Virtual Summer School: profiling

Erik Spence

SciNet HPC Consortium

3 September 2021

Material for this class

The slides and code for this class can be found here:

`https://scinet.courses/584`

All the material for the 2021 Virtual Summer School can be found here:

`https://scinet.courses/573`

Make sure that you confirm your attendance before the end of class.

Details about this course

To get credit for this course:

- You need to attend 2 out of 3 sessions.
    - Each session is 1.5 hours,
    - Sessions are 12:30 - 2:00pm, Monday, Wednesday, Friday, August 30 - September 3.
    - To demonstrate attendance, you must take the attendance test each class.
- You must submit the assignment.

Ask questions!

Today's class

The purpose of this second set of material is to introduce you to profiling. We will cover the following topics:

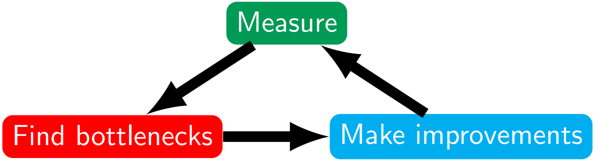
- Profiling in general,
- time,
- gprof,
- valgrind.

Note that this class will be exclusively about compiled languages (C, C++, Fortran). We will not be looking at profiling interpreted languages (Python, R).

Profiling

What is profiling?

- Profiling is measuring where your program is spending its resources.
- Like debuggers for debugging, profilers are evidence-based methods for finding performance problems.
- You can't improve what you don't measure.



Profiling, continued

The strategy for performance profiling:

- Where in the program is time being spent?
- Focus on the "expensive" parts of the code. Don't waste time optimizing parts that don't matter.
- Find the bottlenecks.

There are two main approaches to profiling:

- Tracing versus sampling,
- Instrumented versus instrumentation-free.

We will go over these, as well as whole-program profiling.

Built-in utilities

Let's begin by looking at some utilities provided by your computer's operating system.

- time
- top, ps, htop, Iotop,
- vmstat, free,
- lsof, iostat,
- tcpdump, iptraf, iftop,
- and others.

These are an easy place to get some crude performance numbers for your program.

Timing the whole program

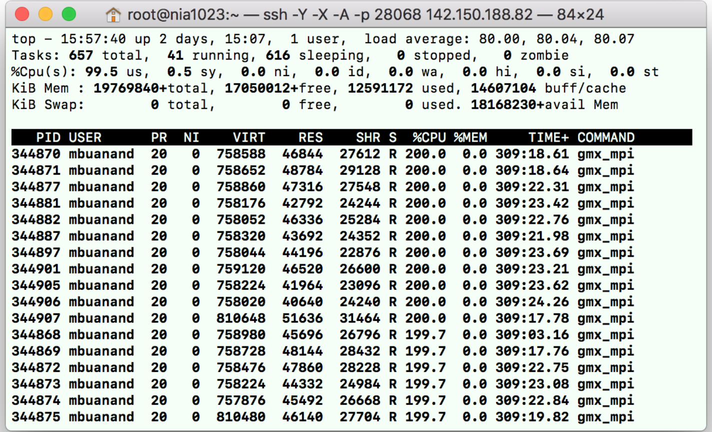
The simplest thing you can do is time the whole program, using the "time" command.

- Easy; can run on any command (program).
- For a serial program: real = usr + sys
- For parallel programs, ideally user = nprocs × real
- Can run on tests to identify performance *regressions*.

```
ejspence@mycomp ~>
ejspence@mycomp ~> time ./myProgram
.   .   .
[ your program output ]
.   .   .
real 0m2.448s   <-- Elapsed "walltime"
user 0m2.383s   <-- Actual user time
sys  0m0.027s   <-- System time:  Disk, I/O, ...
ejspence@mycomp ~>
```

A large system time can sometimes indicate opportunties for improvement.

Watching a program run



```
top - 15:57:40 up 2 days, 15:07,  1 user,  load average: 80.00, 80.04, 80.07
Tasks: 657 total,  41 running, 616 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.5 us,  0.5 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 19769840+total, 17050012+free, 12591172 used, 14607104 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 18168230+avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
344870 mbuanand  20   0  758588  46844  27612 R 200.0  0.0 309:18.61 gmx_mpi
344871 mbuanand  20   0  758652  48784  29128 R 200.0  0.0 309:18.64 gmx_mpi
344877 mbuanand  20   0  758860  47316  27548 R 200.0  0.0 309:22.31 gmx_mpi
344881 mbuanand  20   0  758176  42792  24244 R 200.0  0.0 309:23.42 gmx_mpi
344882 mbuanand  20   0  758052  46336  25284 R 200.0  0.0 309:22.76 gmx_mpi
344887 mbuanand  20   0  758320  43692  24352 R 200.0  0.0 309:21.98 gmx_mpi
344897 mbuanand  20   0  758044  44196  22876 R 200.0  0.0 309:23.69 gmx_mpi
344901 mbuanand  20   0  759120  46520  26600 R 200.0  0.0 309:23.21 gmx_mpi
344905 mbuanand  20   0  758224  41964  23096 R 200.0  0.0 309:23.62 gmx_mpi
344906 mbuanand  20   0  758020  40640  24240 R 200.0  0.0 309:24.26 gmx_mpi
344907 mbuanand  20   0  810648  51636  31464 R 200.0  0.0 309:17.78 gmx_mpi
344868 mbuanand  20   0  758980  45696  26796 R 199.7  0.0 309:03.16 gmx_mpi
344869 mbuanand  20   0  758728  48144  28432 R 199.7  0.0 309:17.76 gmx_mpi
344872 mbuanand  20   0  758476  47860  28228 R 199.7  0.0 309:22.75 gmx_mpi
344873 mbuanand  20   0  758224  44332  24984 R 199.7  0.0 309:23.08 gmx_mpi
344874 mbuanand  20   0  757876  45492  26668 R 199.7  0.0 309:22.84 gmx_mpi
344875 mbuanand  20   0  810480  46140  27704 R 199.7  0.0 309:19.82 gmx_mpi
```

You can use "top" to watch your code run, but it's not very *efficient*!

Instrumenting regions of code

It's more efficient to instrument your code.

- This means putting measurement tools directly into your source code.
- Simple, but incredibly useful.
- Can trivially see if changes make things better or worse.

```
/* simple timer definitions */
void tick(struct timeval *t) {
        gettimeofday(t, NULL);
}


/* returns time in seconds from now to time
     described by t */
double tock(struct timeval *t) {
        struct timeval now;
        gettimeofday(&now, NULL);
        return (double)(now.tv_sec - t->tv_sec) +
    ((double)(now.tv_usec - t->tv_usec)/1000000.);
}
```

```
#include <sys/time.h>
struct timeval init, calc, io;
double inittime, calctime, iotime;

    /*... */

tick(&init);
/* do initialization */
inittime = tock(&init);

tick(&calc);
/* do big computation */
calctime = tock(&calc);

tick(&io);
/* do IO */
iotime = tock(&io);


/* other timers ... */


printf("Timing␣summary:\n\tInit:␣%8.5f␣sec\n\tCalc:␣
     %8.5f␣sec\n\tI/O␣:␣%8.5f␣sec\n",
  inittime, calctime, iotime);
```

Instrumenting regions of code, example

Simple example:
matrix-vector multiply:

- Initializes data, does multiplication, saves the result.
- We'll examine where it spends its time, and try to speed it up.
- It will give us options for how to better access the data, and output the data.

```
/* initialize data */
tick(&init);
gettimeofday(&t, NULL);
seed = (unsigned int) t.tv_sec;

for (int i=0; i<size; i++) {
  x[i] = (double)rand_r(&seed)/RAND_MAX
           ;
  y[i] = 0.;
}

if (transpose) {
  for (int i=0; i<sizel i++) {
    for (int j=0; j<size; j++) {
      a[i][j] = (double)(rand_r(&seed))
                /RAND_MAX;
    }
  }
} else {
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      a[i][j] = (double)(rand_r(&seed))
                /RAND_MAX;
    }
  }
}

inittime = tock(&init);
```

```

tick(&calc);
if (transpose) {
  for (int i=0; i<size; i++) {
    for (int j=0; j<size; j++) {
      y[i] += a[i][j]*x[j];
    }
  }
} else {
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      y[i] += a[i][j]*x[j];
    }
  }
}

calctime = tock(&calc);
```

Matrix-vector multiply

Instrumenting the code didn't take very long.

- We can now get an overview of the time spent easily, because we instrumented our code (∼12 lines!).
- As we can see, there's a huge I/O (file Input/Output) bottleneck.

```
ejspence@mycomp ~>
ejspence@mycomp ~> mvm --matsize=2500
.   .   .
Timing Summary:
Init:  0.00952 sec
Calc:  0.06638 sec
I/O : 5.07121 sec
ejspence@mycomp ~>
```

Matrix-vector multiply: I/O

It's not a huge surprise that things are going slowly:

- I/O is being done in ASCII!
- The code is looping over the data, converting to string, writing to output.
- There are ≈ 6.252.500 write operations!

```
// ASCII output
out = fopen("Mat-vec.asc","w");
fprintf(out,"%d\n", size);

for (int i=0; i<size; i++)
  fprintf(out, "%f␣", x[i]);
fprintf(out,"\n",out);

for (int i=0; i<size; i++)
        fprintf(out, "%f␣", y[i]);
fprintf(out,"\n",out);

for (int i=0; i<size; i++) {
  for (int j=0; j<size; j++) {
    fprintf(out, "%f␣", a[i][j]);
  }
  fprintf(out,"\n",out);
}
fclose(out);
```

Let's try a **--binary** option:

```
// BINARY output
out = fopen("Mat-vec.bin","wb");
fwrite(&size,        sizeof(int),
          1,      out);
fwrite(    x,      sizeof(float),
       size,      out);
fwrite(    y,      sizeof(float),
       size,      out);
fwrite(&(a[0][0]), sizeof(float),
      size*size, out);
fclose(out);
```

Well, the code is shorter ...

Matrix-vector multiply: I/O, continued

But not just shorter!

- Much much (36×) faster!
- The file is 4× smaller.
- It's still slow, but file I/O is always going to be slower than a calculation (ie. multiplication).

```
ejspence@mycomp ~>
ejspence@mycomp ~> mvm --matsize=2500
Timing Summary:
Init:  0.00952 sec
Calc:  0.06638 sec
I/O : 5.07121 sec
ejspence@mycomp ~>
ejspence@mycomp ~> mvm --matsize=2500 --binary
Timing Summary:
Init:  0.00976 sec
Calc:  0.06695 sec
I/O : 0.14218 sec
ejspence@mycomp ~>
ejspence@mycomp ~> du -h Mat-vec.*
89M       Mat-vec.asc
20M       Mat-vec.bin
ejspence@mycomp ~>
```

Performance and File I/O

Lesson about performance and HPC: always always use BINARY formats for I/O!

- There is no conversion (numbers to strings) needed (reduces CPU cycles).
- The file sizes are usually smaller (reduces actual file IOPs).
- There is no precision lost due to conversion.
- There are even more advantages if you use a standard storage format (netCDF, or HDF5).

Don't dump lots of small files; it wastes time. Instead, bundle things whenever possible.

Sampling for profiling

Rather than instrumenting the code, a different approach is to sample the code while it's running.

- This allows us to get finer-grained (more detailed) information about where time is being spent.
- We can't instrument every single line of the code, especially for large codes.
- Compilers have built-in tools for *sampling* execution paths.

How does sampling work?

- As the program executes, every so often ($\sim$100ms) a timer goes, off, and the current location of execution is recorded.
- This shows where time is being spent in the code.

Sampling

Sampling is useful, but not perfect.

- Advantages:
  - Very low overhead,
  - Easy to implement,
  - No extra instrumentation.
- Disadvantages:
  - It doesn't tell us *why* the code was spending time where it does.
  - Statistics: we have to run long enough to have a good "sample size".

The **gprof** tool is a good sampling-based code profiling tool.

- Free, open-source.
- Common on Unix-type systems. Available on all SciNet systems.
- Easy to script, put into batch jobs.
- Low overhead.
- As with graphical debuggers, there are versions with GUIs as well.

The **gprof** tool is a quick-and-easy way to implement sampling in your code.

gprof for sampling

Specific compilation flags need to be invoked to use **gprof**:

- **-pg** turns on profiling,
- **-g** activate debugging symbols (optional, but more info).

```
ejspence@mycomp ~>
ejspence@mycomp ~> gcc -O3 -pg -g mat-vec-mult.c --std=c++11
ejspence@mycomp ~> icc -O3 -pg -g mat-vec-mult.c -c++11
ejspence@mycomp ~>
ejspence@mycomp ~> ./mvm-profile --matsize=2500
.  .  .
[ output ]
ejspence@mycomp ~> ls
Makefile Mat-vec.asc gmon.out mat-vec-mult.c mvm-profile
ejspence@mycomp ~>
```

During execution nothing has to be actually done, at the end there is a new file named "gmon.out" containing the information about the samples collected during runtime.

gprof, examining the results

This gives the time used by each function. This is usually handy, but not so useful in this toy problem.

Adding **--line** gives profiling by line. This can make things easier to read.

Monolithic code vs Modular code... another good reason in favour of modularity!

```
$ gprof mvm-profile gmon.out

Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.24      0.41     0.41       3       0.00            main
  0.00      0.41     0.00       3       0.00    0.00    tick
  0.00      0.41     0.00       3       0.00    0.00    tock
  0.00      0.41     0.00       2       0.00    0.00    alloc1d
  0.00      0.41     0.00       2       0.00    0.00    free1d
```

```
$ gprof --line mvm-profile gmon.out | more

Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
68.46      0.28    0.28     main  (mat-vec-mult.c:82 @ 401
14.67      0.34    0.06     main  (mat-vec-mult.c:113 @ 40
 7.33      0.37    0.03     main  (mat-vec-mult.c:63 @ 401
 4.89      0.39    0.02     main  (mat-vec-mult.c:112 @ 40
 4.89      0.41    0.02     main  (mat-vec-mult.c:113 @ 40
 0.00      0.41    0.00       3      0.00  0.00  tick (mat-vec-mult.c:159 @ 40
 0.00      0.41    0.00       3      0.00  0.00  tock (mat-vec-mult.c:164 @ 40
 0.00      0.41    0.00       2      0.00  0.00  alloc1d (mat-vec-mult.c:152 @
 0.00      0.41    0.00       2      0.00  0.00  free1d (mat-vec-mult.c:171 @
```

Analyzing the results

```
$ gprof --line mvm-profile gmon.out | more

Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call   name
68.46      0.28     0.28     main  (mat-vec-mult.c:82 @ 401
14.67      0.34     0.06     main  (mat-vec-mult.c:113 @ 40
 7.33      0.37     0.03     main  (mat-vec-mult.c:63 @ 401
 4.89      0.39     0.02     main  (mat-vec-mult.c:112 @ 40
 4.89      0.41     0.02     main  (mat-vec-mult.c:113 @ 40
```

So what do we see?

- The code is spending most of the time deep in loops:
- # 1: multiplication ... **line 82**
- # 2: I/O (ASCII output) ... **line 113**

```
 80    for (int j=0; j<size; j++) {
 81      for (int i=0; i<size; i++) {
 82        y[i] += a[i][j]*x[j];
 83      }
 84    }
```

. . .

```
 99  // ASCII output
100  out = fopen("Mat-vec.asc","w");
101  fprintf(out,"%d\n", size);
102
103  for (int i=0; i<size; i++)
104    fprintf(out, "%f␣", x[i]);
105  fprintf(out,"\n",out);
106
107  for (int i=0; i<size; i++)
108          fprintf(out, "%f␣", y[i])
                     ;
109  fprintf(out,"\n",out);
110
111  for (int i=0; i<size; i++) {
112    for (int j=0; j<size; j++) {
113      fprintf(out, "%f␣", a[i][j]);
114    }
115    fprintf(out,"\n",out);
116  }
117  fclose(out);
```

Memory Profiling

Most profilers use time as a performance *metric*, but what about *memory*? That's also a valid way to judge the code's performance.

There are many memory profilers available. In particular, our old friend Valgrind:

- Massif: Memory Heap Profiler
  - `valgrind --tool=massif ./mycode`
  - `ms_print massif.out`
- Cachegrind: Cache Profiler
  - `valgrind --tool=cachegrind ./mycode`
  - Kcachegrind (gui frontend for cachegrind)

`http://valgrind.org/`

Memory Profiling: Valgrind Massif

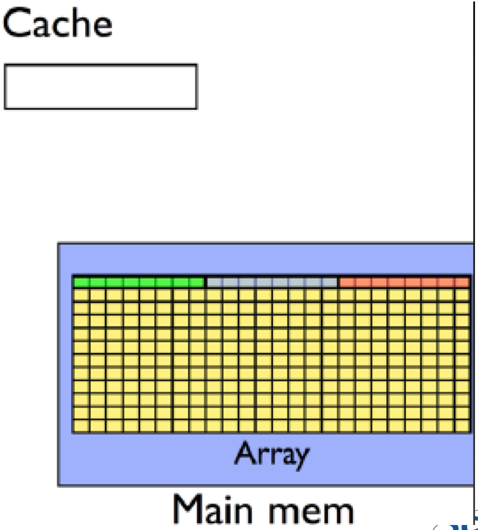
Example of output from `ms_print`, showing heap memory usage.

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 11 17,558,376,865      108,721,536      108,079,702       641,834            0
 12 18,730,053,265      108,746,848      108,104,510       642,338            0
 13 19,748,755,982      108,742,200      108,099,974       642,226            0
 14 21,351,204,796      108,745,520      108,103,214       642,306            0
 15 22,575,905,502      108,742,200      108,099,974       642,226            0
 16 24,344,627,331      108,742,200      108,099,974       642,226            0
 17 25,780,057,465      108,742,200      108,099,974       642,226            0
 18 27,215,452,841      108,742,200      108,099,974       642,226            0
99.41% (108,099,974B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->55.61% (60,466,176B) 0x873A8A: BlockMat::setup() (in navierstokes3Dthermallyperfect.5)
| ->55.61% (60,466,176B) 0x47A0F5: Hexa_NKS_Solver<State>::allocate() (NKS.h:192)
|   ->55.61% (60,466,176B) 0x477796: int HexaSolver<State>(char*, int) (HexaSolver.h:150)
|     ->55.61% (60,466,176B) 0x476A9F: main (NavierStokes3DThermallyPerfect.cc:226)
|
->10.07% (10,948,608B) 0x47A3B2: Hexa_NKS_Solver<State>::allocate() (NKS.h:186)
| ->10.07% (10,948,608B) 0x477796: int HexaSolver<State>(char*, int) (HexaSolver.h:150)
|   ->10.07% (10,948,608B) 0x476A9F: main (NavierStokes3DThermallyPerfect.cc:226)
|
->09.15% (9,953,280B) 0x47A390: Hexa_NKS_Solver<Statee>::allocate() (NKS.h:186)
| ->09.15% (9,953,280B) 0x477796: int HexaSolver<State>(char*, int) (HexaSolver.h:150)
|   ->09.15% (9,953,280B) 0x476A9F: main (NavierStokes3DThermallyPerfect.cc:226)
```

Cache Thrashing I

An easy problem to fall into is known as "cache thrashing".

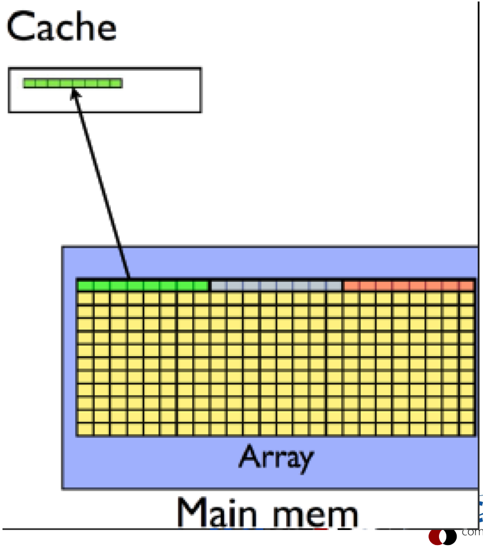
- Memory bandwith is key to getting good performance on modern systems.
- Main Memory is big and slow.
- The cache is small and fast.
- The cache saves recent memory accesses, one "line" of data at a time.



Cache Thrashing II

When accessing memory *in order*,

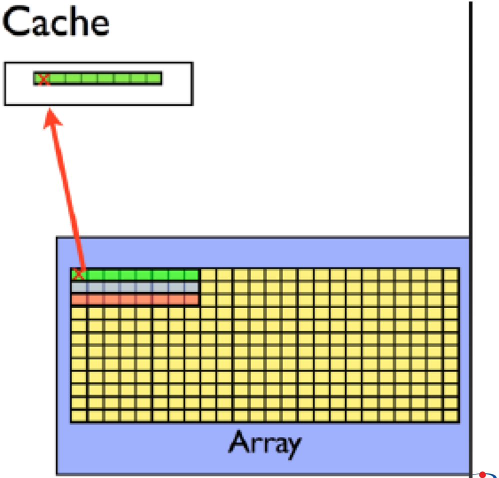
- only one memory access to (slow) main memory is needed for many data points.
- This is much faster than accessing memory multiple times.



Cache Thrashing III

When accessing memory *out of order*:

- a single, or a few, pieces of data are grabbed with each memory access,
- things slow down significantly.
- Each memory access is a new cache line (**cache miss**). Accessing main memory is slow.
- You can see $\sim 10\times$ slowdown in performance.



Cache Thrashing IV

Ok, so how do we keep from cache-thrashing? You need to know how your programming language stores memory.

- In C, a row-major language, the cache-friendly order is to make the last array index the most-quickly varying.
- The opposite is true of Fortran, a column-major language.
- You can see cache problems with **valgrind** + visualizer
- **valgrind --tool=cachegrind**
- The KDE tool **kcachegrind** is available for Windows, Linux and Mac OS X.

```
tick(&calc);
if (transpose) {
  // GOOD! ie. cache-friendly...
  for (int i=0; i<size; i++) {
    for (int j=0; j<size; j++) {
      y[i] += a[i][j]*x[j];
    }
  }
} else {
  // BAD!
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      y[i] += a[i][j]*x[j];
    }
  }
}

calctime = tock(&calc);
```

Cache Thrashing V

Checking our code once again, we can see that once cache thrashing is fixed, and assuming I/O can't be further improved, "Init" is now the bottleneck...

```
ejspence@mycomp ~>
ejspence@mycomp ~> ./mvm --matsize=2500 --transpose --binary
Timing Summary:
Init:  0.00947 sec
Calc:  0.00811 sec
I/O : 0.14881 sec
ejspence@mycomp ~>
```

Other Profiling Tools

There are many other profiling tools out there.

- Scalasca
- Open SpeedShop
- TAU Performance System
- HPC Tool Kit
- Arm MAP (Forge)
- Xcode (OS X)
- Nvidia Profiler (nvprof)

The Intel Parallel Studio XE has many useful tools:

- Intel VTune Amplifier XE (performance)
- Intel Inspector XE (memory)
- Intel Advisor XE (vector/thread)
- Intel Trace Analyzer and Collector (MPI)

There's a great variety of profiling tools available. The Intel Parallel Studio, and Arm Forge are particularly good.

Arm Forge MAP

The Map tool, which comes with Arm Forge (DDT), is also very powerful.

- The Arm Forge suite is made available through the DDT module.
- Performance reports are generated using the ”perf report ...” command.
- This will generate .txt, .html and .map files.

```
ejspence@teach01 ~>
ejspence@teach01 ~> module load gcc/7.3.0
ejspence@teach01 ~> module load openmpi/3.1.1
ejspence@teach01 ~> module load ddt/20.1.3
ejspence@teach01 ~>
ejspence@teach01 ~> perf report mpirun -np 4 ./mycode
ejspence@teach01 ~>
```

Map can also be used through the interactive client-server setup which we saw last class.

Profiling – Summary

A review of what we've discussed today:

- There are two main approaches to profiling: tracing vs sampling.
- Tracing:
  - Put timers in the code in/around important sections, find out where time is being spent.
  - If something important changes, you'll know in what section.
- Sampling:
  - Sample the location of the program in the code at regular intervals.
  - `gprof` is easy to use and excellent at finding where the time is spent.
  - Know the 'expensive' parts of your code and spend your programming time accordingly.
- `valgrind` is good for all things memory; performance, cache, and usage.
- Arm map is a great tool, if you have it available use it!
- As with debugging, the usual advices applies: write less code (ie. use libraries), write modular code, follow best-practices for file I/O, ...