Performance Analysis Tools

Karl Fuerlinger
fuerling@eecs.berkeley.edu

With slides from David Skinner, Sameer Shende, Shirley Moore, Bernd Mohr, Felix Wolf, Hans Christian Hoppe and others.
Outline

Motivation
- Why do we care about performance

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

Tools
- PAPI: Access to hardware performance counters
- ompP: Profiling of OpenMP applications
- IPM: Profiling of MPI apps
- Vampir: Trace visualization
- KOJAK/Scalasca: Automated bottleneck detection of MPI/OpenMP applications
- TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
Motivation

Performance Analysis is important

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

- Goal: solve larger problems
- Goal: solve problems faster
Outline

- Motivation
  - Why do we care about performance

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

- Tools
  - PAPI: Access to hardware performance counters
  - ompP: Profiling of OpenMP applications
  - IPM: Profiling of MPI apps
  - Vampir: Trace visualization
  - KOJAK/Scalasca: Automated bottleneck detection of MPI/OpenMP applications
  - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
Concepts and Definitions

- The typical performance optimization cycle

Code Development

Measure

Analyze

Modify / Tune

Usage / Production

instrumentation

functionally complete and correct program

complete, correct and well-performing program
Instrumentation

- Instrumentation = adding measurement probes to the code to observe its execution.
- Can be done on several levels.
- Different techniques for different levels.
- Different overheads and levels of accuracy with each technique.
- No instrumentation: run in a simulator. E.g., Valgrind.
Instrumentation – Examples (1)

- **Source code instrumentation**
  - *User added* time measurement, etc. (e.g., `printf()`, `gettimeofday()`)

  - Many **tools** expose mechanisms for source code instrumentation in addition to automatic instrumentation facilities they offer

  - **Instrument program phases:**
    - initialization/main iteration loop/data post processing

  - Pramga and pre-processor based
    ```c
    #pragma pomp inst begin(foo)
    #pragma pomp inst end(foo)
    ```

  - **Macro / function call based**
    ```c
    ELG_USER_START("name");
    ...
    ELG_USER_END("name");
    ```
Instrumentation – Examples (2)

- **Preprocessor Instrumentation**
  - Example: Instrumenting OpenMP constructs with Opari
  - Preprocessor operation

  ![Original source code](image)

  ![Pre-processor](image)

  ![Modified (instrumented) source code](image)

- Example: Instrumentation of a parallel region

  ```c
  #pragma omp parallel {
  #pragma omp for
  #pragma omp nested
  #pragma omp barrier
  #pragma omp flush
  } /* ORIGINAL CODE in parallel region */
  ```

  Instrumentation added by Opari

  This is used for OpenMP analysis in tools such as Kojak/Scalasca/ompP.
Instrumentation – Examples (3)

- Compiler Instrumentation
  - Many compilers can instrument functions automatically
  - GNU compiler flag: `-finstrument-functions`
  - Automatically calls functions on function entry/exit that a tool can capture
  - Not standardized across compilers, often undocumented flags, sometimes not available at all
  - GNU compiler example:

```c
void __cyg_profile_func_enter(void *this, void *callsite) {
    /* called on function entry */
}

void __cyg_profile_func_exit(void *this, void *callsite) {
    /* called just before returning from function */
}
```
Instrumentation – Examples (4)

- Library Instrumentation:

  - MPI library interposition
    - All functions are available under two names: `MPI_xxx` and `PMPI_xxx`. `MPI_xxx` symbols are weak, can be over-written by the interposition library.
    - Measurement code in the interposition library measures begin, end, transmitted data, etc... and calls corresponding PMPI routine.
    - Not all MPI functions need to be instrumented.
Instrumentation – Examples (5)

- **Binary Runtime Instrumentation**
  - Dynamic patching while the program executes
  - Example: Paradyn tool, Dyninst API

- **Base trampolines/Mini trampolines**
  - Base trampolines handle storing current state of program so instrumentations do not affect execution
  - *Mini trampolines* are the machine-specific realizations of predicates and primitives
  - One base trampoline may handle many mini-trampolines, but a base trampoline is needed for every instrumentation point

Binary instrumentation is difficult
  - Have to deal with
    - Compiler optimizations
    - Branch delay slots
    - Different sizes of instructions for x86 (may increase the number of instructions that have to be relocated)
    - Creating and inserting mini trampolines somewhere in program (at end?)
    - Limited-range jumps may complicate this

**PIN: Open Source dynamic binary instrumenter from Intel**

Figure by Skylar Byrd Rampersaud
Measurement

- Profiling vs. Tracing
  
  - Profiling
    - Summary statistics of performance metrics
      - Number of times a routine was invoked
      - Exclusive, inclusive time/hpm 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 log of events
      - Message communication events (sends/receives) are tracked
      - Shows when and from/to where messages were sent
      - Large volume of performance data generated usually leads to more perturbation in the program
Measurement: Profiling

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

```
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 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, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definition:

<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>

| 58 | A     | ENTER | 1 |
| 60 | B     | ENTER | 2 |
| 62 | A     | SEND  | B |
| 64 | A     | EXIT  | 1 |
| 68 | B     | RECV  | A |
| 69 | B     | EXIT  | 2 |
| ... | ... | ... | ... |
Tracing: Timeline Visualization

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

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

... |   |
|---|---|

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Timeline visualization showing interactions between master and slave processes.
Measurement: Tracing

- **Tracing**
  - 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 is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation
Performance Data Analysis

- Draw conclusions from measured performance data

- **Manual analysis**
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling

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

- **Vampir: Timeline view**

![Vampir Timeline View Example](image-url)
Trace File Visualization

- Vampir: message communication statistics
3D performance data exploration

- Paraprof viewer (from the TAU toolset)
Automated Performance Analysis

- **Reason for Automation**
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: ~1.6 million cores
  - Trend to multi-core

- **Large amounts of performance data when tracing**
  - 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**
Automation Example

This is a situation that can be detected automatically by analyzing the trace file

-> *late sender* pattern
Outline

- **Motivation**
  - Why do we care about performance

- **Concepts and definitions**
  - The performance analysis cycle
  - Instrumentation
  - Measurement: profiling vs. tracing
  - Analysis: manual vs. automated

- **Tools**
  - PAPI: Access to hardware performance counters
  - ompP: Profiling of OpenMP applications
  - IPM: Profiling of MPI apps
  - Vampir: Trace visualization
  - KOJAK/Scalasca: Automated bottleneck detection of MPI/OpenMP applications
  - TAU: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
PAPI — Performance Application Programming Interface
What is PAPI

- Middleware that provides a consistent programming interface for the performance counter hardware found in most major micro-processors.

- Started in 1998, goal was a portable interface to the hardware performance counters available on most modern microprocessors.

- Countable events are defined in two ways:
  - Platform-neutral Preset Events (e.g., PAPI_TOT_INS)
  - Platform-dependent Native Events (e.g., L3_MISSES)

- All events are referenced by name and collected into EventSets for sampling

- Events can be **multiplexed** if counters are limited

- Statistical **sampling** and **profiling** is implemented by:
  - Software overflow with timer driven sampling
  - Hardware overflow if supported by the platform
PAPI Hardware Events

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

- PAPI runs on most modern processors and Operating Systems of interest to HPC:
  - IBM POWER{3, 4, 5} / AIX
  - POWER{4, 5, 6} / Linux
  - PowerPC{-32, -64, 970} / Linux
  - Blue Gene / L
  - Intel Pentium II, III, 4, M, Core, etc. / Linux
  - Intel Itanium{1, 2, Montecito?}
  - AMD Athlon, Opteron / Linux
  - Cray T3E, X1, XD3, XT{3, 4} Catamount
  - 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. The **low** level interface manages hardware events in user defined groups called *EventSets*, and provides access to advanced features.
  2. The **high** level interface provides the ability to start, stop and read the counters for a specified list of events.
  3. **Graphical** and end-user tools provide data collection and visualization.
PAPI High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (less additional code) than low-level
- Supports 8 calls in C or Fortran:

  PAPI_start_counters()  PAPI_stop_counters()
PAPI_read_counters()  PAPI_accum_counters()
PAPI_num_counters()  PAPI_flips()
PAPI_ipc()  PAPI_flops()
# PAPI High-level Example

```c
#include "papi.h"
#define NUM_EVENTS 2
long_long values[NUM_EVENTS];
unsigned int Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};

/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);

/* What we are monitoring... */
do_work();

/* Stop counters and store results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```
PAPI Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware, and the memory environment
- Multiplexing
- Callbacks on counter overflow
- Profiling
- About 60 functions
#include "papi.h"
#define NUMEVENTS 2
int Events[NUMEVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int EventSet;
long long values[NUMEVENTS];
/* 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,NUMEVENTS);
/* 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);
Many tools in the HPC space are built on top of PAPI

- TAU (U Oregon)
- HPCToolkit (Rice Univ)
- KOJAK and SCALASCA (UTK, FZ Juelich)
- PerfSuite (NCSA)
- Vampir (TU Dresden)
- Open|Speedshop (SGI)
- ompP (Berkeley)
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 Design

PAPI Framework Layer

Low Level API

Hi Level API

Devel API

PAPI Component Layer (network)
- Kernel Patch
- Operating System
- Perf Counter Hardware

PAPI Component Layer (CPU)
- Kernel Patch
- Operating System
- Perf Counter Hardware

PAPI Component Layer (thermal)
- Kernel Patch
- Operating System
- Perf Counter Hardware
OpenMP

- **OpenMP**
  - Threads and fork/join based programming model
  - Worksharing constructs

**Characteristics**

- Directive based (compiler pragmas, comments)
- Incremental parallelization approach
- Well suited for loop-based parallel programming
- Less well suited for irregular parallelism (tasking included in version 3.0 of the OpenMP specification).
- One of the contending programming paradigms for the “multicore era”
OpenMP Performance Analysis with ompP

- **ompP**: Profiling tool for OpenMP
  - Based on source code instrumentation
  - Independent of the compiler and runtime used
  - Tested and supported: Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers
  - Supports HW counters through PAPI
  - Leverages source code instrumenter **opari** from the KOJAK/SCALASCA toolset
  - Available for download (GLP):
    http://www.ompp-tool.com

![Diagram of ompP process]

- **Source Code**
- Automatic instrumentation of OpenMP constructs, manual region instrumentation
- **Executable**
- **Settings** (env. Vars)
  - HW Counters, output format, …
- Execution on parallel machine
- Profiling Report
Usage example

Normal build process:

```bash
$> icc -openmp -o test test.c
$> ./test
$> hello world
$> hello world
...
```

```c
void main(int argc, char* argv[]) {
    #pragma omp parallel
    {
        #pragma omp critical
        {
            printf("hello world\n");
            sleep(1)
        }
    }
}
```

Build with profiler:

```bash
$> kinst-ompp icc -openmp -o test test.c
$> ./test
$> hello world
$> hello world
...
$> cat test.2-0.ompp.txt
```

```
start.2-0.ompp.txt:

... OmP General Information

Start Date      : Thu Mar 12 17:57:56 2009
End Date        : Thu Mar 12 17:57:58 2009
```
ompP’s Profiling Report

- **Header**
  - Date, time, duration of the run, number of threads, used hardware counters,…

- **Region Overview**
  - Number of OpenMP regions (constructs) and their source-code locations

- **Flat Region Profile**
  - Inclusive times, counts, hardware counter data

- **Callgraph**

- **Callgraph Profiles**
  - With Inclusive and exclusive times

- **Overhead Analysis Report**
  - Four overhead categories
  - Per-parallel region breakdown
  - Absolute times and percentages
Profiling Data

- Example profiling data

**Code:**
```c
#pragma omp parallel
{
    #pragma omp critical
    {  
        sleep(1)
    }
}
```

**Profile:**
```
R00002 main.c (34-37) (default) CRITICAL

<table>
<thead>
<tr>
<th>TID</th>
<th>execT</th>
<th>execC</th>
<th>bodyT</th>
<th>enterT</th>
<th>exitT</th>
<th>PAPI_TOT_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.00</td>
<td>1.00</td>
<td>2.00</td>
<td>0.00</td>
<td>0.00</td>
<td>11132</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>6347</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>3</td>
<td>4.00</td>
<td>1.00</td>
<td>3.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>SUM</td>
<td>10.01</td>
<td>4.00</td>
<td>6.00</td>
<td>0.00</td>
<td>0.00</td>
<td>11132</td>
</tr>
</tbody>
</table>
```

**Components:**
- Region number
- Source code location and region type
- Timing data and execution counts, depending on the particular construct
- One line per thread, last line sums over all threads
- Hardware counter data (if PAPI is available and HW counters are selected)
- Data is exact (measured, not based on sampling)
Flat Region Profile (2)

- Times and counts reported by ompP for various OpenMP constructs

<table>
<thead>
<tr>
<th>construct</th>
<th>main</th>
<th>enter</th>
<th>startup</th>
<th>body</th>
<th>section</th>
<th>sectionC</th>
<th>single</th>
<th>singleC</th>
<th>exitBar</th>
<th>exitT</th>
<th>exitT</th>
<th>shutdwT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BARRIER</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER REGION</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRITICAL</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

___T: time
___C: count

Main = enter + body + barr + exit
Callgraph

- Callgraph View
  - ‘Callgraph’ or ‘region stack’ of OpenMP constructs
  - Functions can be included by using Opari’s mechanism to instrument user defined regions: #pragma pomp inst begin(...), #pragma pomp inst end(...)

- Callgraph profile
  - Similar to flat profile, but with inclusive/exclusive times

- Example:

```c
void foo1() {
    #pragma pomp inst begin(foo1)
    bar();
    #pragma pomp inst end(foo1)
}

void foo2() {
    #pragma pomp inst begin(foo2)
    bar();
    #pragma pomp inst end(foo2)
}

void bar() {
    #pragma omp critical
    { sleep(1.0); }
}

main() {
    #pragma omp parallel
    {     foo1();     foo2();   }
}
```
Callgraph (2)

- Callgraph display

<table>
<thead>
<tr>
<th>Incl. CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.22 (100.0%) [APP 4 threads]</td>
</tr>
<tr>
<td>32.06 (99.50%) PARALLEL +--R00004 main.c (42-46)</td>
</tr>
<tr>
<td>10.02 (31.10%) USERREG</td>
</tr>
<tr>
<td>10.02 (31.10%) CRITICAL</td>
</tr>
<tr>
<td>16.03 (49.74%) USERREG +--R00002 main.c (26-28) ('foo2')</td>
</tr>
<tr>
<td>16.03 (49.74%) CRITICAL +--R00003 main.c (33-36) (unnamed)</td>
</tr>
</tbody>
</table>

- Callgraph profiles (execution with four threads)

```
[*00] critical ia64 omp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
  TID  execI  execE  execC
  0     1.00    0.00   1
  1     3.00    0.00   1
  2     2.00    0.00   1
  3     4.00    0.00   1
  SUM  10.01    0.00   4

[*00] critical ia64 omp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
  TID  execI  execC  bodyI/E  bodyE  enterT  exitT
  0     1.00    1     1.00    1.00    0.00    0.00
  1     3.00    1     1.00    1.00    2.00    0.00
  2     2.00    1     1.00    1.00    1.00    0.00
  3     4.00    1     1.00    1.00    3.00    0.00
  SUM  10.01    4     4.00    4.00    6.00    0.00
```

Karl Fuerlinger
Overhead Analysis (1)

- Certain timing categories reported by ompP can be classified as overheads:
  - Example: `exitBarT`: time wasted by threads idling at the exit barrier of work-sharing constructs. Reason is most likely an imbalanced amount of work.

- Four overhead categories are defined in ompP:
  - `Imbalance`: waiting time incurred due to an imbalanced amount of work in a worksharing or parallel region.
  - `Synchronization`: overhead that arises due to threads having to synchronize their activity, e.g. `barrier` call.
  - `Limited Parallelism`: idle threads due not enough parallelism being exposed by the program.
  - `Thread management`: overhead for the creation and destruction of threads, and for signaling critical sections, locks as available.
## Overhead Analysis (2)

<table>
<thead>
<tr>
<th>construct</th>
<th>main</th>
<th>enter</th>
<th>startup</th>
<th>body</th>
<th>section</th>
<th>single</th>
<th>exit</th>
<th>exitT</th>
<th>shutdunT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>execT</td>
<td>execC</td>
<td>enterT</td>
<td>startupT</td>
<td>bodyT</td>
<td>sectionT</td>
<td>singleT</td>
<td>singleC</td>
<td>exitBarT</td>
</tr>
<tr>
<td>MASTER</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>•</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BARRIER</td>
<td>•</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER REGION</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRITICAL</td>
<td>•</td>
<td>•</td>
<td>S</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>•</td>
<td>•</td>
<td>S</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECTIONS</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**S**: Synchronization overhead  
**I**: Imbalance overhead  
**M**: Thread management overhead  
**L**: Limited Parallelism overhead
ompP’s Overhead Analysis Report

--- ompP Overhead Analysis Report ----------------------------

Total runtime (wallclock) : 172.64 sec [32 threads]
Number of parallel regions : 12
Parallel coverage : 134.83 sec (78.10%)

Parallel regions sorted by wallclock time:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARALL</td>
<td>mgrid.F (360-384)</td>
<td>55.75 (32.29)</td>
</tr>
<tr>
<td>PARALL</td>
<td>mgrid.F (403-427)</td>
<td>23.02 (13.34)</td>
</tr>
<tr>
<td>PARALL</td>
<td>mgrid.F (204-217)</td>
<td>11.94 ( 6.92)</td>
</tr>
<tr>
<td>...</td>
<td>SUM</td>
<td>134.83 (78.10)</td>
</tr>
</tbody>
</table>

Overheads wrt. each individual parallel region:

Total Ovhd (%): Synch (%) + Imbal (%) + Limpar (%) + Mgmt (%)

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Total Ovhd</th>
<th>Synch (%)</th>
<th>Imbal (%)</th>
<th>Limpar (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011</td>
<td>1783.95</td>
<td>0.00</td>
<td>305.75</td>
<td>0.00</td>
<td>31.51</td>
</tr>
<tr>
<td>R00019</td>
<td>736.80</td>
<td>0.00</td>
<td>104.28</td>
<td>0.00</td>
<td>25.66</td>
</tr>
<tr>
<td>R00009</td>
<td>382.15</td>
<td>0.00</td>
<td>96.47</td>
<td>0.00</td>
<td>86.67</td>
</tr>
<tr>
<td>R00015</td>
<td>276.11</td>
<td>0.00</td>
<td>51.15</td>
<td>0.00</td>
<td>17.70</td>
</tr>
<tr>
<td>...</td>
<td>SUM</td>
<td>4314.62</td>
<td>872.92</td>
<td>404.97</td>
<td>100.98</td>
</tr>
</tbody>
</table>

Overheads wrt. whole program:

Total Ovhd (%): Synch (%) + Imbal (%) + Limpar (%) + Mgmt (%)

<table>
<thead>
<tr>
<th>Parallel Region</th>
<th>Total Ovhd</th>
<th>Synch (%)</th>
<th>Imbal (%)</th>
<th>Limpar (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011</td>
<td>1783.95</td>
<td>0.00</td>
<td>305.75</td>
<td>0.00</td>
<td>31.51</td>
</tr>
<tr>
<td>R00009</td>
<td>382.15</td>
<td>0.00</td>
<td>96.47</td>
<td>0.00</td>
<td>86.67</td>
</tr>
<tr>
<td>R00005</td>
<td>264.16</td>
<td>0.00</td>
<td>63.92</td>
<td>0.00</td>
<td>100.98</td>
</tr>
<tr>
<td>R00007</td>
<td>230.63</td>
<td>0.00</td>
<td>68.58</td>
<td>0.00</td>
<td>83.33</td>
</tr>
<tr>
<td>...</td>
<td>SUM</td>
<td>4314.62</td>
<td>872.92</td>
<td>404.97</td>
<td>100.98</td>
</tr>
</tbody>
</table>
OpenMP Scalability Analysis

- **Methodology**
  - Analyze how overheads behave for increasing thread counts
  - Graphs show accumulated runtime over all threads for fixed workload (strong scaling)
  - Horizontal line = perfect scalability

- **Example: NAS parallel benchmarks**
  - Class C, SGI Altix machine (Itanium 2, 1.6 GHz, 6MB L3 Cache)
SPEC OpenMP Benchmarks (1)

- **Application 314.mgrid_m**
  - Scales relatively poorly, application has 12 parallel loops, all contribute with increasingly severe load imbalance
  - Markedly smaller load imbalance for thread counts of 32 and 16. Only three loops show this behavior
  - In all three cases, the iteration count is always a power of two (2 to 256), hence thread counts which are not a power of two exhibit more load imbalance
Application 316.applu
- Super-linear speedup
- Only one parallel region (ssor.f 138-209) shows super-linear speedup, contributes 80% of accumulated total execution time
- Most likely reason for super-linear speedup: increased overall cache size
SPEC OpenMP Benchmarks (3)

- **Application 313.swim**
  - Dominating source of inefficiency is thread management overhead
  - Main source: reduction of three scalar variables in a small parallel loop in swim.f 116-126.
  - At 128 threads more than 6 percent of the total accumulated runtime is spent in the reduction operation
  - Time for the reduction operation is larger than time spent in the body of the parallel region
SPEC OpenMP Benchmarks (4)

- **Application 318.galgel**
  - Scales very badly, large fraction of overhead not accounted for by ompP (most likely memory access latency, cache conflicts, false sharing)
  - lapack.f90 5081-5092 contributes significantly to the bad scaling
    - accumulated CPU time increases from 107.9 (2 threads) to 1349.1 seconds (32 threads)
    - 32 thread version is only 22% faster than 2 thread version (wall-clock time)
    - 32 thread version parallel efficiency is only approx. 0.08

**Whole application**

**Region lapack.f90 5081-5092**
Incremental Profiling (1)

- **Profiling vs. Tracing**
  - **Profiling:**
    - low overhead
    - small amounts of data
    - easy to comprehend, even as simple ASCII text
  - **Tracing:**
    - Large quantities of data
    - hard to comprehend manually
    - allows temporal phenomena to be explained
    - causal relationship of events are preserved

- **Idea: Combine advantages of profiling and tracing**
  - Add a temporal dimension to profiling-type performance data
  - See what happens during the execution without capturing full traces
  - Manual interpretation becomes harder since a new dimension is added to the performance data
Incremental Profiling (2)

- Implementation:
  - Capture and dump profiling reports not only at the end of the execution but several times while the application executes
  - Analyze how profiling reports change over time
  - Capture points need not be regular

“One-shot” Profiling

Incremental Profiling
Incremental Profiling (3)

Possible triggers for capturing profiles:

- **Timer-based, fixed**: capture profiles in regular, uniform intervals: predictable storage requirements (depends only on duration of program run, size of dataset).

- **Timer-based, adaptive**: Adapt the capture rate to the behavior of the application: dump often if application behavior changes, decrease rate if application behavior stays the same.

- **Counter overflow based**: Dump a profile if a hardware counter overflows. Interesting for floating point intensive application.

- **User-added**: Expose API for dumping profiles to the user aligned to outer loop iterations or phase boundaries.
Incremental Profiling

- **Trigger currently implemented in ompP:**
  - Capture profiles in regular intervals
  - Timer signal is registered and delivered to profiler
  - Profiling data up to capture point stored to memory buffer
  - Dumped as individual profiling reports at the end of program execution
  - Perl scripts to analyze reports and generate graphs

- **Experiments**
  - 1 second regular dump interval
  - SPEC OpenMP benchmark suite
    - Medium variant, 11 applications
  - 32 CPU SGI Altix machine
    - Itanium-2 processors with 1.6 GHz and 6 MB L3 cache
    - Used in batch mode
Incremental Profiling Profiling: Data Views (2)

- **Overheads over time**
  - See how overheads change over the application run
  - How is each $\Delta t$ (1sec) spent for work or for one of the overhead classes:
  - Either for whole program or for a specific parallel region
  - Total incurred overhead = integral under this function

**Application: 328.fma3d_m**

Initialization in a critical section, effectively serializing the execution for approx. 15 seconds. Overhead = $31/32 = 96\%$
Incremental Profiling

- **Performance counter heatmaps**
  - x-axis: Time, y-axis: Thread-ID
  - Color: number of hardware counter events observed during sampling period
  - Application “applu”, medium-sized variant, counter: LOADS RETIRED
  - Visible phenomena: iterative behavior, thread grouping (pairs)
IPM — MPI profiling
IPM: Design Goals

- Provide high-level performance profile
  - “event inventory”
  - How much time in communication operations
  - Less focus on drill-down into application

- Fixed memory footprint
  - 1-2 MB per MPI rank
  - Monitoring data is kept in a hash-table, avoid dynamic memory allocation

- Low CPU overhead
  - 1-2 %

- Easy to use
  - HTML, or ASCII-based output format

- Portable
  - Flip of a switch, no reccompilation, no instrumentation
IPM: Methodology

- **MPI_Init()**
  - Initialize monitoring environment, allocate memory

- **For each MPI call**
  - Compute hash key from
    - Type of call (send/recv/bcast/...)
    - Buffer size (in bytes)
    - Communication partner rank
  - Store/update value in hash table with timing data
    - Number of calls,
    - minimum duration, maximum duration, summed time

- **MPI_Finalize()**
  - Aggregate, report to stdout, write XML log
How to use IPM: basics

1) Do “module load ipm”, then run normally
2) Upon completion you get

```bash
##IPMv0.85#################################################################
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55    wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17    %comm     : 32.43
# gbytes  : 2.57604e+00 total    gflop/sec : 2.04615e+00 total
## ###########################################################################
```

Maybe that’s enough. If so you’re done. Have a nice day.

Q: How did you do that? A: MP_EUILIBPATH, LD_PRELOAD, XCOFF/ELF
## IPMv0.85#####################################################################

# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                         mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55              wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17              %comm     : 32.43
# gbytes  : 2.57604e+00 total              gflop/sec : 2.04615e+00 total
#
# [total]    <avg>           min           max
# wallclock  1373.67       21.4636       21.1087       24.2784
# user       936.95       14.6398         12.68           20.3
# system     227.7        3.55781         1.51            5
# mpi        503.853       7.8727         4.2293       9.13725
# %comm      32.4268       17.42           14.6398       41.407
# gflop/sec  2.04614     0.0319709       0.02724       0.04041
# gbytes     2.57604       0.0402507       0.0399284       0.0408173
# gbytes_tx  0.665125     0.0103926     1.09673e-05       0.0368981
# gbyte_rx   0.659763     0.0103088       9.83477e-07       0.0417372
#
Want more detail? IPM_REPORT=full

# PM_CYC                 3.00519e+11   4.69561e+09   4.50223e+09   5.83342e+09
# PM_FPU0_CMPL           2.45263e+10   3.83223e+08    3.3396e+08   5.12702e+08
# PM_FPU1_CMPL           1.48426e+10   2.31916e+08   1.90704e+08    2.8053e+08
# PM_FPU_FMA             1.03083e+10   1.61067e+08   1.36815e+08   1.96841e+08
# PM_INST_CMPL           3.33597e+11   5.21245e+09   4.33725e+09   6.44214e+09
# PM_LD_CMPL             1.03239e+11   1.61311e+09   1.29033e+09   1.84128e+09
# PM_ST_CMPL             7.19365e+10   1.12401e+09   8.77684e+08   1.29017e+09
# PM_TLB_MISS             1.67892e+08   2.62332e+06   1.16104e+06   2.36664e+07
#
# MPI_Bcast                  352.365          2816         69.93        22.68
# MPI_Waitany                81.0002        185729         16.08         5.21
# MPI_Allreduce              38.6718          5184          7.68         2.49
# MPI_Allgatherv             14.7468           448          2.93         0.95
# MPI_Isend                  12.9071        185729          2.56         0.83
# MPI_Gatherv                2.06443           128          0.41         0.13
# MPI_Irecv                   1.349        185729          0.27         0.09
# MPI_Waitall                0.606749          8064          0.12         0.04
# MPI_Gather                 0.0942596         192          0.02         0.01

###############################################################################
IPM: XML log files

- There’s a lot more information in the logfile than you get to stdout. A logfile is written that has the hash table, switch traffic, memory usage, executable information, ...

- Parallelism in writing of the log (when possible)

- The IPM logs are durable performance profiles serving

  - HPC center production needs: https://www.nersc.gov/nusers/status/ljson/
  http://www.sdsc.edu/user_services/top/ipm/

  - HPC research: ipm_parse renders txt and html
  http://www.nersc.gov/projects/ipm/ex3/

  - your own XML consuming entity, feed, or process
Message Sizes: CAM 336 way

- per MPI call
- per MPI call & buffer size
Scalability: Required

32K tasks AMR code

What does this mean?
More than a pretty picture

Discontinuities in performance are often key to 1\textsuperscript{st} order improvements

But still, what does this really mean? How the !@#!& do I fix it?
Scalability: Insight

- Domain decomp
- Task placement
- Switch topology

Aha.
Portability: Profoundly Interesting

A high level description of the performance of a well known cosmology code on four well known architectures.
Vampir – Trace Visualization
Vampir overview statistics

- Aggregated profiling information
  - Execution time
  - Number of calls

- This profiling information is computed from the trace
  - Change the selection in main timeline window

- Inclusive or exclusive of called routines
Timeline display

- To *zoom*, mark region with the mouse
Timeline display – zoomed
Timeline display – contents

- Shows all selected processes
- Shows state changes (activity color)
- Shows messages, collective and MPI–IO operations
- Can show parallelism display at the bottom
Timeline display – message details

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

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

- **Message statistics** by length, tag or communicator
  - Byte and message count
  - Min/max/avg bandwidth
Collective operations

- For each process: mark operation locally
- Connect start/stop points by lines
Collective operations

- Filter collective operations
- Change display style
Collective operations statistics

- Statistics for collective operations:
  - operation counts, Bytes sent/received
  - transmission rates
- Filter for collective operation

MPI_Gather only
Activity chart

- Profiling information for all processes
Process–local displays

- Timeline (showing calling levels)
- Activity chart
- Calling tree (showing number of calls)
Effects of zooming

Updated message statistics

Updated summary

Select one iteration
KOJAK / Scalasca
Basic Idea

- "Traditional" Tool
  - For non-standard / tricky cases (10%)
  - For expert users

- Automatic Tool
  - For standard cases (90% ?!)
  - For "normal" users
  - Starting point for experts

⇒ More productivity for performance analysis process!
MPI-1 Pattern: Wait at Barrier

- Time spent in front of MPI synchronizing operation such as barriers
MPI-1 Pattern: Late Sender / Receiver

- **Late Sender**: Time lost waiting caused by 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?

Location
How is the problem distributed across the machine?

Region Tree
Where in source code? In what context?

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
TAU
TAU Parallel Performance System

- [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)

- 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
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
ParaProf – 3D Scatterplot (SWEEP3D CUBE)
PerfExplorer - Cluster Analysis

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

- 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**

- **ompP**
  - [http://www.ompp-tool.com](http://www.ompp-tool.com)

- **IPM**

- **TAU**
  - [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)

- **VAMPIR**
  - [http://www.vampir-ng.de/](http://www.vampir-ng.de/)

- **Scalasca**
  - [http://www.scalasca.org](http://www.scalasca.org)
The space is big

There are many more tools than covered here

- Vendor’s tools: Intel VTune, Cray PAT, SUN Analyzer,...
  - Can often use intimate knowledge of the CPU/compiler/runtime system
  - Powerful
  - Most of the time not portable

- Specialized tools
  - STAT debugger tool for extreme scale at Lawrence Livermore Lab

Thank you for your attention!
Backup Slides
Sharks and Fish II

- Sharks and Fish II: $N^2$ force summation in parallel
- E.g. 4 CPUs evaluate force for a global collection of 125 fish

| 31 | 31 | 31 | 32 |

- Domain decomposition: Each CPU is “in charge” of ~31 fish, but keeps a fairly recent copy of all the fishes positions (replicated data)
- Is it not possible to uniformly decompose problems in general, especially in many dimensions
- Luckily this problem has fine granularity and is 2D, let’s see how it scales
Sharks and Fish II: Program

Data:
- n_fish is global
- my_fish is local
- fish_i = \{x, y, \ldots\}

MPI_Allgatherv(myfish_buf, \text{len}[\text{rank}], ..

for (i = 0; i < my_fish; ++i) {
    for (j = 0; j < n_fish; ++j) {
        // i!=j
        a_i += g * mass_j * (fish_i - fish_j) / r_{ij}
    }
}

Move fish
Sharks and Fish II: How fast?

Running on a machine seaborgfranklin.nersc.gov

- 100 fish can move 1000 steps in
  - 1 task → 0.399s
  - 32 tasks → 0.194s

  2.06x speedup

- 1000 fish can move 1000 steps in
  - 1 task → 38.65s
  - 32 tasks → 1.486s

  26.0x speedup

What’s the “best” way to run?
- How many fish do we really have?
- How large a computer do we have?
- How much “computer time” i.e. allocation do we have?
- How quickly, in real wall time, do we need the answer?

1 Seaborg – Franklin more than 10x improvement in time, speedup factors remarkably similar…
Scaling: Good 1st Step: Do runtimes make sense?

```
In[31]:= wtime1 = {100, 0.399197, 200, 1.16549, 300, 3.5097, 400, 6.2177, 500, 9.69267, 600, 13.9481, 700, 18.9689, 800, 24.7653, 900, 31.3224, 1000, 38.6466};

Number of fish

Wallclock time

s1 = Fit[wtime1, 1, x^2, x]
0.0299005 + 0.000038633 x^2
```
Walltime is (all)important but let’s define some other scaling metrics
Scaling: Definitions

- Scaling studies involve changing the degree of parallelism.
  - Will we be change the problem also?

- Strong scaling
  - Fixed problem size

- Weak scaling
  - Problem size grows with additional resources

- Speed up = $T_s/T_p(n)$

- Efficiency = $T_s/(n*T_p(n))$

Be aware there are multiple definitions for these terms
Scaling: Speedups
Scaling: Efficiencies
Scaling: Analysis

- In general, changing problem size and concurrency expose or remove compute resources. Bottlenecks shift.

- In general, first bottleneck wins.

- Scaling brings additional resources too.
  - More CPUs (of course)
  - More cache(s)
  - More memory BW in some cases