Debugging and Profiling

Scientific Computing Course, Feb 2013

Homework

- Questions about homework?
  - Diffusion Class
  - Tracer Class

Debugging

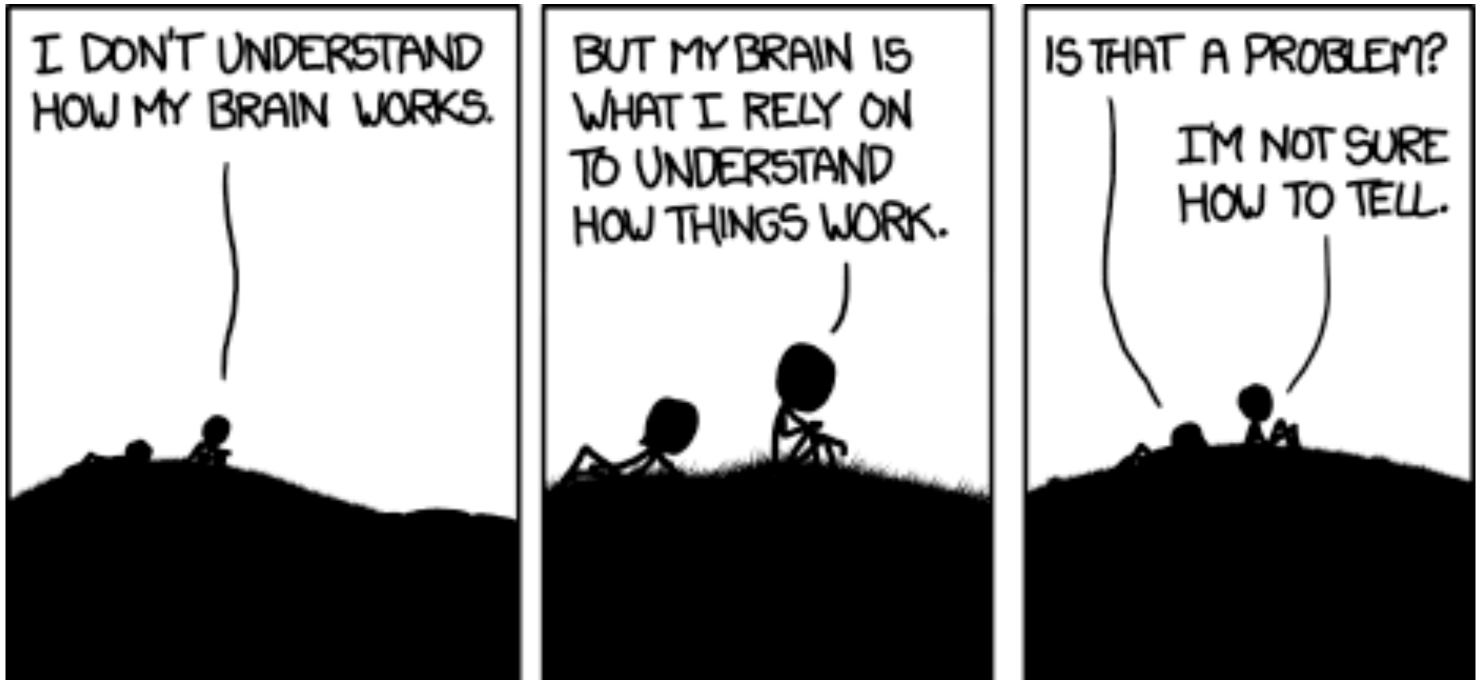
Scientific Computing Course, Feb 2013

Debugging

- All programs execute correctly.
- We just told it to do the wrong thing.
- Debugging is the art of reconciling *your* mental model of what the code "is" doing with what it is **actually** doing; then adjusting the code back to what you intended.
- This is a genuinely difficult task; you're effectively debugging your own thought process.

SciNet compute • calcul CANADA

Debugging



http://imgs.xkcd.com/comics/debugger.png

SciNet compute • calcul CANADA

Tips to avoid debugging,

or at least make it less of a soul-sucking time-sink

- Write better code
  - Every time you write a line and think "I'm pretty sure no one would pass a negative `n` into here, anyway", stop and insert a test - at **least**, `assert( n >= 0 );`
  - Practice modularity - no global variables, break things up into meaningful chunks that are self contained. Don't have to go hunting through multiple files to debug one routine.

Tips to avoid debugging,

or at least make it less of a soul-sucking time-sink

- Write straightforward code
    - Simple, clear; commented.
    - Straightforward logic; no “cute tricks”.
    - “Debugging is harder than coding. So if you were being as witty as you could possibly be while writing the code, you are by definition not smart enough to debug it.”

Tips to avoid debugging,
or at least make it less of a soul-sucking time-sink

- Don't write code
  - Every line of code you don't write is a line that will never turn out to be wrong.
  - Use (well-tested, well-maintained) other peoples' libraries when possible.
  - Re-use previous code where possible.
  - Don't re-invent the wheel. (DRY)

Tips to avoid debugging,

or at least make it less of a soul-sucking time-sink

- Write more tests
  - Exploit modularity in your code by writing tests for each module - can help find if something's gone awry
  - Find the bug as early as possible
  - If your tests aren't picking up the bug, can you write a simple additional test that **does** show the bad behaviour?
  - Keep that test in the test suite

Tips to avoid debugging,
or at least make it less of a soul-sucking time-sink

- Get outside help
  - *Your* blind spots are different from *their* blind spots.
  - Code review is shown time and time again to be the most effective way of finding bugs (bugs per person-hour) *and* to keep bugs out of code.
  - If you're working on a joint project, make code review before merge standard practice
  - Works particularly well for ~100 line-sized chunks

SciNet compute • calcul CANADA

Basic Debugging Workflow: (1) Reproducable Example

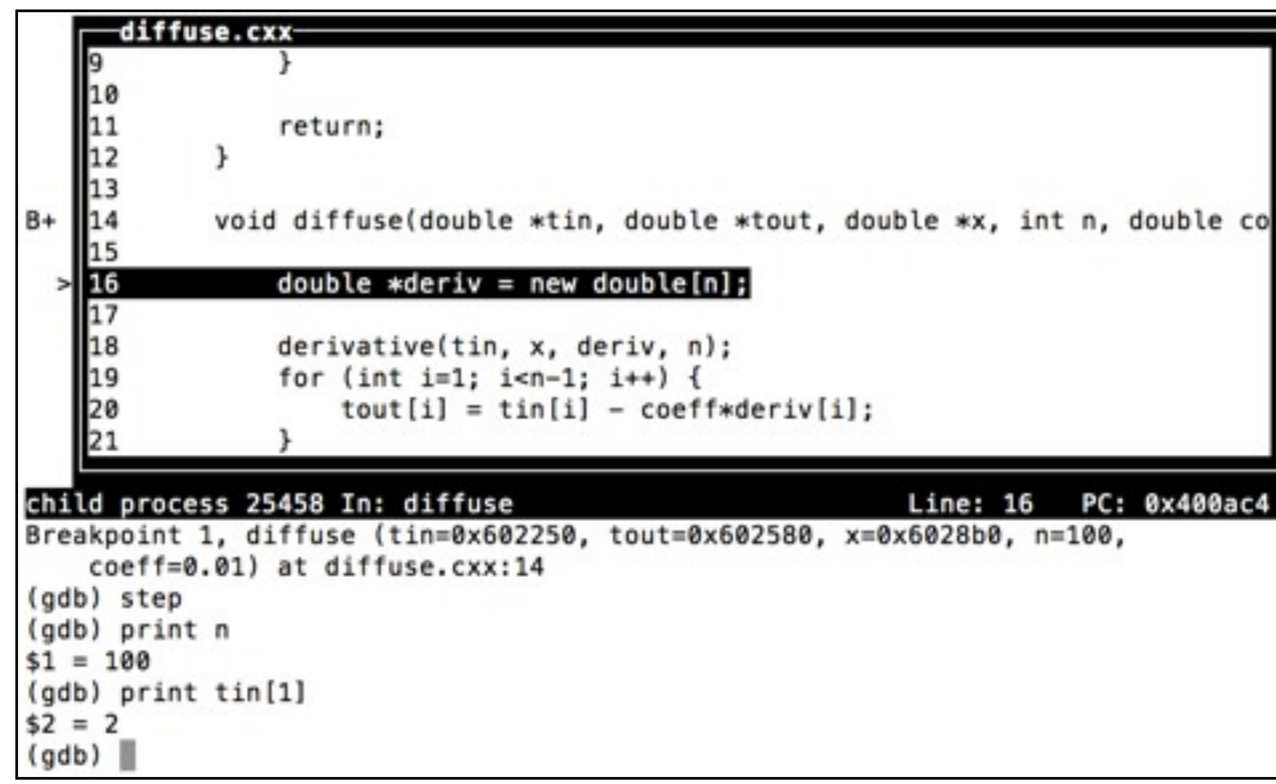
- As soon as you are convinced there is a real problem, job #1 is to create the simplest situation in which it repeatedly occurs.
- This is science: model, hypothesis, experiment, conclusion.
- Do ***not*** charge in, saying "I'm pretty sure it's in here! I'll just change this..." Now you've got two bugs.
- Try a smaller problem size, turning off different physical effects with options, etc, until you have a simple, fast, repeatable example.

Basic Debugging Workflow: (2) Narrow down the problem

- Again, this is science: model, hypothesis, experiment, conclusion.
- Try to narrow down in what module the bug is introduced.
- Unit tests: Maybe my diffusion operator doesn't work on non-monotone data. If that's the case, then this test should find it... No, that seems to be working fine.
- Absent clear evidence like the above, avoid the trap of "Oh, I'm *sure* it's not in there..."
- Integrated calculation: Write out intermediate results to a file, inspect them.

Tools to help you debug

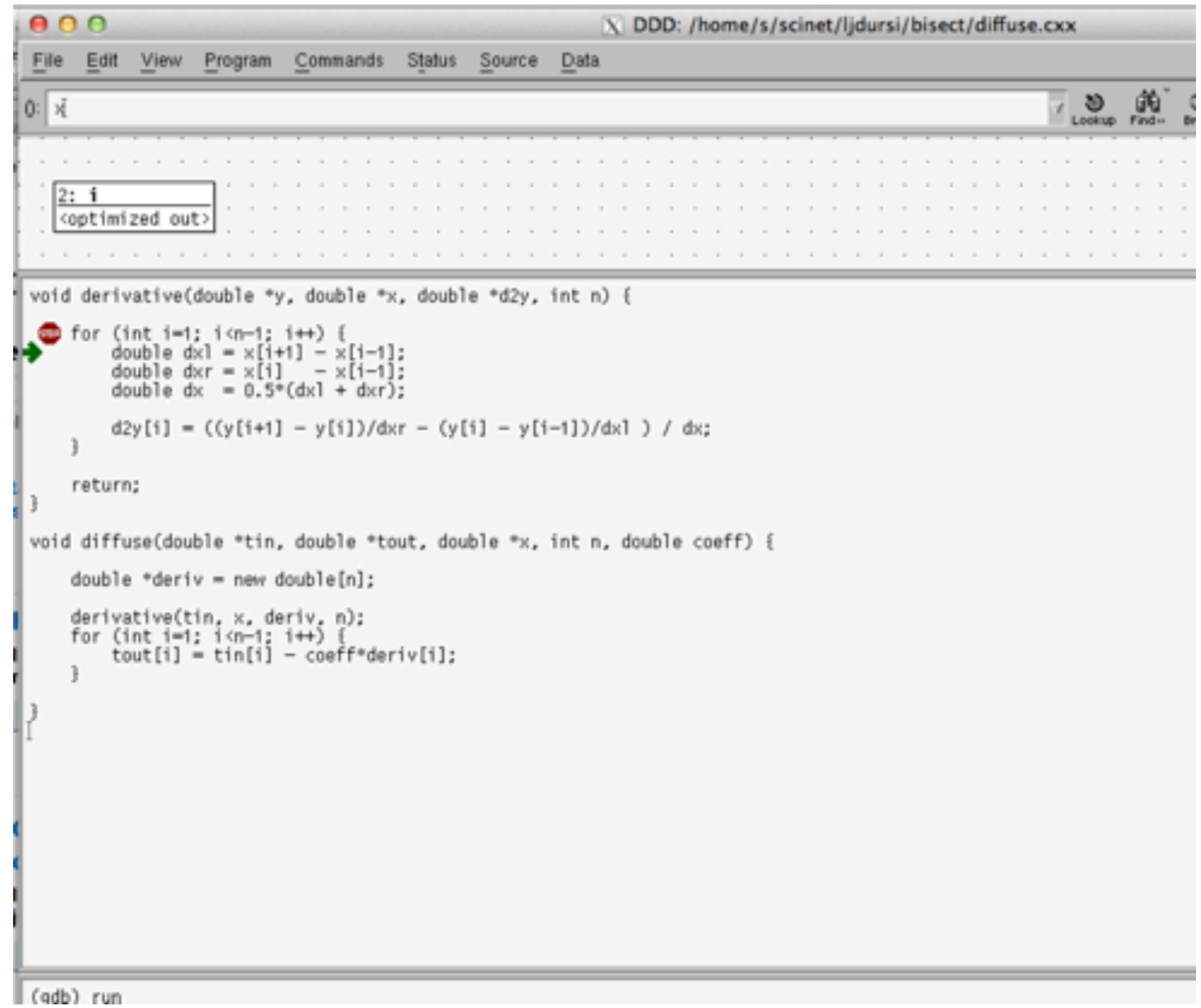
- Symbolic "debuggers"
- Allow you to step through the code, print variables - eg, see what code is really doing.
- To use this, more information needs to be stored in the executable than computer would generally need
- compile with –g flag



SciNet compute • calcul CANADA

Tools to help you debug

- Graphical and text-based
- Same basic functionality.
- Graphical is easier to use (can see more at once)
- Text often has advantage over network connection.
- Note "Optimized out"; sometimes advantageous to reduce optimization level of compilation while debugging. (–O0)



SciNet compute • calcul CANADA

Best possible case: core dump

- In general, more spectacular the failure, easier to debug
- Bugs that cause *slightly* wrong answers are most challenging, dangerous.
- Segmentation fault: trying to access illegal memory.
- Scientific code: often out-of-bounds array indices, or bad arguments to a function

```
gpc-f103n084-$ make
g++ -c -o tests.o -O2 -Wall -g tests.cxx
g++ -o tests -lm tests.o diffuse.o
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault
gpc-f103n084-$ ulimit -c unlimited
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
gpc-f103n084-$ ls -l core*
-rw------- 1 ljdursi scinet 483328 Feb  4 21:20 core.2586
gpc-f103n084-$ 
```

SciNet compute • calcul CANADA

Best possible case: core dump

- POSIX type systems will try to "dump core" (write contents of memory) on sufficiently spectacular failure.
- This is often turned off by user limits (copies of all of processes memory can be quite large).
- `ulimit -c unlimited` will allow these dump files.

```
gpc-f103n084-$ make
g++ -c -o tests.o -O2 -Wall -g tests.cxx
g++ -o tests -lm tests.o diffuse.o
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault
gpc-f103n084-$ ulimit -c unlimited
gpc-f103n084-$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)
gpc-f103n084-$ ls -l core*
-rw------- 1 ljdursi scinet 483328 Feb  4 21:20 core.2586
gpc-f103n084-$
```

SciNet compute • calcul CANADA

Best possible case: core dump

- With core file, and executable compiled with symbols, debugger will take you immediately to the point of seg fault.
- (Not *necessarily* point of the bug)
- gdb:
  `gdb executable corefile`

```
$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)

$ gdb tests core.27900
GNU gdb (GDB) 7.3.1
[...]
Core was generated by `./tests'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400635 in doConstTest (n=100) at tests.cxx:14
14          in[i] = 17.;
(gdb) where
#0  0x0000000000400635 in doConstTest (n=100) at tests.cxx:14
#1  0x00000000004008ac in main (argc=1, argv=0x7fffb0070b48)
(gdb) list
9       double *out = new double[n];
10      double *x = new double[n];
11      int j=0;
12
13      for (int i=0; j<n; i++) {
14          in[i] = 17.;
15      }
16
17      derivative(in, x, out, n);
18
(gdb) print i
$1 = 16894
(gdb) print n
$2 = 100
(gdb) quit
```

Best possible case: core dump

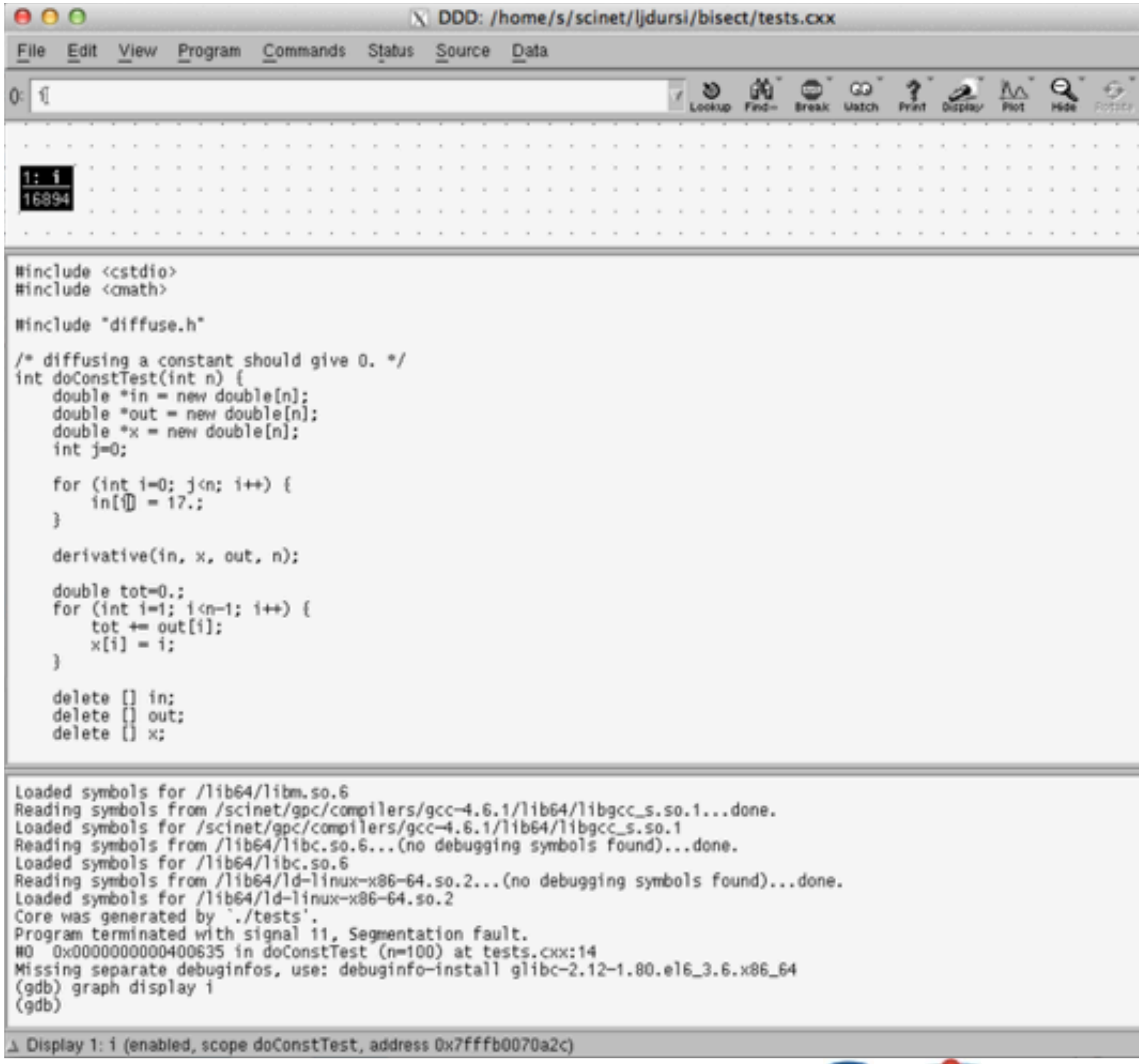
- Important commands in this context:
- `where` shows you where in the stack frame you are. main called doConstTest at line 69.
- `list` shows you lines of code above and below current cursor
- `print` - prints variables.

```
$ ./tests
Performing Constant Test...
Segmentation fault (core dumped)

$ gdb tests core.27900
GNU gdb (GDB) 7.3.1
[...]
Core was generated by `./tests'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400635 in doConstTest (n=100) at tests.cxx:14
14          in[i] = 17.;
(gdb) where
#0  0x0000000000400635 in doConstTest (n=100) at tests.cxx:14
#1  0x00000000004008ac in main (argc=1, argv=0x7fffb0070b48) at tests.cxx:69
(gdb) list
9       double *out = new double[n];
10      double *x = new double[n];
11      int j=0;
12
13      for (int i=0; j<n; i++) {
14          in[i] = 17.;
15      }
16
17      derivative(in, x, out, n);
18
(gdb) print i
$1 = 16894
(gdb) print n
$2 = 100
(gdb) quit
```

compute • calcul CANADA

Best possible case: core dump

- ddd - graphical debugger
- Same arguments (pretty common)
- `ddd executable corefile`
- Can click on or hover over variable to see value, etc
- Can even plot array values



SciNet compute • calcul CANADA

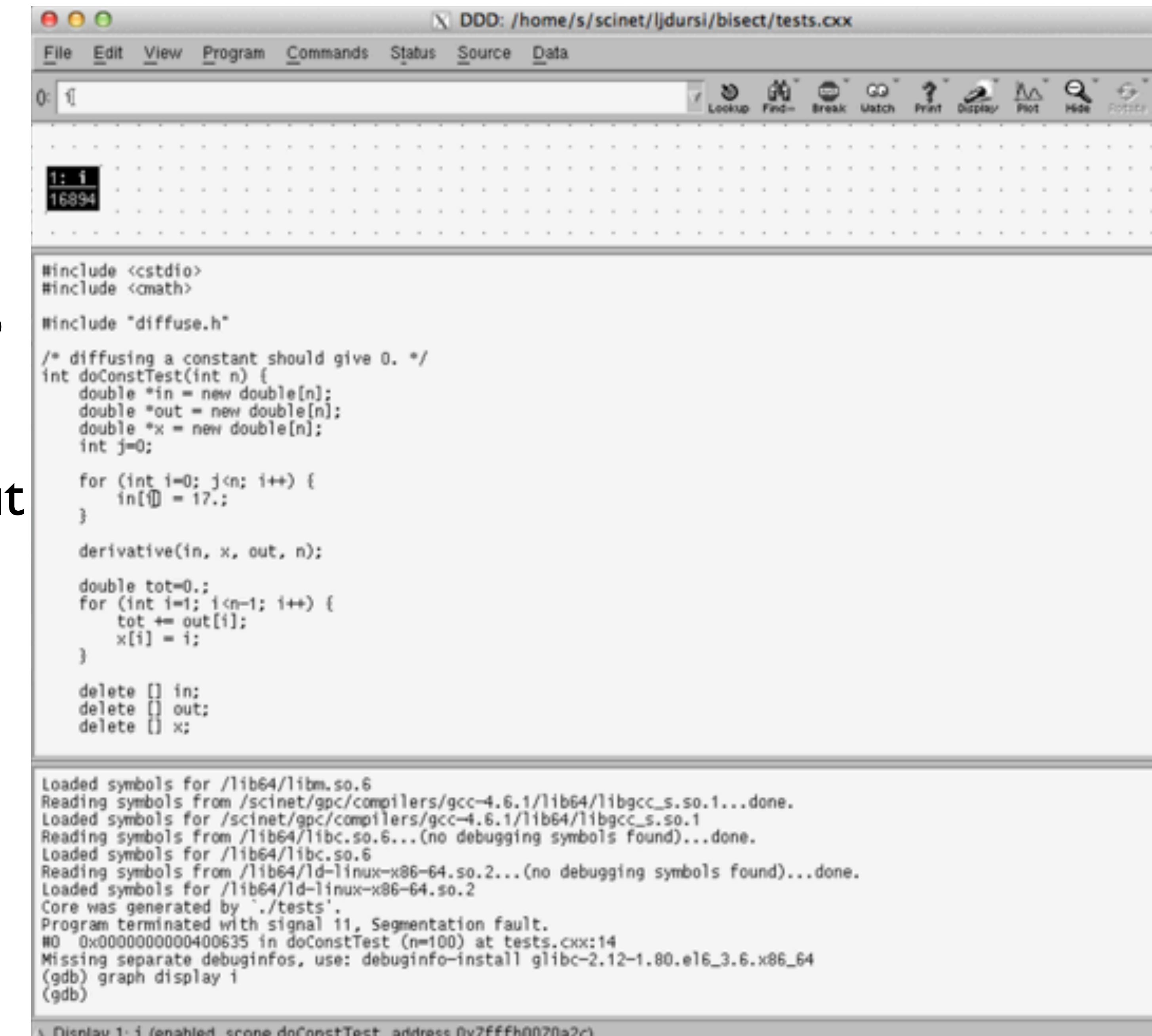
Aside - Valgrind

```
$ valgrind --tool=memcheck ./tests
==11069== Memcheck, a memory error detector
==11069== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==11069== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==11069== Command: ./tests
==11069==
Performing Constant Test...
==11069== Invalid write of size 8
==11069==    at 0x400635: doConstTest(int) (tests.cxx:14)
==11069==    by 0x4008AB: main (tests.cxx:69)
==11069==  Address 0x595d360 is 0 bytes after a block of size 800 alloc'd
==11069==    at 0x4C268CF: operator new[](unsigned long) (vg_replace_malloc.c:348)
==11069==    by 0x4005DF: doConstTest(int) (tests.cxx:8)
==11069==    by 0x4008AB: main (tests.cxx:69)
```

- Memory errors do not always give segfaults
- Had to go **way** out of bounds to get segfault above.
- Write into other variables - hard to find problem.
- Valgrind - slow, thorough. Finds illegal accesses.
- If you use external libraries, sometimes false positives

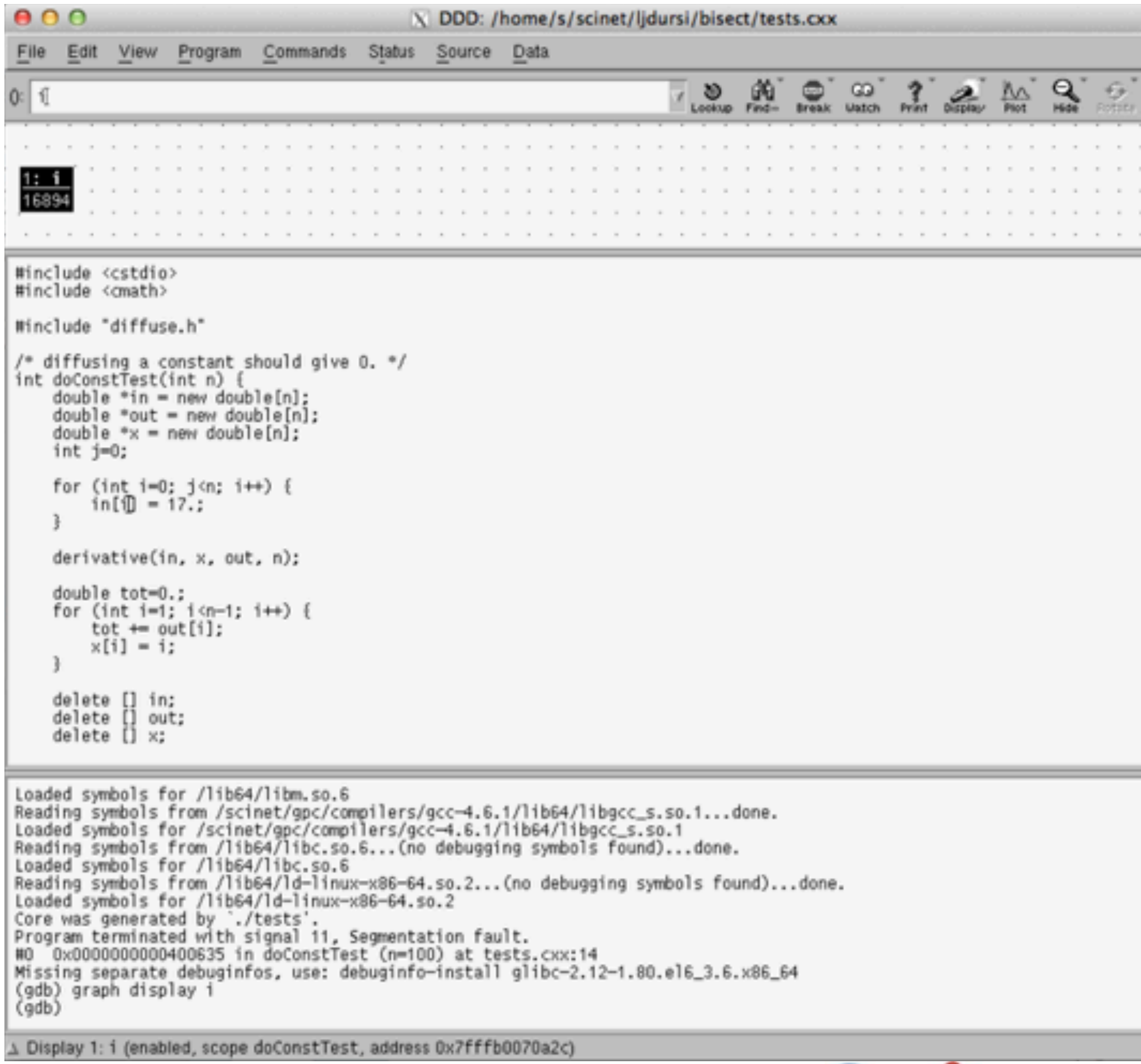
More typical case

- Generally, though, you aren't given such a clean starting point for investigation.
- Once you've narrowed down the problem, you launch the debugger to step through sections of code.
- *Can* insert printf()'s/cout's throughout code and run - but this is usually a sign that you haven't done your homework to narrow down the problem sufficiently yet.
- gdb <executable>;
set args arg1 arg2;
run or
ddd <executable>



Workflow

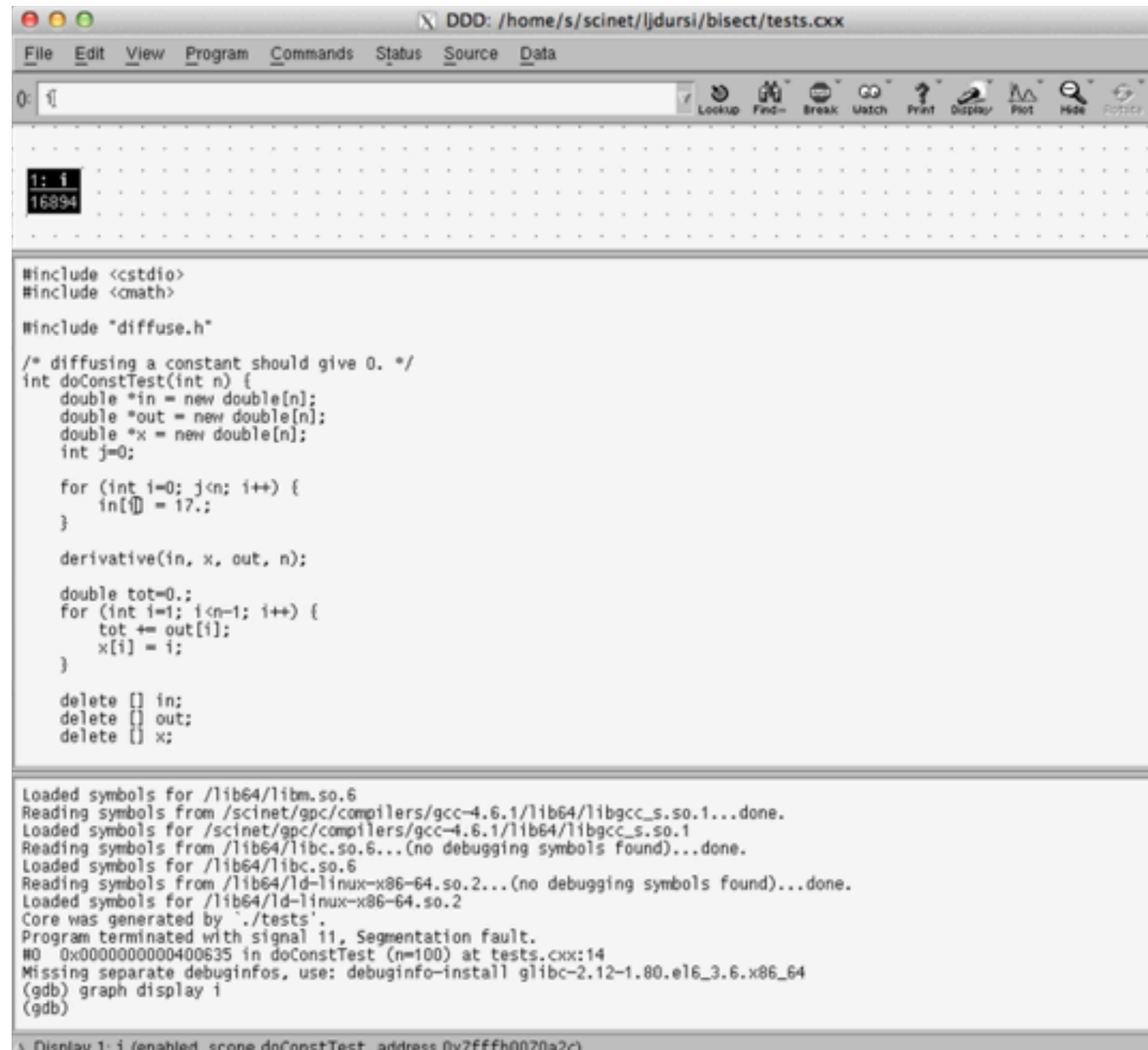
- Start up debugger with your simple, repeatable test case.
- Set a breakpoint for about half-way through, and we can see if bug has manifested itself.
- gdb: `break doConstTest` or `break tests.cxx:7`
- ddd: Go to code in viewer (may have to search for it) and click on it and click "break" icon, or right click.
- gdb: `run`; ddd: click on "run"



SciNet compute • calcul CANADA

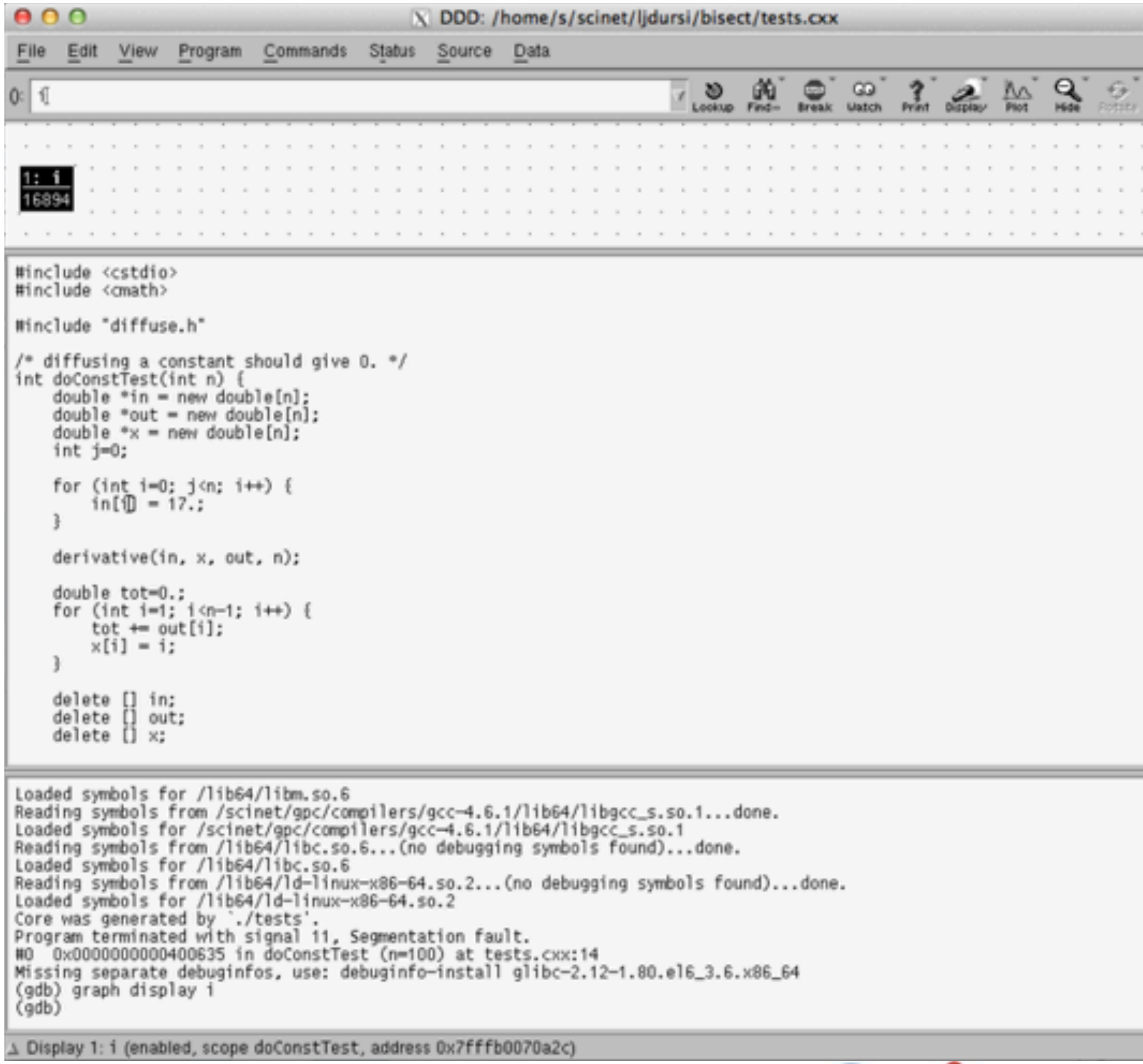
Workflow

- step - steps to following line of code, stepping into functions if necessary
- next - goes to next line of code in the current function; doesn’t go into subroutines
- print - as before
- finish/return - finish in this routine, continue from where it was called



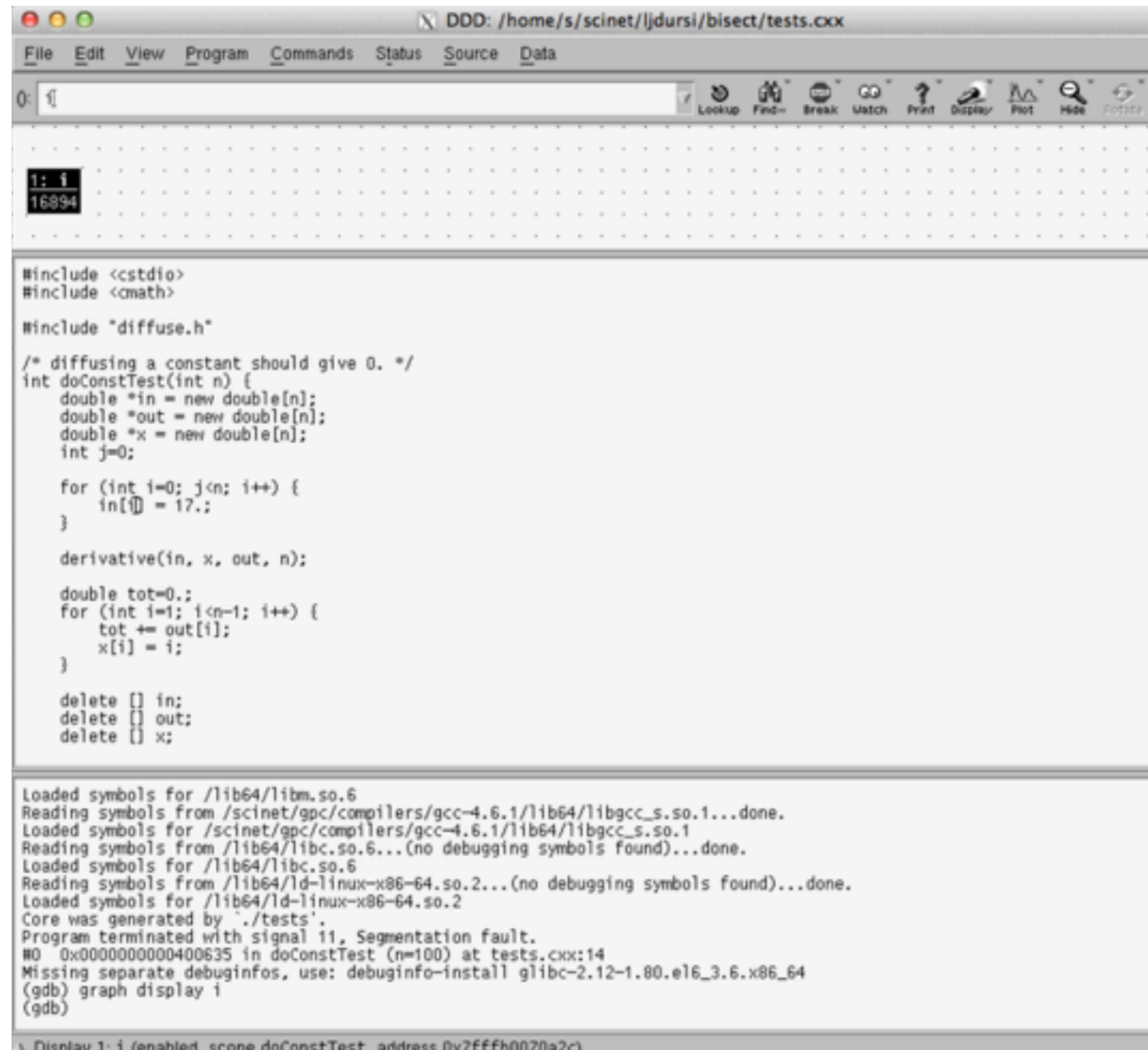
Workflow

- If bug has manifested itself, then bug was in first half; re-run, set breakpoint for 1/4 way mark.
- Otherwise, set a new breakpoint for 3/4-way mark.
- Repeat.



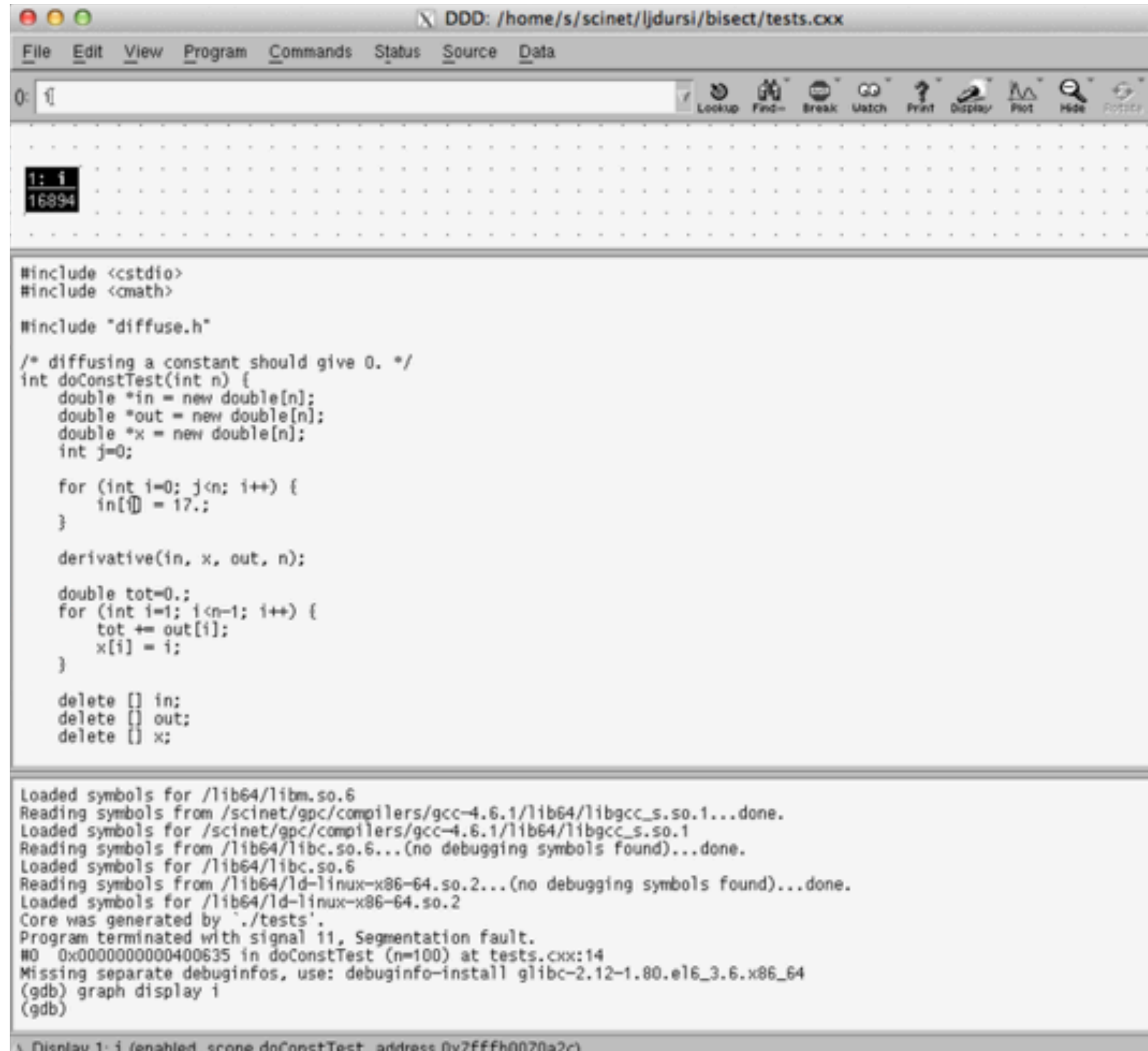
Pro Tip #1

- Most debuggers let you set *conditional* breakpoints
- Break in this loop if i > 50.
- gdb: `break tests.cxx: 14 if i > 50`
- ddd: option in break pull-down menu



Pro Tip #2

- Most debuggers let you set watchpoints.
- Break at *any* line of code if the given variable is changed. eg, `watch x`
- Variable must be visible from where you are when you set the watchpoint.
- Useful when you know what variable is being mis-set but don't know who's mis-setting it.
- Very useful if you're debugging legacy code with global variables.

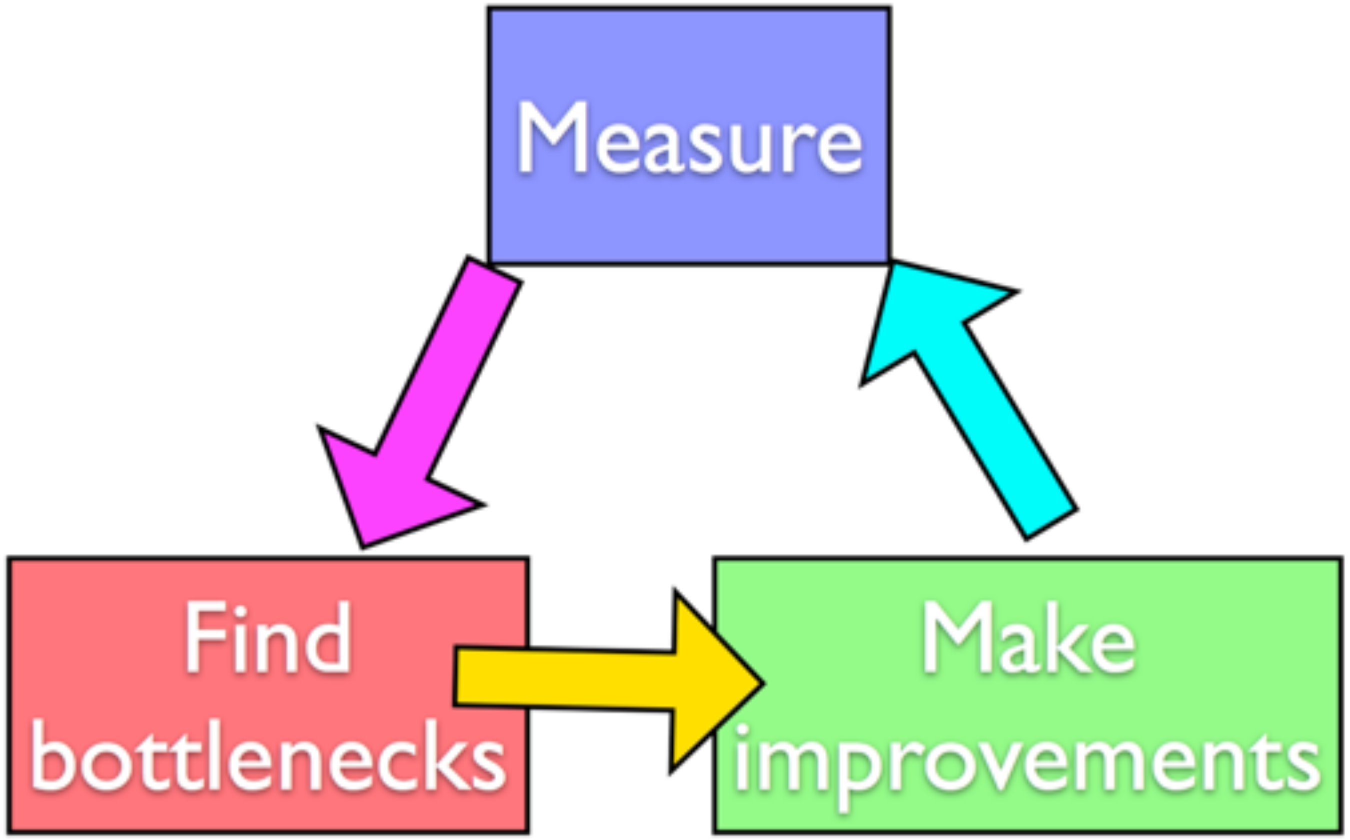


Profiling

Scientific Computing Course, Feb 2013

Profiling

- Like debuggers for debugging, profilers are evidence-based methods to find performance problems.
- Can't improve what you don't measure.



Profiling

- Where in your program is time being spent?
- Find the expensive parts
  - Don't waste 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[3] = 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\tdt\ttime\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_timestep_yx(&d, bcs, dt);
    apply_all_bcs(&d,bcs);
}
tock(&tt);
```

Profiling

- Tracing vs. Sampling
- Instrumenting vs. instrumentation-free

.

```
    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[3] = 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\tdt\ttime\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(ncdfilename,"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, ncdfilename);
        domain_plot(&d);
        outnum++;
    }
    kl_timestep_xy(&d, bcs, dt);
    apply_all_bcs(&d,bcs);

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

SciNet
compute • calcul
CANADA

Timing whole program

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

```
$ time ./a.out

[ your job output ]

real    0m2.448s
user    0m2.383s
sys     0m0.027s
```

Elapsed "walltime"

Actual user time

System time: Disk, I/O...

SciNet compute • calcul CANADA

Watching 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 then user time - not very efficient

Instrumenting regions of code

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

```c
    struct timeval calc;

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

    printf("Timing summary:\n");
    /* other timers.. */
    printf("Calc: %8.5f\n", calctime);


void tick(struct timeval *t) {
    gettimeofday(t, NULL);
}


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.);
}
```

SciNet compute • calcul CANADA

Instrumenting regions of code

- Simple example - matrix-vector multiply
- Initializes data, does multiply, saves result
- Look to see where it spends its time, speed it up.
- Options for how to access data, output data.

```c
/* 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<size; 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];
        }
    }
}
```

Matrix-vector multiply

- Simple example - matrix-vector multiply
- Initializes data, does multiply, saves result
- Look to see where it spends its time, speed it up.
- Options for how to access data, output data.

```c
/* 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<size; 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];
        }
    }
}
```

Net
compute • calcul
CANADA

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

- I/O being done in ASCII
- having to loop over data, convert to string, write to output.
- 6,252,500 write operations!
- Let's try a --binary option:

```
out = fopen("Mat-vec.dat","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);
```

Matrix-vector multiply

- Let's try a --binary option:
- Shorter...

```
out = fopen("Mat-vec.dat","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);
```

Matrix-vector multiply

- And much (36x!) faster
- File 4x smaller
- Still slow, but file I/O is always going to be slower than a multiplication.
- On to calculation...

```
$ mvm --matsize=2500
--binary
Timing summary:
   Init:   0.00976 sec
   Calc:   0.06695 sec
   I/O :   0.14218 sec
```

```
$ ./mvm --binary
$ du -h Mat-vec.dat
89M       Mat-vec.dat
$ ./mvm --binary
$ du -h Mat-vec.dat
20M       Mat-vec.dat
```

Sampling for Profiling

- How to get finer-grained information about where time is being spent?
- Can't instrument every single line.
- Compilers have tools for *sampling* execution paths.

Sampling for Profiling

- As program executes, every so often (~100ms) a timer goes off, and the current location of execution is recored
- Shows where time is being spent.

```c
/* 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<size; 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];
        }
    }
}
```

```
Line 7
Line 18
Line 223
Line 9
```

Sampling for Profiling

- Advantages:
  - Very low overhead
  - No extra instrumentation
- Disadvantages:
  - Don't know *why* code was there
  - Statistics - have to run long enough job

```c
/* 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<size; 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];
        }
    }
}
```

```
Line 7
Line 18
Line 223
Line 9
```

gprof for sampling

```
$ gcc -O3 -pg -g mat-vec-mult.c --std=c99
$ icc -O3 -pg -g mat-vec-mult.c -c99
```

turn on profiling

debugging symbols (optional, but more info)

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

gprof examines 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
[...]
```

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

gprof --line

```
gpc-f103n084-$ 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 @
  0.00      0.41     0.00       1     0.00     0.00  alloc2d (mat-vec-mult.c:130 @
  0.00      0.41     0.00       1     0.00     0.00  free2d (mat-vec-mult.c:144 @
  0.00      0.41     0.00       1     0.00     0.00  get_options (mat-vec-mult.c:1
```

SciNet compute • calcul CANADA

Then can compare to source

- Code is spending most time deep in loops
- #1 - multiplication
- #2 - I/O (old way)

```c
        for (int j=0; j<size; j++) {
            for (int i=0; i<size; i++) {
                y[i] += a[i][j]*x[j];
            }
        }

                    ...

        out = fopen("Mat-vec.dat","w");
        fprintf(out,"%d\n",size);

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

        fprintf(out,"\n");

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

        fprintf(out,"\n");

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

compute • calcul CANADA

gprof pros/cons

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

Mac OS X note

- Sadly, as of 10.5, Mac OS X no longer supports gprof.
- Instruments app in Xcode
  - Open Instruments.
  - Select the "Time Profiler" template.
  - Select your program as the "Target" dropdown menu.
  - Hit the red circle ("record") button.
  - Hit the record button again to stop recording.
  - Use the tools in Instruments to analyze your results.

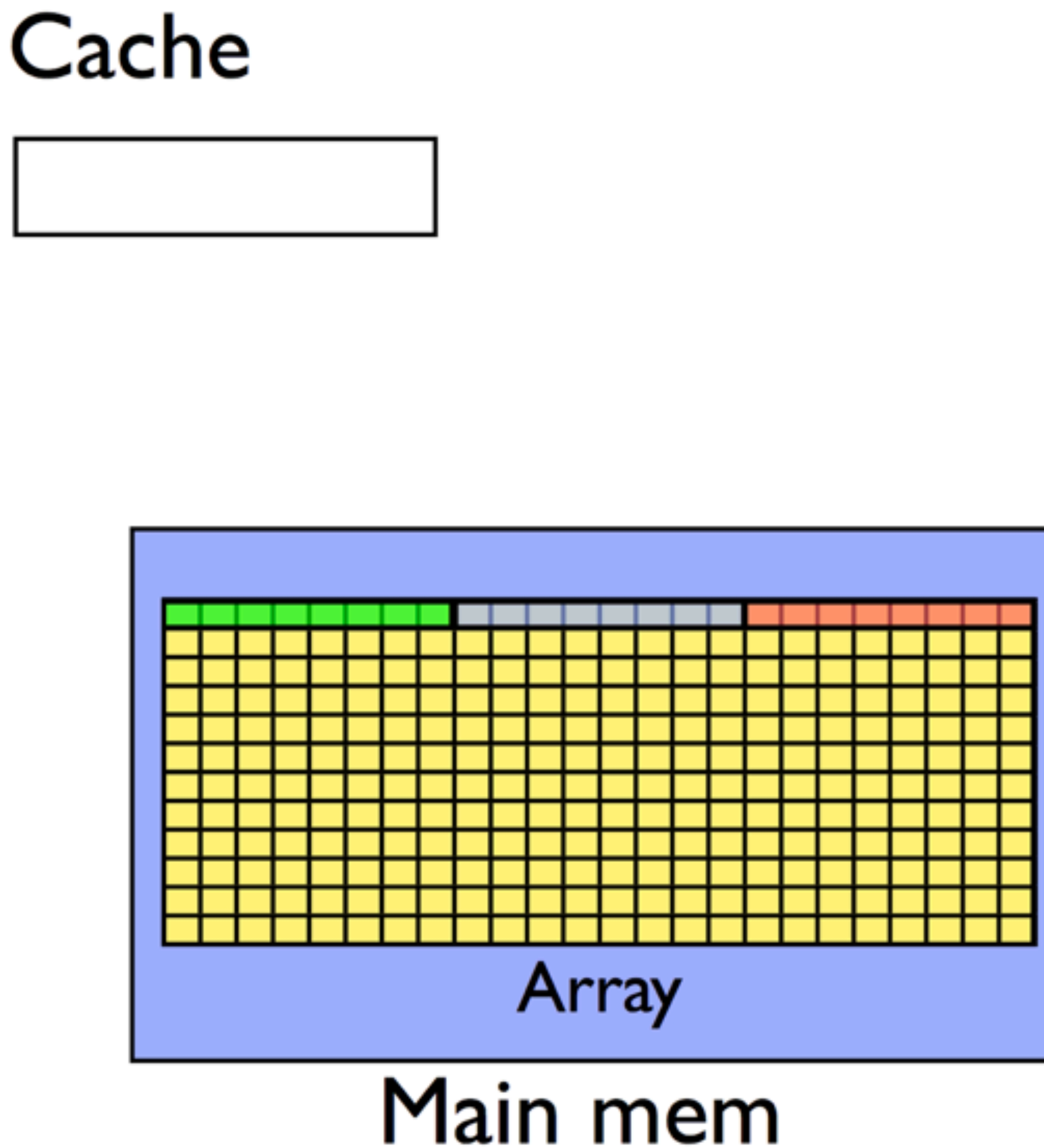
SciNet compute • calcul CANADA

Then can compare to source

```c
        for (int j=0; j<size; j++) {
            for (int i=0; i<size; i++) {
                y[i] += a[i][j]*x[j];
            }
        }

...

        out = fopen("Mat-vec.dat","w");
        fprintf(out,"%d\n",size);

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

        fprintf(out,"\n");

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

        fprintf(out,"\n");

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

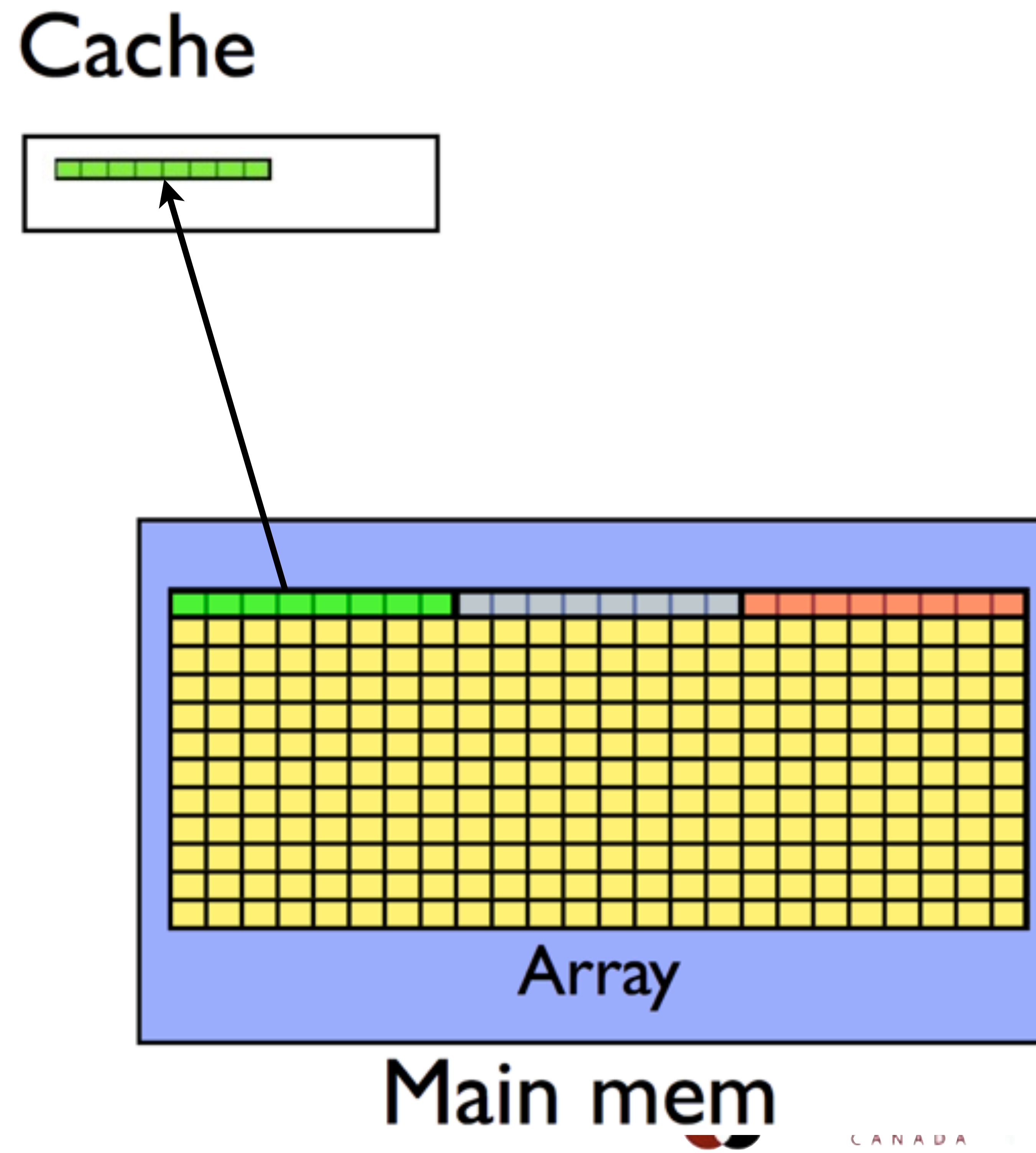
- Code is spending most time deep in loops
- #1 - multiplication
- #2 - I/O (old way)

Cache Thrashing



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

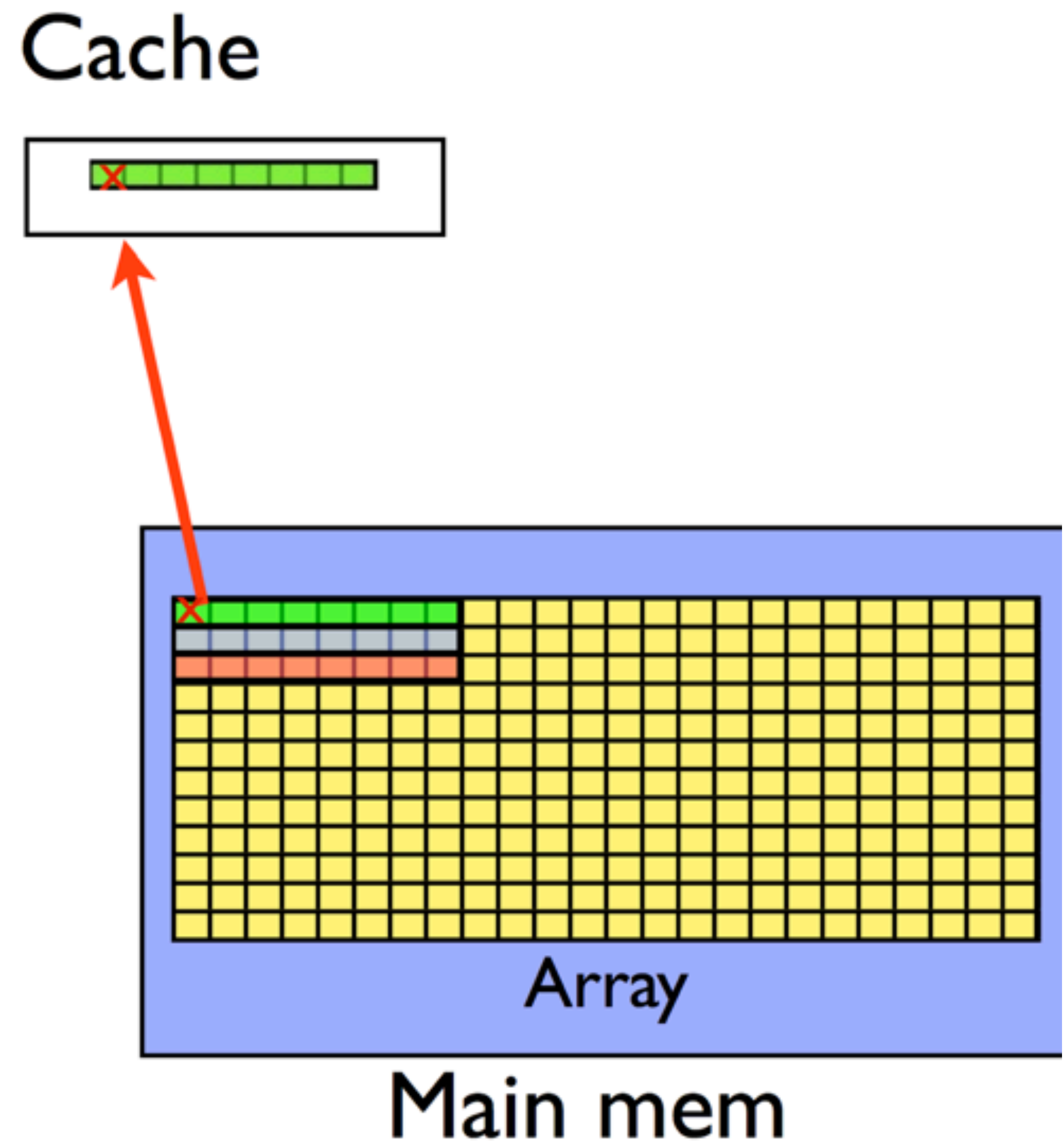
Cache Thrashing



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

Cache Thrashing

- When accessing memory out of order, much worse
- Each access is new cache line (cache miss)- slow access to main memory
- Can see ~10x slowdown



Cache Thrashing

- In C, cache-friendly order is to make last index most quickly varying

Good

```
/* do multiplication */

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);
```

Bad

Cache Thrashing

- In C, cache-friendly order is to make last index most quickly varying

Good

```c
/* do multiplication */

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);
```

Bad

compute • calcul
CANADA

Cache Thrashing

- Can see cache problems with valgrind + visualizer:
- valgrind --tool=cachegrind
- KDE tool kcachegrind available for window,s linux, mac os x.

Good

```
/* do multiplication */

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);
```

Bad

SciNet compute • calcul CANADA

Cache Thrashing

- Once cache thrashing is fixed, and assuming I/O can't be improved, Init is now the bottleneck!
- So it goes...

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

SciNet compute • calcul CANADA

IDEs

- Many choices for IDEs - integrated editor, build manager, debugger, profiler.
- Visual Studio, Xcode, Eclipse,..
- Can be **extremely** powerful, useful, especially when learning new language, code base
- Benfits/Costs of integration: have to do everything through IDE.