PHY1610H - Scientific Computing: Profiling

Ramses van Zon and Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

February, 2021

Physics
UNIVERSITY OF TORONTO

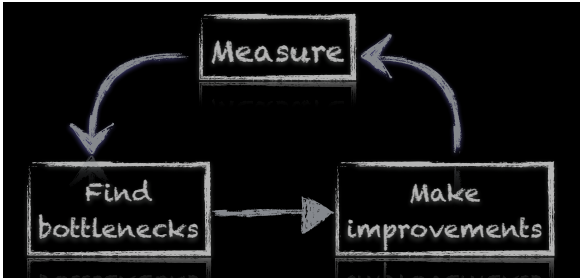
Lecture 09 Topics

- Profiling
  - Techniques: Tracing vs Sampling
- Tools
  - time
  - gprof
  - valgrind
  - Intel & Allinea

SciNet Physics UNIVERSITY OF TORONTO

Profiling

- is a form of *dynamic program analysis* that **measures**, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.
- Like debuggers for finding bugs, **profilers** are *evidence-based* methods to find performance problems.
- Most commonly, profiling information serves to aid program optimization.
- Can not improve what we don't measure.



Profiling

- Where in the program is time being spent?
- Find and focus in the "expensive" parts.
- Don't wate time optimizing parts that don't matter.
- Find bottlenecks.

```
  case SIM_PROJECTILE:
    ymin = xmin = 0.;
    ymax = xmax = 1.;
    dx = (xmax-xmin)/npts;
    dy = (ymax-ymin)/npts;
    init_domain(&d, npts, npts, KL_NGUARD,
         xmin, ymin, xmax, ymax);
    projectile_initvalues(&d, psize, pdens,
         pvel);
    outputvar = DENSVAR;
    break;
}
```

```
/* apply Boundary Conditions and make thermodynamically
     consistant */
bcs[0] = xbc;  bcs[1] = xbc;
bcs[2] = ybc;  bcs[4] = ybc;
apply_all_bcs(&d, bcs);
domain_backward_dp_eos(&d);
domain_ener_internal_to_tot(&d);

/* main loop */
tick(&tt);
if (output) domain_plot(&d);
printf("Step␣\t␣dt␣\t␣time␣\n");
for (time=0.,step=0; step < nsteps; step++, time+=2.+dt) {
  printf("%d␣\t␣%g␣\t␣%g␣\n", step, dt, time);
  if (output && ((step % outevery) == 0) ) {
    sprintf(ppmfilename, "dens_test_%d.ppm", outnum);
    sprintf(binfilename, "dens_test_%d.bin", outnum);
    sprintf(h5filename, "dens_test_%d.h5", outnum);
    sprintf(ncdffilename, "dens_test_%d.nc", outnum);
    domain_output_ppm(&d, outputvar, ppmfilename);
    domain_output_bin(&d, binfilename);
    domain_output_hdf5(&d, h5filename);
    domain_output_netcdf(&d, ncdffilename);
    domain_plot(&d);
    outnum++;
  }

  kl_timestep_xy(&d, bcs, dt);
  apply_all_bcs(&d, bcs);

  kl_tiemstep_yx(&d, bcs, dt);
  apply_all_bcs(&d, bcs);
}
tock(&tt);
```

Profiling

Two main approaches for Profiling

- Tracing vs. Sampling
- Instrumenting vs. Instrumentation-Free

```
/* apply Boundary Conditions and make thermodynamically
      consistant */
bcs[0] = xbc;  bcs[1] = xbc;
bcs[2] = ybc;  bcs[4] = ybc;
apply_all_bcs(&d, bcs);
domain_backward_dp_eos(&d);
domain_ener_internal_to_tot(&d);

/* main loop */
tick(&tt);
if (output) domain_plot(&d);
printf("Step␣\t␣dt␣\t␣time␣\n");
for (time=0.,step=0; step < nsteps; step++, time+=2.+dt) {
  printf("%d␣\t␣%g␣\t␣%g␣\n", step, dt, time);
  if (output && ((step % outevery) == 0) ) {
    sprintf(ppmfilename, "dens_test_%d.ppm", outnum);
    sprintf(binfilename, "dens_test_%d.bin", outnum);
    sprintf(h5filename, "dens_test_%d.h5", outnum);
    sprintf(ncdffilename, "dens_test_%d.nc", outnum);
    domain_output_ppm(&d, outputvar, ppmfilename);
    domain_output_bin(&d, binfilename);
    domain_output_hdf5(&d, h5filename);
    domain_output_netcdf(&d, ncdffilename);
    domain_plot(&d);
    outnum++;
  }

  kl_timestep_xy(&d, bcs, dt);
  apply_all_bcs(&d, bcs);

  kl_tiemstep_yx(&d, bcs, dt);
  apply_all_bcs(&d, bcs);
}
tock(&tt);
```

OS Utilities: time, top, ...

Let's start by looking at some utilities provided by the OS...

- **`time`**
- **`top`**, **`ps`**, **`htop`**, **`lotop`**, ...
- ...
- vmstat, free, ...
- lsof, iostat, ...
- tcpdump, iptraf, iftop, ...
- ...

SciNet Physics UNIVERSITY OF TORONTO

Timing the whole program

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

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

SciNet Physics UNIVERSITY OF TORONTO

Watching a program run...

```
$ top
```

```
top - 21:56:45 up  5:56,  1 user,  load average: 5.55, 1.73, 0.88
Tasks: 234 total,   1 running, 233 sleeping,   0 stopped,   0 zombie
Cpu(s): 11.4%us, 36.2%sy,  0.0%ni, 52.2%id,  0.0%wa,  0.0%hi,  0.2%si,  0.0%st
Mem:  16410900k total,  1542768k used, 14868132k free,        0k buffers
Swap:        0k total,        0k used,        0k free,   294628k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  P COMMAND
22479 ljdursi   18   0  108m 4816 3212 S 98.5  0.0   1:04.81  6 gameoflife
22480 ljdursi   18   0  108m 4856 3260 S 98.5  0.0   1:04.85 13 gameoflife
22482 ljdursi   18   0  108m 4868 3276 S 98.5  0.0   1:04.83  2 gameoflife
22483 ljdursi   18   0  108m 4868 3276 S 98.5  0.0   1:04.82  8 gameoflife
22484 ljdursi   18   0  108m 4832 3232 S 98.5  0.0   1:04.80  9 gameoflife
22481 ljdursi   18   0  108m 4856 3256 S 98.2  0.0   1:04.81  3 gameoflife
22485 ljdursi   18   0  108m 4808 3208 S 98.2  0.0   1:04.80  4 gameoflife
22478 ljdursi   18   0  117m 5724 3268 D 69.6  0.0   0:46.07 15 gameoflife
 8042 root       0 -20 2235m 1.1g  16m S  2.3  6.8   0:30.59  8 mmfsd
10735 root      15   0  3792  452  372 S  1.3  0.0   0:16.80  0 cat
```

More **system time** than **user time** ⇒ not very *efficient*!

Instrumenting regions of code

- *Instrumenting* the code
- Simple, but incredibly useful
- Runs every time your code is run
- Can trivially see if changes make thigns better or worse

```c
/* 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.);
}
```

```c
#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 – an example *matrix-vector multiply*

- Simple example: matrix-vector multiply **You'd never do this though, cause...**
- Initializes data, does multiply, save result
- Look to see where it spends its time, speed it up.
- Options for how to access data, output 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);
```

```
/* do multiplication */

tick(&calc);
if (transpose) {
  # pragma omp parallel for default(
       none) shared(x,y,a,size)
  for (int i=0; i<size; i++) {
    for (int j=0; j<size; j++) {
      y[i] += a[i][j]*x[j];
    }
  }
} else {
  # pragma omp parallel for default(
       none) shared(x,y,a,size)
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      y[i] += a[i][j]*x[j];
    }
  }
}

calctime = tock(&calc);
```

SciNet Physics UNIVERSITY OF TORONTO

Matrix-vector multiply

- Can get an overview of the time spent easily, because we instrumented our code (∼12 lines!)
- I/O huge bottleneck.

```
$ mvm --matsize=2500
.  .  .
Timing Summary:
Init:  0.00952 sec
Calc:  0.06638 sec
I/O : 5.07121 sec
```

Matrix-vector multiply: IO

- Not surprise here...
- I/O being done in ASCII
- having to loop over data, convert to string, write to output.
- ≈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);
```

- shorter ...

Matrix-vector multiply: IO

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

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

About Performance and File I/O

Recall our lecture on File I/O...

* Always use BINARY formats for I/O!!!
  - no conversion needed (reduces CPU cycles)
  - usually smaller files (reduces actual file IOPs)
  - (no precission lost)
  - (even more advantages if you use a standard format –eg. netCDF or HDF5–)
* Don't dump into a lot of small files... instead bundle things when possible

SciNet Physics UNIVERSITY OF TORONTO

Sampling for Profiling

- Allow us to get finer-grained (more detailed) information about where time is being spent
- Can't instrument every single line
- Compilers have tools for *sampling* execution paths

Sampling...

- As the program executes, every so often (∼100ms) a timer goes, off, and the current location of execution is recorded
- Shows where time is being spent
- Caveats: how long the program takes to run, "number of samples", ...

```
/* 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);
```

Lines 7 / 18 / 223 / 9 / ...

```
/* do multiplication */

tick(&calc);
if (transpose) {
  # pragma omp parallel for default(
       none) shared(x,y,a,size)
  for (int i=0; i<size; i++) {
    for (int j=0; j<size; j++) {
      y[i] += a[i][j]*x[j];
    }
  }
} else {
  # pragma omp parallel for default(
       none) shared(x,y,a,size)
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      y[i] += a[i][j]*x[j];
    }
  }
}

calctime = tock(&calc);
```

Sampling...

* Advantages:
  - Very low overhead
  - No extra instrumentation
* Disadvantages:
  - Don't tell us *why* the code was there
  - Statistics: have to run long enough to have a good "sample size"

```
/* 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);
```

Lines 7 / 18 / 223 / 9 / ...

```
/* do multiplication */

tick(&calc);
if (transpose) {
  # pragma omp parallel for default(
        none) shared(x,y,a,size)
  for (int i=0; i<size; i++) {
    for (int j=0; j<size; j++) {
      y[i] += a[i][j]*x[j];
    }
  }
} else {
  # pragma omp parallel for default(
        none) shared(x,y,a,size)
  for (int j=0; j<size; j++) {
    for (int i=0; i<size; i++) {
      y[i] += a[i][j]*x[j];
    }
  }
}

calctime = tock(&calc);
```

`gprof` for sampling

* Compilation:

```
gcc -O3 -pg -g mat-vec-mult.c --std=c++14
icc -O3 -pg -g mat-vec-mult.c -c++14
```

* Execution:

```
$ ./mvm-profile --matsize=2500
.  .  .
[ output ]
$ ls
Makefile   Mat-vec.asc   gmon.out   mat-vec-mult.c
mvm-profile
```

`-pg` turns on profiling

`-g` activate debugging symbols (optional, but more info)

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

`gprof`: Examining the results, "gmon.out"

```
$ 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
  0.00      0.41     0.00        1     0.00     0.00    alloc2d
  0.00      0.41     0.00        1     0.00     0.00    free2d
  0.00      0.41     0.00        1     0.00     0.00    get_options
```

```
$ 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:102
 0.00      0.41     0.00        2     0.00     0.00  free1d (mat-vec-mult.c:117 @
 0.00      0.41     0.00        1     0.00     0.00  alloc2d (mat-vec-mult.c:130 @
```

Gives data by function usually handy, not so useful in this toy problem

`--line` gives profiling by line...

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

Analizing the results, ie. comparing to the source code...

```
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
```

- Code is spending most of the time deep in loops
- # 1: multipication ... **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);
```

gprof – Summary: pros/cons

- Exists (almost) everywhere
- Easy to script, put in batch jobs
- Low overhead
- As with graphical debbuggers, many nice graphical profiles exist as well

Memory Profiling

Most profilers use time as a possible *metric*, but what about *memory*?

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/`

SciNet Physics UNIVERSITY OF TORONTO

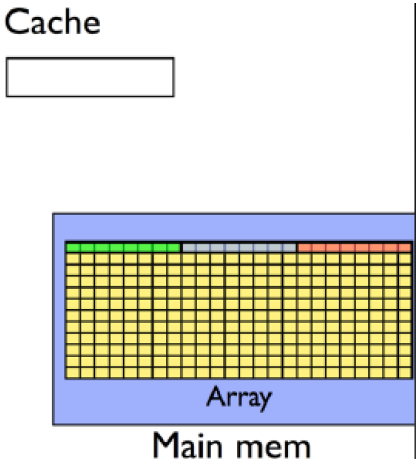
Memory Profiling: Valgrind Massif

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

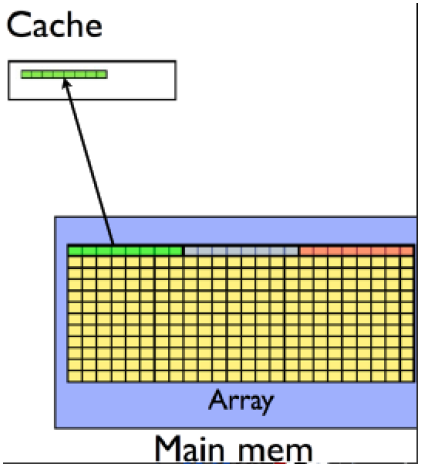
- Memory bandwith is key to getting good performance on modern systems
- Main Memory: big, slow
- Cache: small, fast
  - Saves recent accesses, a "line" of data at a time



SciNet Physics UNIVERSITY OF TORONTO

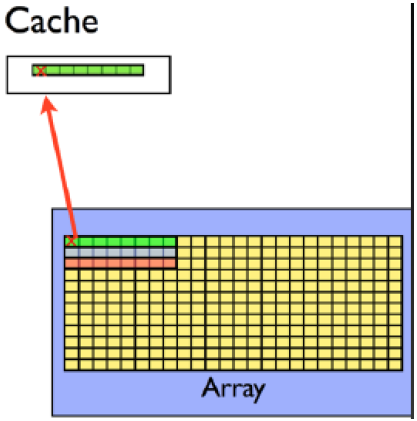
Cache Thrashing II

- When accessing memory *in order*, only one access to slow main memory for many data points
- Much faster



Cache Thrashing III

- When accessing memory *out of order*, much worse
- Each access is new cache line (**cache miss**) – slow access to main memory
- Can see $\sim$ **10**$\times$ slowdown



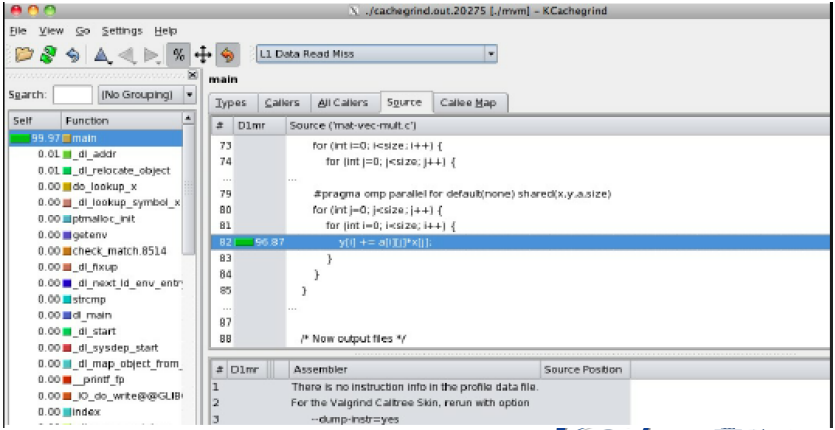
Cache Thrashing IV

- In C, cache-friendly order is to make the last index most quickly varying
- Can see cache problems with **valgrind** + visualizer
- **valgrind --tool=cachegrind**
- KDE tool **kcachegrind** available for Windows, Linux and Mac OS X

```
/* do multiplication */

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

```
$ module load valgrind
$ valgrind --tool=cachegrind ./mvm --matsize=2500
```



Cache Thrashing VI

- Once cache thrashing is fixed, assuming I/O can't be improved, "Init" is now the bottleneck...

```
$ ./mvm --matsize=2500 --transpose --binary
Timing Summary:
Init:  0.00947 sec
Calc:  0.00811 sec
I/O : 0.14881 sec
```

Other Profiling Tools

- Scalasca
- Open SpeedShop
- TAU Performance System
- HPC Tool Kit
- Allinea MAP (Forge)
- Intel (ITAC/Inspector/Advisor/Amplifier (VTune))
- Xcode (OS X)
- Nvidia Profiler (nvprof)

SciNet Physics UNIVERSITY OF TORONTO

Intel Parallel Studio XE

Applications

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

SciNet Physics UNIVERSITY OF TORONTO

Allinea MAP (Forge)

SciNet

```
module load intel intelmpi
module load ddt/20.1.3
```

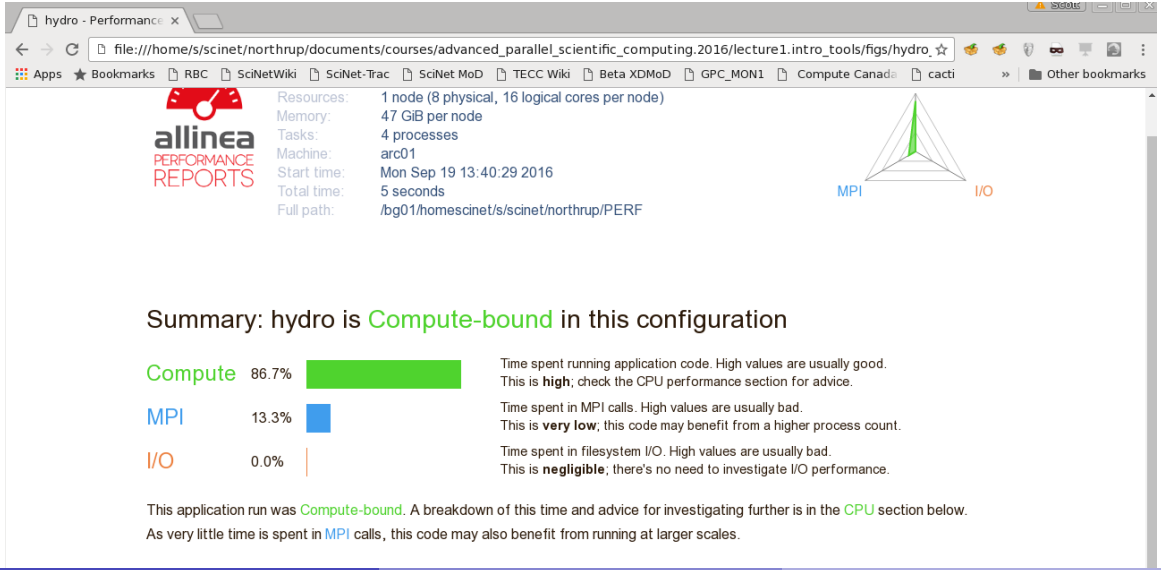
Performance Reports

- Compile with debugging on, ie **-g**
- **perf-report mpirun -np 4 ./mycode**
- Generates **.txt, .html, .map files**

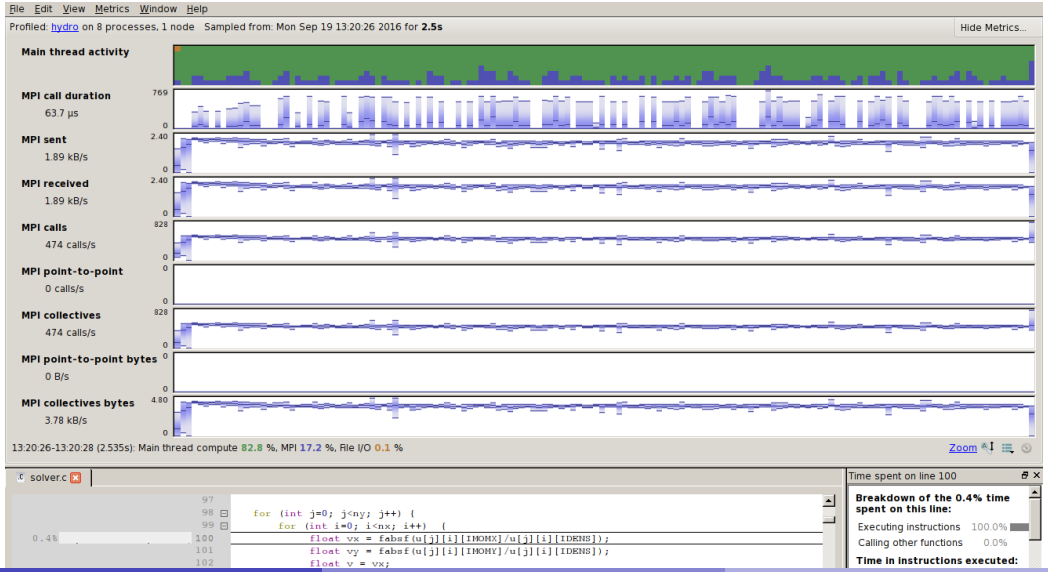
MAP

- Compile with debugging on, ie **-g**
- **map**

Allinea Performance Reports



Allinea MAP (Forge)



Profiling – Summary

- Two main approches: tracing vs sampling
- Put your own timers in the code in/around important sections, find out where time is being spent.
  - if something changes, know in what section
- **`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.
- Allinea map is a great tool, if you have it available use it!
- As with debugging, similar advices apply: write less code (ie. use libraries), write modular code, follow best-practices for file I/O, ...

SciNet Physics UNIVERSITY OF TORONTO