HPC Workshop Series: *Profiling*

Nick Featherstone

Research Computing

Slides:
https://github.com/ResearchComputing/hpc_profiling.git

Outline

- Basic Timing
  - the Linux “time” utility
  - timing functions in C++, Python, and Fortran
- Serial Profiling with Intel Vtune
  - Command Line
  - GUI
- Parallel Scaling Analysis

Before we begin

- ssh -X username@tlogin1.rc.colorado.edu
- ssh –X scompile
- Clone the repository (all one line):
- git clone https://github.com/ResearchComputing/hpc_profiling.git

Before we begin

- Start an interactive session:
  - sinteractive –n4 –t60 --reservation=feathern-workshop
  - Set up the environment and compile the code:
  - module load intel impi python/3.5.1
  - cd hpc_profiling
  - make

Nano survival guide

- Simple, text-based editor
- These 4 commands are useful

- Ctrl+o save (need to confirm filename)
- Ctrl+x exit
- Ctrl+k cut
- Ctrl+u paste

The Linux Time Utility

- First place to start when profiling your program

```
export OMP_NUM_THREADS=4
time ./prog.omp  (try ./prog.serial also)

real    0m17.801s    Wall Clock Time
user    0m58.125s    Threads x Wall Clock Time
sys     0m0.081s     System Overhead
```

- Works with MPI as well
- Only reports info on cpu that called mpirun

```
time mpirun –np 4 ./prog.mpi
```

Fine-Grained Timing

- Often useful to time portions of a program
- Good idea when developing your own code
- Tough when its 3rd-party software
- Useful functions:
  - Fortran: system_clock
  - C++ clock()
  - Python: time.time( )
- Examples follow

Simple Timing (Fortran)

- Use the system_clock function (F90 onward)

```
INTEGER :: t1, t2, count_rate, count_max
REAL*8 :: elapsed
CALL SYSTEM_CLOCK(t1, count_rate, count_max)
test code
CALL SYSTEM_CLOCK(t2, count_rate, count_max)
elapsed = real(t2-t1) / real(count_rate)
WRITE(6,*)'Time (s): ', elapsed
```

Simple Timing (C++)

```
#include <iostream>
#include <cstdio>
#include <ctime>
int main() {
    std::clock_t tstart;
    std::clock t_end;
    double elapsed;
    tstart = std::clock();
    test code
    tend=std::clock();
    elapsed = ( tend- tstart ) / (double) CLOCKS_PER_SEC;
    std::cout<<"printf: "<< elapsed <<'\n';
}
```

- With c++ 11, can use std::chrono as well

Simple Timing (Python)

- Use the **time** function from the **time** module

```
import time
t1 = time.time( )
Test code
t2 = time.time( )
seconds = t2-t1
print('Elapsed time: ', seconds)
```

Advanced Timing (Python)

- Use the cProfile module to profile your code
- https://docs.python.org/3/library/profile.html
- Examine:
  - my_code.py
  - time_my_code.py
- Run **python time_my_code.py**

Timing with MPI

- Use the MPI_WTIME function
- Calling syntax same in C++ & FORTRAN
- Can be synchronized across all processes

```
USE MPI
REAL*8 :: t1,t2
REAL*8 :: elapsed
t1 = MPI_WTIME()
test code
t2 = MPI_WTIME()
elapsed = t2 - t1
WRITE(6,*)'Time (s): ', elapsed
```

Good Idea: Write a Timer Class

- Examine Serial_Timing.f90
- Makes it easy to change our clock function if desired
- Helps prevent typos!

Attributes
Accrued time
Time1
Time2

Methods
Start clock
Stop clock
Increment Accrued Time

Profiling with Intel Vtune

- Provided with the intel compiler
- Installed on Summit
- In-depth profiling tool
- We only have time for a quick test run
- Online tutorials here:
  - https://software.intel.com/en-us/articles/intel-vtune-amplifier-tutorials

Preparing to Run Vtune

- Additional environment variables need to be set:
  - Run **source ready_vtune**

Vtune: Command Line

- Runs in two stages: collection & reporting
- Have a look at serial.F90
- Try this:

```
amplxe-cl -collect hotspots ./prog.serial
```

collection

```
amplxe-cl -report top-down -r r000hs > hotspots.txt
nano hotspots.txt
```

reporting

Vtune GUI

- Fairly sophisticated GUI interface
- Let's have a look (brace yourself)
- Follow along with me. Start by typing:

  amplxe-gui

Close your interactive session

- Type exit
- Should see prompt change to shas0136 or 0137

Part II: Parallel Profiling

Outline

- What do we mean by “scaling”?
- Strong vs. Weak Scaling
- Example and Exercise

Analysis of Parallel Performance

Run a sample problem at multiple core counts and measure elapsed time. (i.e., conduct a scaling study).

As core count increases, how does your application perform relative to ideal behavior?

Weak Scaling:

Increase problem size alongside number of cores used.

"Can I run a larger problem?"

Strong Scaling:

Fix problem size, and increase core count.

"Can I decrease my time to solution?"

Ideal Scaling

The expected performance in the absence of any communication overhead, etc.

Never achieved in practice, but standard reference point.

Ideal weak scaling:

Time-to-solution is independent of core-count (though may depend on global problem size)

Ideal strong scaling:

Time-to-solution decreases as $1/n_{cores}$

Efficiency

- Efficiency = Expected Time / Actual Time
- Expected (ideal) time is often taken relative to application's serial performance, or performance at a low core count.
- Good efficiency? Who knows? Trade-off between CPU hours and human effort.
- Suggestion: Always run your application with core counts that realize at least 80% efficiency (90 is better…).

Scaling Study Procedure

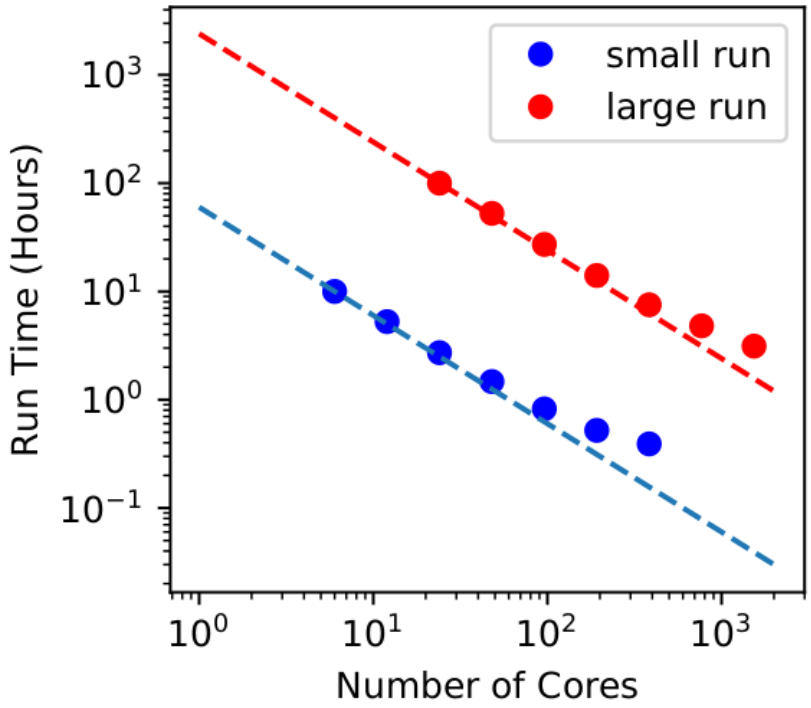
- Pick a few representative problem sizes (e.g., number of images to process, size of grid domain, number of genomes to examine)
- Decide on a sensible duration for the test. If code iterates in some fashion, run for just a few iterations.
- Run code for a short time at each problem size and with multiple core counts.
- For each test, record or calculate elapsed time, expected time, efficiency
- Plot results and ask for CPU time!

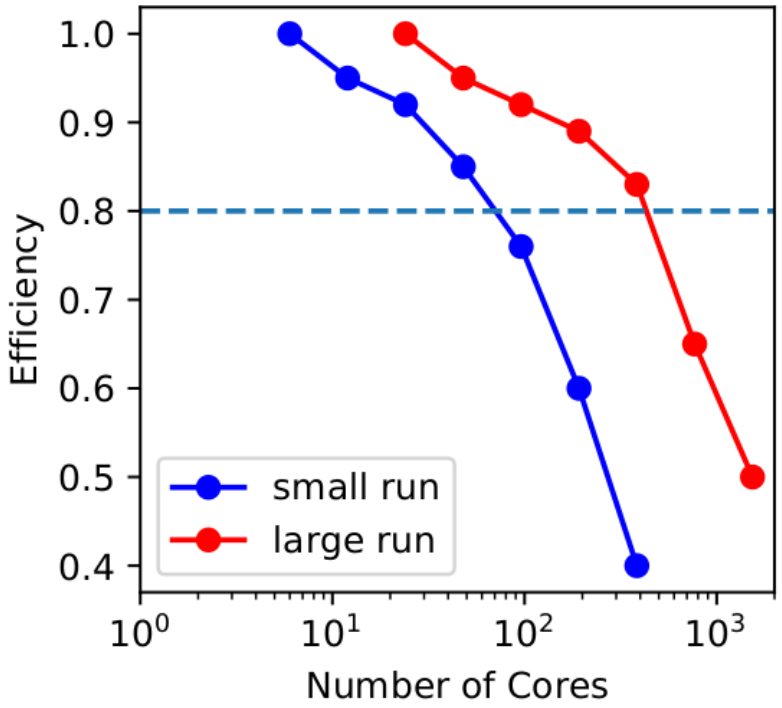
Sample Strong Scaling Data

Fixed problem size at different core counts.

Strong-Scaling Data for Fluid Simulations			
Small-Run (128^3) Timings			
Cores	Measured Time (seconds)	Ideal Time (seconds)	Efficiency
6	60.5	60.50	1.00
12	31.84	30.25	0.95
24	16.44	15.13	0.92
48	8.90	7.56	0.85
96	4.98	3.78	0.76
192	3.15	1.89	0.60
384	2.36	0.95	0.40
Large-Run (512^3) Timings			
24	181.20	181.20	1.00
48	95.37	90.60	0.95
96	49.24	45.30	0.92
192	25.45	22.65	0.89
384	13.64	11.33	0.83
768	8.71	5.67	0.65
1536	5.6625	2.83	0.50

Strong Scaling Plots





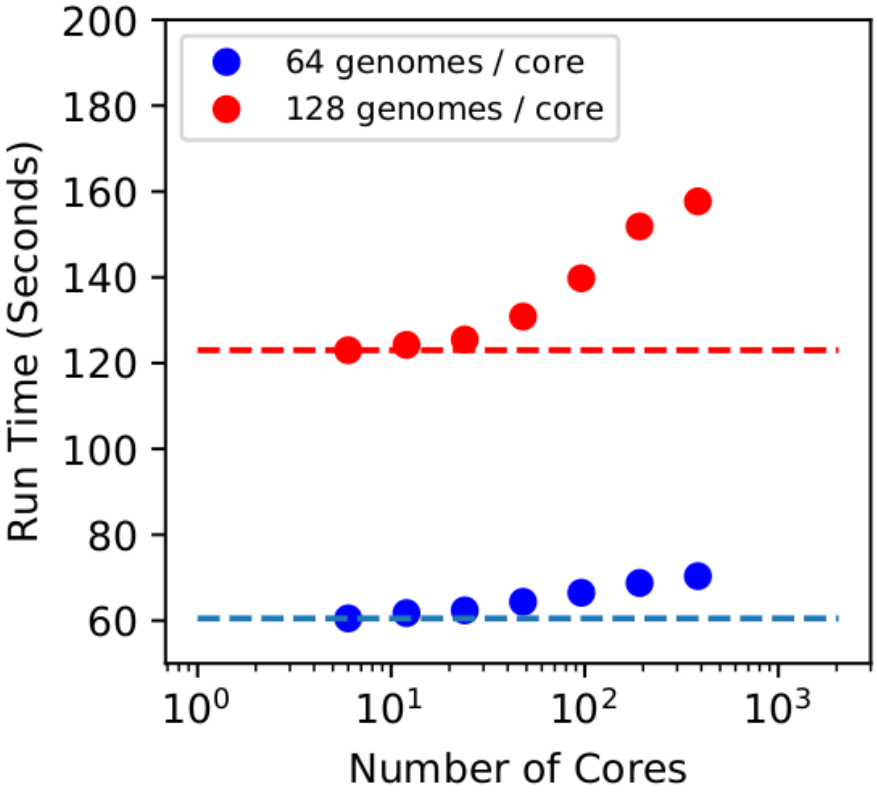
Suggestion: Run with at least 80% efficiency

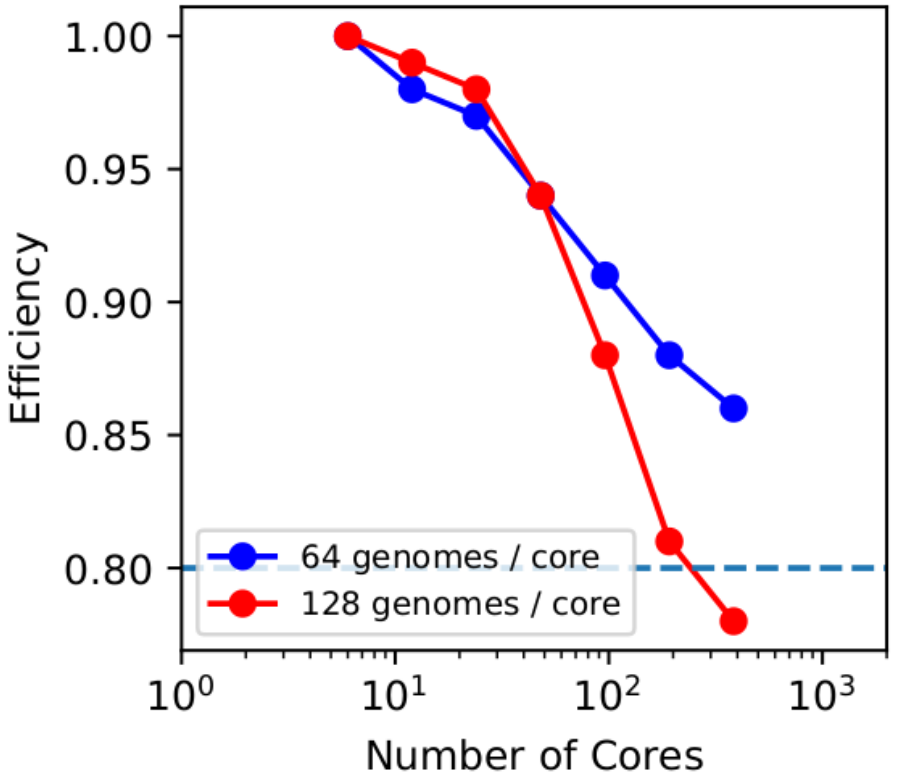
Sample Weak Scaling Data

Vary problem size in proportion to core count.

Weak-Scaling Data for Genomics Study			
64 Genomes per Core			
Cores	Measured Time (seconds)	Ideal Time (seconds)	Efficiency
6.0	60.5	60.5	1.0
12.0	61.7346938776		0.98
24.0	62.3711340206		0.97
48.0	64.3617021277		0.94
96.0	66.4835164835		0.91
192.0	68.75		0.88
384.0	70.3488372093		0.86
128 Genomes per Core			
6.0	123.0	123.0	1.0
12.0	124.242424242		0.99
24.0	125.510204082		0.98
48.0	130.85106383		0.94
96.0	139.772727273		0.88
192.0	151.851851852		0.81
384.0	157.692307692		0.78

Weak Scaling Plots





In-Class Exercise: Analyzing a Sample Program

- Examine hpc_profiling/scale.f90
- 2-D, iterative smoothing operation with nearest neighbor communication
- Grid dimensions (nx,ny) and number of iterations (nt) can be controlled via command-line arguments.
  - mpiexec –np 16 ./prog.mpi –nx 128 –ny 256 –nt 100
- We are going to run (but not edit) this program. We will:
  - Generate strong and weak scaling data and plot the results.

Notes

- Reusable tools provided:
  - Scaling Bash script
  - Python plotting program
- Submitting the job:
  - sbatch job.sh
- Code outputs: n_{cores}, time, nxglobal, nyglobal, niter
- A sample Python analysis script is provided.
- Let's create some scaling data together…