spent in each?
- How efficient are they currently?

More complex concepts

- Roofline model (comparing throughput to theoretical maxima)
- Load balancing: what code is responsible for idle resources?
- Critical path analysis (e.g. Scalasca)

Strong and weak scaling

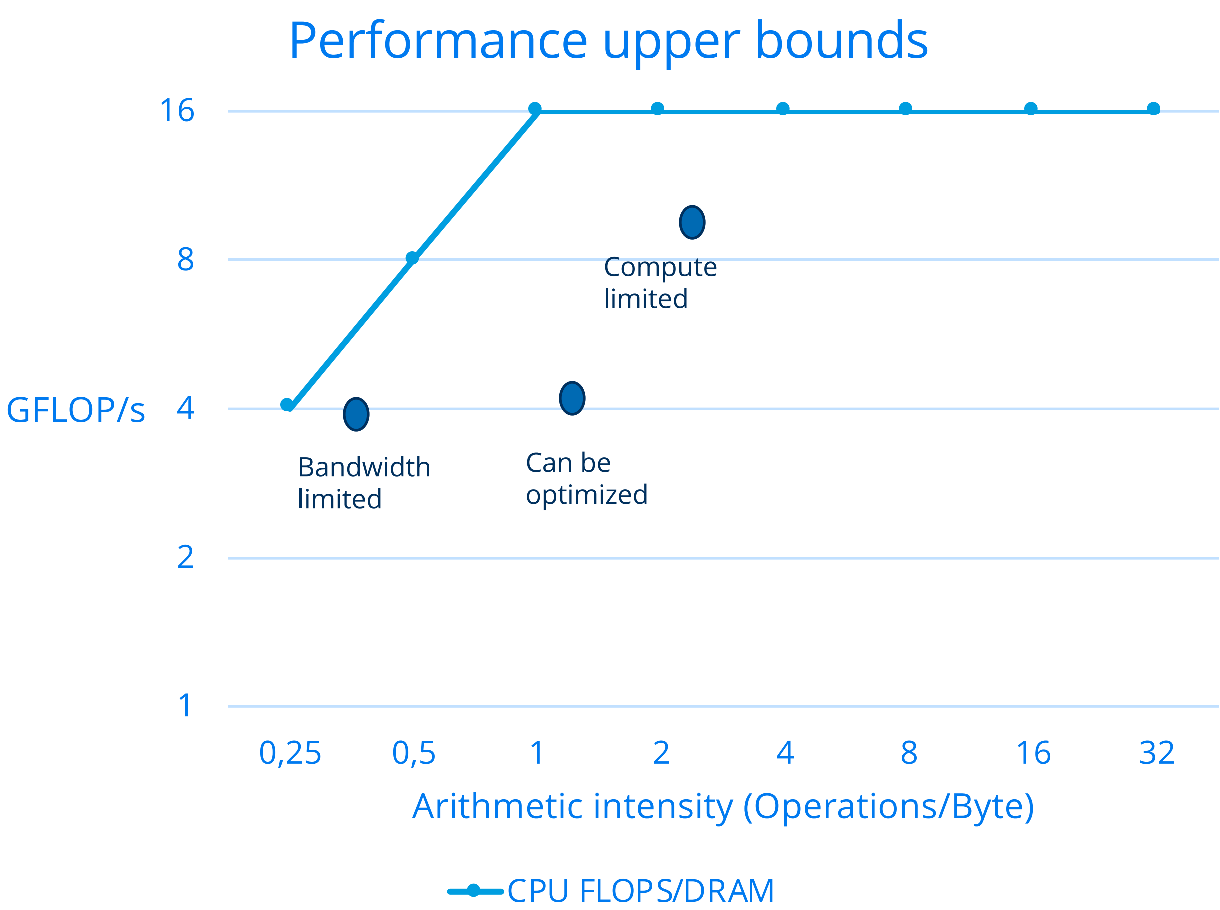
Strong scaling: increasing compute power yields faster solutions on the same problem

— Amdahl's law: Speedup = $\frac{t_{serial}+t_{parallel}}{t_{serial}+\frac{t_{parallel}}{N}}$

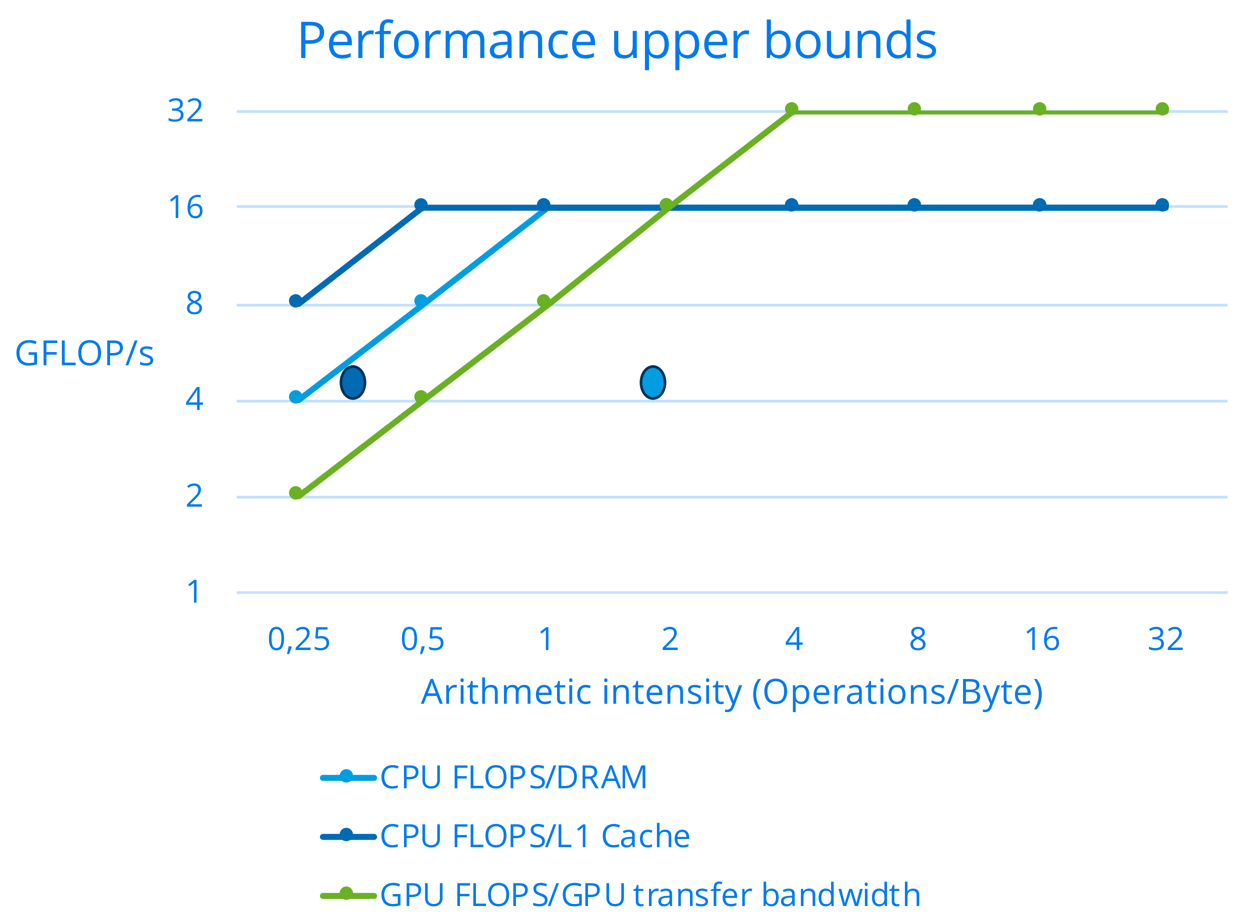
Weak scaling: increasing compute power yields larger problems solved in the same time

— Gustafson's law: convert Amdahl's law to measure scaled speedup (as a factor of problem size)

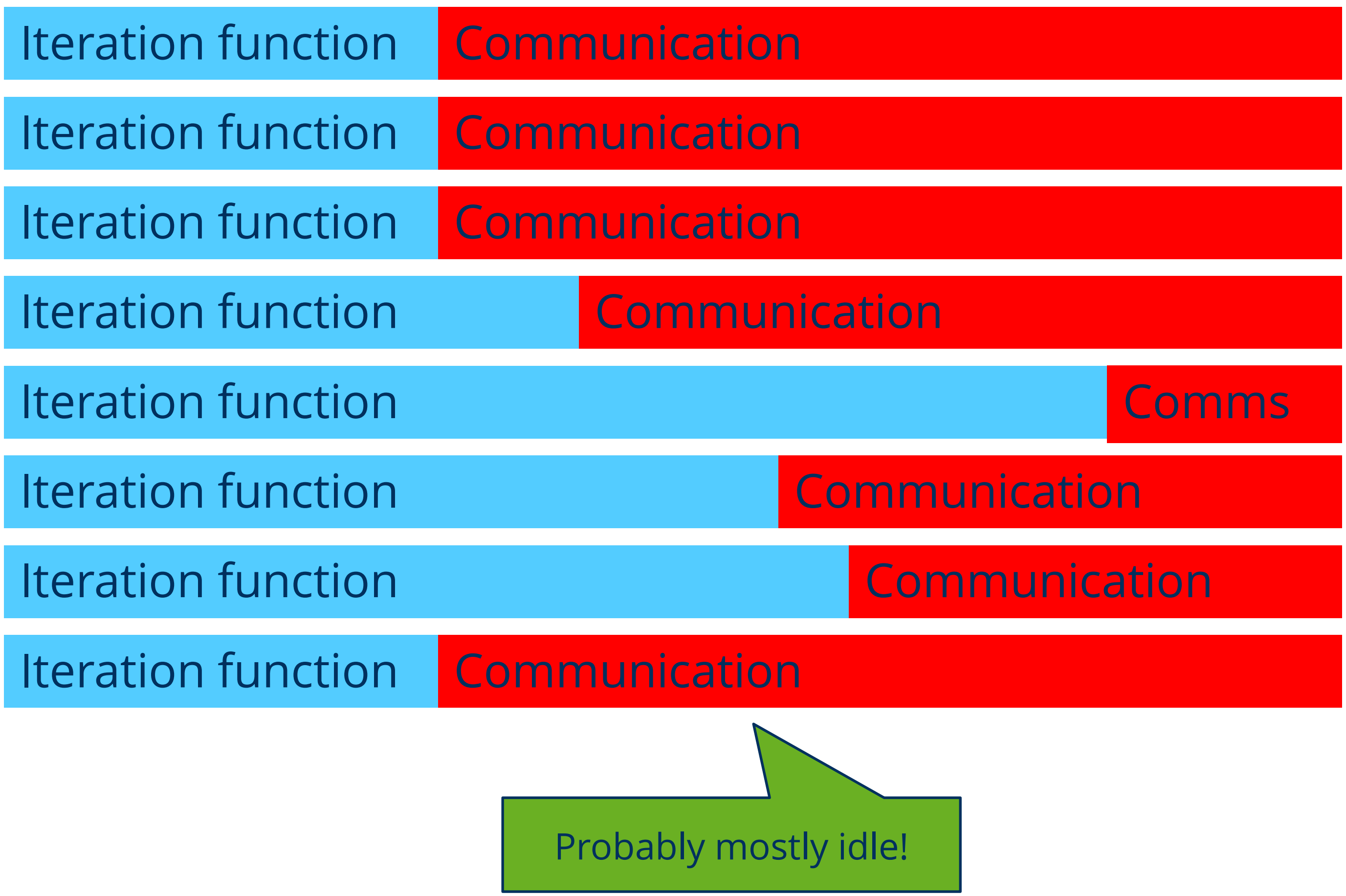
Roofline Model



Multiple Roofline Model



Load balancing



Critical Path Analysis

Key concept: critical path is the sequence of tasks that govern execution time

- At any given time, what is the job waiting for? May be computation or transfer or a combination!

Optimizing tasks off the critical path can’t speed up execution

Example: load imbalance due to a rare case being 2x slower than the common case

- The slow rare case may only be 1% of aggregated execution time, but responsible for 50% of wall time in iterations
- Optimize the critical path = make the rare case closer to the speed of the common case

What to Measure

So you have some hypothesis about how your code will behave

This requires certain data

- Simple scaling models: execution time, possibly subdivided between serial and parallel parts
- Roofline model: operations/second and bytes/second corresponding to one or more rooflines
- Load balancing: distribution of time spent in computation and communication
- Critical path: detailed measurement of execution time across all nodes and threads

Allows you to ignore certain other data

- Example: load balancing
- Detection typically based on communication wait states
- Don't need to analyze computation details for that

When possible, measure only what you need to test your hypothesis

- All-in-one-run only when it's unavoidable

Measurement Practices

Measurements on HPC systems are noisy

— Shared resources: anything short of full-system DAT probably shares something (and maybe even then, if you use site-shared filesystems)

— Nondeterminism: cache effects, which nodes were allocated, small race conditions

Particularly relevant to wall time, but can affect other metrics

As with all scientific measurements, repeat the experiment

— Especially if the initial results look weird!

Measurement issues

Accuracy

— Intrusion overhead

Measurement itself needs time and thus lowers performance

— Perturbation

Measurement alters program behaviour

E.g., memory access pattern

— Accuracy of timers & counters

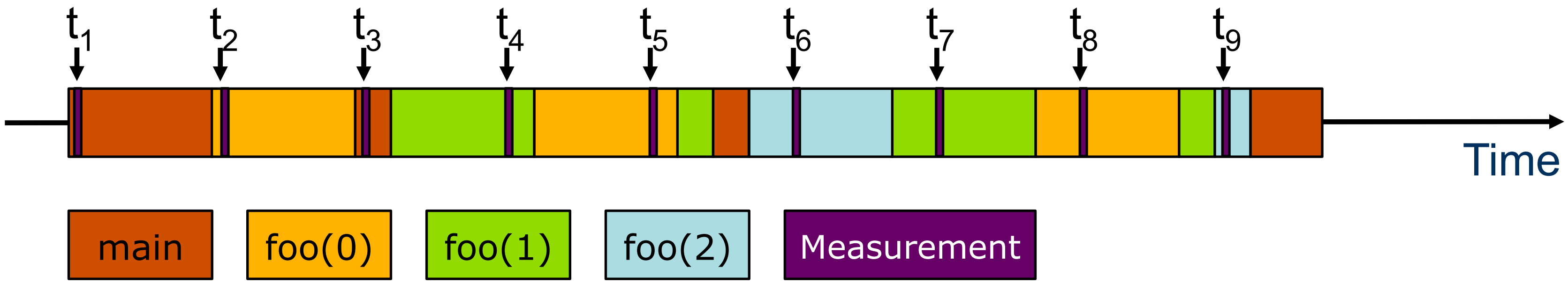
Granularity

— How many measurements?

— How much information / processing during each measurement?

Tradeoff: Accuracy vs. Expressiveness of data

Sampling




```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

Running program is periodically interrupted to take measurement

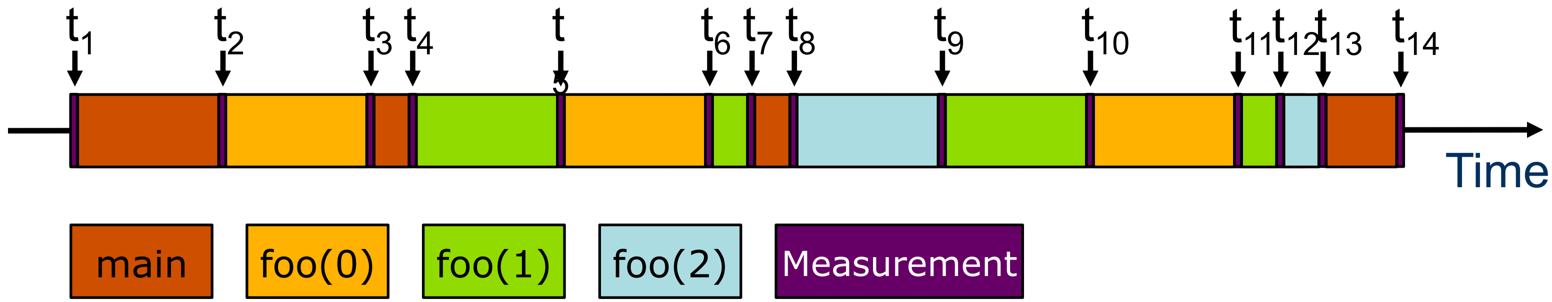
- Timer interrupt, OS signal, or HWC overflow
- Service routine examines return-address stack
- Addresses are mapped to routines using symbol table information

Statistical inference of program behaviour

- Not very detailed information on highly volatile metrics
- Requires long-running applications

Works with unmodified executables

Instrumentation




```
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i - 1);
  Leave("foo");
}
```

Measurement code is inserted such that every event of interest is captured directly

— Can be done in various ways

Advantage:

— Much more detailed information

Disadvantage:

— Processing of source-code / executable necessary

— Large relative overheads for small functions

Profiling / Runtime summarization

Recording of aggregated information
- Total, maximum, minimum, …

For measurements
- Time
- Counts
  - Function calls
  - Bytes transferred
  - Hardware counters

Over program and system entities
- Functions, call sites, basic blocks, loops, …
- Processes, threads

Profile = summarization of events over execution interval

Tracing

Recording detailed information about significant points (events) during execution of the program

- Enter / leave of a region (function, loop, …)
- Send / receive a message, …

Save information in event record

- Timestamp, location, event type
- Plus event-specific information (e.g., communicator, sender / receiver, …)

Abstract execution model on level of defined events

Event trace = Chronologically ordered sequence of event records

Tracing Pros & Cons

Tracing advantages

- Event traces preserve the temporal and spatial relationships among individual events
  (☞ context)
- Allows reconstruction of dynamic application behaviour on any required level of abstraction
- Most general measurement technique
  - Profile data can be reconstructed from event traces

Disadvantages

- Traces can very quickly become extremely large
- Writing events to file at runtime may causes perturbation

No single solution is sufficient!

A combination of different methods, tools and techniques is typically needed!

Analysis

— Statistics, visualization, automatic analysis, data mining, ...

Measurement

— Sampling / instrumentation, profiling / tracing, ...

Instrumentation

— Source code / binary, manual / automatic, ...

Typical performance analysis procedure

Do I have a performance problem at all?

— Time / speedup / scalability measurements

What is the key bottleneck (computation / communication)?

— MPI / OpenMP / flat profiling

Where is the key bottleneck?

— Call-path profiling, detailed basic block profiling

Why is it there?

— Hardware counter analysis, trace selected parts to keep trace size manageable

Does the code have scalability problems?

— Load imbalance analysis, compare profiles at various sizes function-by-function