Profiling and Performance Analysis

Mark Howison

User Services & Support

Other upcoming workshops

- "Parallel I/O Libraries and Techniques"
  - Monday, April 4, 1-2pm, Petteruti Lounge

- We will probably repeat this semester's workshop schedule every semester
- We may also plan a multi-day "boot camp" in the summer, covering the same topics
- Please let us know if you have specific requests for other topics!

Overview

- What is profiling?
- Easy serial profiling
- Memory profiling with Valgrind
- OProfile for batch jobs
- IPM for MPI applications
- Python profiling
- CUDA profiling

What is profiling?

- Analyzing the behavior of a program *while it is running*
- Different methods:
  - *Sampling* means polling the status of the program at regular time intervals
    - Resulting profiles are statistical – not exact
    - Usually has minimal interference with program's runtime
  - *Tracing* or *instrumenting* means interposing profiling calls within your program's regular calls
    - More exact, but also more resource-intensive
- Usually refers to analyzing CPU performance, but can also apply to memory or I/O

Why or when to profile?

- During performance optimization: goal is to find and remove bottlenecks in your program
- You can find performance problems by...
  - Making a wild guess (“Maybe it is X”)
  - Making an educated guess (“Last time it was X, so maybe it is X this time too”)
  - Taking advice from an expert (“Prestigious Author says it is usually X”)
- OR you could *profile* to collect evidence that helps you narrow down the possibilities
- There is a trade-off between the time it takes to profile vs. for guess and check
  - Utility of profiling usually grows with size/complexity of code

Easy serial profiling

- Simplest way to sample: stop your program where it is taking a long time, and see what it is doing
- Works if your program runs on timescales >1s
- “ezprofile” script on Oscar does just this
- It wraps functionality from the binutils package:
  - “pstack” shows a stack trace of a running program
  - “addr2line” matches a program address with its corresponding line in source code
  - Requires compiling with debug symbols (-g)

Memory profiling with Valgrind

- Lots of functionality, but also high overhead
  - Your program could run many times slower
- Available on Oscar with:
  `module load valgrind`
- Use on serial programs with:
  `valgrind program [args]`
- Good for finding memory leaks
  `--leak-check=full`
- Will also identify the cause of most segfaults (debugging more than profiling...)

OProfile

- The Linux kernel comes with a built-in profiler
- You can use it to profile an *entire system*
  - Can also filter the data to isolate a specific program
- Requires root access to load the kernel module and start/stop the daemon
- But on Oscar, this can be done *automatically* for your batch job if you qsub with "-T oprofile"
- The profile data will be saved to: /gpfs/scratch/shared/oprofile/<jobid>/<node>

Viewing OProfile data

- “opreport” parses the OProfile output
- Whole system:
  `opreport –session-dir=<profile_data>`
- Single program/library:
  `opreport -l –session-dir=<profile_data> /path/to/program`
- “opannotate” can correlate profile results with lines in your source code

IPM for MPI programs

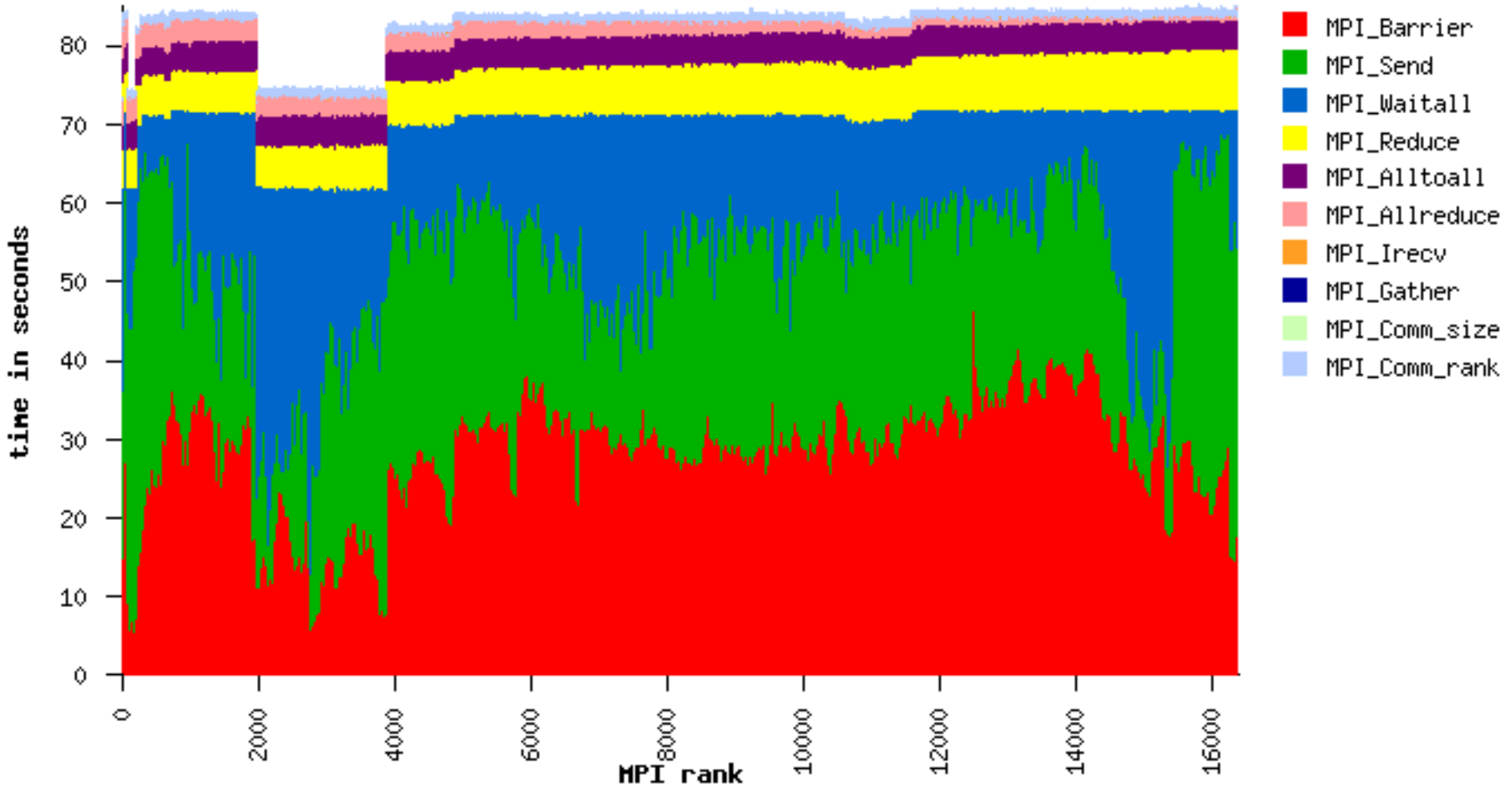
- The Integrated Performance Monitor will show high-level stats about MPI communication in your program
- The MPI library exposes “hooks” for intercepting each MPI call, which IPM uses to start/stop timers
- Scalable, low overhead
- Available on Oscar with:
  `module load ipm`
- Once loaded, automatically profiles any mpirun program
  - Uses LD_PRELOAD to load itself before your program

IPM output

- Writes a summary to stdout after your program calls MPI_Finalize()
- Set IPM_REPORT=full to enable more detail in the summary
- Outputs a detailed XML file to: `/gpfs/scratch/shared/ipm/<username>.<id>.xml`
- Use `ipm_parse` tool to parse XML file

IPM output (Cont'd)

- Example of plots that can be generated from XML:
  (see http://ipm-hpc.sourceforge.net/)



Python profiling

- Use the cProfile module
- To profile a function from within a script or the interpreter:

```
import cProfile
cProfile.run('func()' [, 'output_file'])
```

- To profile an entire script from the command line:

```
python -m cProfile [-o output_file] ...
```

CUDA Profiling

- Built in profiler will provide information on data transfers and kernel execution
- Simply set the appropriate environment variables:
  - CUDA_PROFILE=1 (turns profiling on)
  - CUDA_PROFILE_CONFIG=file
    (points to text file that lists performance counters)
  - CUDA_PROFILE_CSV=1
    (enables CSV output; easier to import into Excel, etc.)
- For list of counters and other options, see "doc/ComputeProfiling.txt" in the CUDA Toolkit
- Helpful overview from SC10:
  http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Analysis_Driven_Optimization.pdf

Performance tips

Use Existing Optimizations

- Link against optimized libraries when you can (BLAS, LAPACK) instead of reinventing the wheel
  - Although these libraries may be optimized for very large data, so if you have small data, it can still be better to write your own routine
- Use MPI collectives instead of point-to-point communication when possible
  - Usually the collectives have additional optimizations that are specific to the system you are running

Performance tips (Cont'd)

In Your Own Code

- Pay attention to how you order multi-dimensional arrays:
  - C expects the last dimension is sequential in memory
  - Fortran expects the first dimension is sequential
  - If you use the wrong layout, your program will stride through memory very inefficiently!
- Function calls have some overhead (usually worse in C++)
  - Sometimes it can help to force small functions to be “inlined” (meaning copied in place instead of called)
- Conditionals within nested loops can be expensive because of branch mispredictions

Performance tips (Cont'd)

In Your Own Code

- Replace expensive operations like division, exp, log, trig functions, etc. with precomputed lookup tables
  - Only works if you are operating on the same set of values over and over again
  - Sometimes you can also find versions that are faster but less precise, if that is acceptable for your computation
- In nested C loops, use the "__restrict__" keyword to indicate when arrays are disjoint (they don't overlap)
- Avoid lots of mallocing/freeing or new/delete operations
  - Also consider using a malloc replacement like Hoard