CSC 7700: Scientific Computing

Module C: Advanced Programming Tools

Lectures 1/2: Command Line Tools

Dr. Steven R. Brandt

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

September 20, 2013

1. Goals
2. Binary Utilities
   - nm - list symbols
   - readelf - display information about ELF tables
   - c++filt - demangle C++ and Java symbols
   - ldd - shared library dependencies
3. Debugging Utilities
   - gdb
   - Valgrind
   - Helgrind
   - Clang
   - Misc.
4. Performance Utilities
   - Gprof
   - Coverage Testing
   - Memusage
   - Profile-Guided Optimization
   - HPC Toolkit
   - Perfexpert
   - Homework

Goals

Goals

- The module *Advanced Programming Tools* will teach:
  - binary utilities
  - tools for performance analysis
  - tools for debugging
- We will use Cactus as an example of an Application Framework.

Binary Utilities

Binary Utilities

Cactus normally shields programmers from the need to know about the details of the compilation process, but occasionally some difficult problem will be encountered that requires more analysis.

- nm/readelf
- c++filt
- ldd

nm - list symbols

nm - list symbols

This tool is useful for figuring out dependencies when things are not compiling properly.

- T indicates a subroutine defined in the current object
- U indicates a subroutine or external variable that is defined in an external object file or library
- B a variable that is defined in the uninitialized data segment
- C a variable that is defined in the uninitialized data segment
- b a static variable that is defined in the uninitialized data segment
- D a variable that is defined in the initialized data segment
- d a variable that is defined in the initialized data segment

nm - list symbols

```
// demo.c
extern int var1,var2;
int var1;
int var2=3;

static int var3,var4=2;

void subroutine1();
void subroutine2();

void subroutine2() {
    var4=var3=1;
    subroutine1();
}
```

nm - list symbols

```
[sbrandt@localhost]$ gcc -c demo.c
[sbrandt@localhost]$ nm demo.o
                 U subroutine1
0000000000000000 T subroutine2
0000000000000004 C var1
0000000000000000 D var2
0000000000000000 b var3
0000000000000004 d var4
```

nm - list symbols

Things turn out differently with the C++ compiler. The C++ compiler scrambles type information into the symbol name.

```
[sbrandt@localhost]$ g++ -c demo.c
[sbrandt@localhost]$ nm demo.o
                 U _Z11subroutine1v
0000000000000000 T _Z11subroutine2v
0000000000000004 b _ZL4var3
0000000000000004 d _ZL4var4
                 U __gxx_personality_v0
0000000000000000 B var1
0000000000000000 D var2
```

nm - list symbols

Note that -C unscrambles the C++ names as they appear inside the object files. This makes things more readable.

```
[sbrandt@localhost]$ g++ -c demo.c
[sbrandt@localhost]$ nm -C demo.o
                 U subroutine1()
0000000000000000 T subroutine2()
0000000000000004 b var3
0000000000000004 d var4
                 U __gxx_personality_v0
0000000000000000 B var1
0000000000000000 D var2
```

nm - list symbols

Using the -A flag to add the file name to each line

```
sbrandt@localhost$ gcc -c demo.c demo2.c
sbrandt@localhost$ nm -g -A *.o
demo2.o:0000000000000006 T main
demo2.o:                 U printf
demo2.o:0000000000000000 T subroutine1
demo2.o:                 U subroutine2
demo2.o:0000000000000004 C var1
demo.o:                  U subroutine1
demo.o:0000000000000000 T subroutine2
demo.o:0000000000000004 C var1
demo.o:0000000000000000 D var2
```

Now you can combine with grep and see which object file defines a symbol and which object files need it.

```
sbrandt@localhost$ nm -g -A *.o|grep subroutine1
demo2.o:0000000000000000 T subroutine1
demo.o:                  U subroutine1
```

nm - list symbols

Using the -u flag to list undefined symbols

```
sbrandt@localhost$ gcc -c -g demo.c demo2.c
sbrandt@localhost$ nm -uA *.o
demo2.o:                  U printf
demo2.o:                  U subroutine2
demo.o:                  U subroutine1
```

If you'd like to know where you've tried to use those undefined symbols in your source code, use -l.

```
sbrandt@localhost$ nm -uAl *.o
demo2.o:                  U printf    /home/sbrandt/demo2.c:8
demo2.o:                  U subroutine2  /home/sbrandt/demo2.c:7
demo.o:                  U subroutine1   /home/sbrandt/demo.c:13
```

readelf - display information about ELF tables

ELF stands for "Executable and Linking Format." ELF can be used to get essentially the same information as nm, with one important advantage. It can work on some files that nm can't (stripped shared libraries). Lines with UND aren't defined in the current source file.

```
sbrandt@localhost$ readelf -s demo.o

Symbol table '.symtab' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS demo.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 var3
     6: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 var4
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
```

c++filt - demangle C++ and Java symbols

c++filt - demangle C++ and Java symbols

- Simple tool that transforms mangled C++ symbol definitions on the input stream to human readable form.
- "nm demo.o — c++filt" does the same thing as "nm -C demo.o"
- readelf has no -C flag, so "readelf -s demo.o | c++filt" is how to do it.
- First version was written by yours truly in the 80's.

ldd - shared library dependencies

ldd - shared library dependencies

Ldd can tell you where your shared libraries are.

```
[sbrandt@localhost]$ gcc --warn-all demo.c demo2.c
[sbrandt@localhost]$ ldd ./a.out
    linux-vdso.so.1 =>  (0x00007fff53dff000)
    libc.so.6 => /lib64/libc.so.6 (0x00000376d600000)
    /lib64/ld-linux-x86-64.so.2 (0x00000376d200000)
```

ldd - shared library dependencies

Sometimes ldd can't find your shared library. You can fix this by modifying the LD_LIBRARY_PATH variable.
First, let's create our own shared library from our demo.c source file. We need the -shared and -fPIC (Position Independent Code) flags to make this work.

```
[sbrandt@localhost]$ gcc -fPIC -shared -o libdemo.so demo.c
[sbrandt@localhost]$ gcc demo2.c libdemo.so
sbrandt@localhost$ ./a.out
./a.out: error while loading shared libraries: libdemo.so:
    cannot open shared object file: No such file or directory
```

ldd - shared library dependencies

What went wrong?

```
[sbrandt@localhost]$ ldd a.out
    linux-vdso.so.1 =>  (0x00007ffffc3ff000)
    libdemo.so => not found
    libc.so.6 => /lib64/libc.so.6 (0x00000376d600000)
    /lib64/ld-linux-x86-64.so.2 (0x00000376d200000)
```

You see that libdemo.so is not found? That's because linux won't look in the current directory by default. We have to tell it to do so.

```
[sbrandt@localhost]$ LD_LIBRARY_PATH=. ldd a.out
    linux-vdso.so.1 =>  (0x00007fff30784000)
    libdemo.so => ./libdemo.so (0x00007f9634fe7000)
    libc.so.6 => /lib64/libc.so.6 (0x00000376d600000)
    /lib64/ld-linux-x86-64.so.2 (0x00000376d200000)
```

alternate shared library dependency tracing

An alternative to using ldd is to set some variables. Note that LD_DEBUG appends the process id to the file name. That's helpful for MPI. Note that "ls -t | head -1" shows the most recently modified file.

```
[sbrandt@localhost]$ export LD_DEBUG=libs
[sbrandt@localhost]$ export LD_DEBUG_OUTPUT=debug.txt
[sbrandt@localhost]$ ./a.out
./a.out: error while loading shared libraries: libdemo.so:
    cannot open shared object file: No such file or directory
[sbrandt@localhost]$ ls -t | head -1
debug.txt.4478
[sbrandt@localhost]$ cat debug.txt.4478
      4478: find library=libdemo.so [0]; searching
      4478:  search cache=/etc/ld.so.cache
      4478:  search path=/lib64/tls/x86_64:/lib64/tls:/lib64/x86_64
      4478:   trying file=/lib64/tls/x86_64/libdemo.so
      4478:   trying file=/lib64/tls/libdemo.so
      ...
```

Debugging Utilities

Debugging Utilities

Memory problems (indexing outside of array bounds, using unallocated memory, etc.) and race conditions are insidious problems. Results may be difficult to reproduce and the causes difficult to identify. You have to debug early and often if you want to stay sane. These tools can help you.

- gdb
- Valgrind
- Helgrind
- clang
- misc

gdb

gdb

The gnu debugger is widely available. Very helpful when you have a memory problem. It has many useful features:

1. where - Gives the stack trace at the point of failure
2. print - Prints the contents of a variable
3. break/clear - Set/clear break point
4. step/next - advance line by line
5. run/cont - run/continue

gdb

Let's see it in action. This code has an obvious problem:

```
int main() {
    int *a = 0;
    a[10] = 4;
    return 0;
}
```

gdb

```
sbrandt@sbrandt-think examples$ gdb ./test1
GNU gdb (GDB) Fedora (7.6-34.fc19)
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/sbrandt/repos/sci-comp-2013/Module-
(gdb) run
Starting program: /home/sbrandt/repos/sci-comp-2013/Module-C/e

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005c4 in main () at test1.c:4
4       a[10] = 4;
Missing separate debuginfos, use: debuginfo-install glibc-2.17
(gdb) where
#0  0x00000000004005c4 in main () at test1.c:4
(gdb)
```

gdb

```
#include <stdlib.h>
void set10(int *a) {
    a[10] = 4;
}
int main() {
    int *a = (int *)malloc(11*sizeof(int));
    set10(a);
    set10(0);
    return 0;
}
```

```
(gdb) where
#0  0x0000000000400600 in set10 (a=0x0) at test1b.c:3
#1  0x0000000000400637 in main () at test1b.c:8
```

gdb

```
(gdb) run
(gdb) break test1b.c:6
Breakpoint 1, main () at test1b.c:6
6       int *a = (int *)malloc(11*sizeof(int));
(gdb) step
7       set10(a);
(gdb) step
set10 (a=0x602010) at test1b.c:3
3       a[10] = 4;
(gdb) step
4   }
(gdb) step
main () at test1b.c:8
8       set10(0);
(gdb) step
set10 (a=0x0) at test1b.c:3
3       a[10] = 4;
(gdb) step
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400600 in set10 (a=0x0) at test1b.c:3
3       a[10] = 4;
```

Valgrind

Valgrind

Valgrind simulates your program in memory. It is slow, but it reveals many problems.

```
[sbrandt@localhost t]$ cat test.c
#include <stdio.h>
int main() {
    int a;
    if(a < 0) printf("a is negative\n");
    return 0;
}
[sbrandt@localhost t]$ gcc -g test.c
[sbrandt@localhost t]$ valgrind ./a.out
...
==3512== Conditional jump or move depends on uninitialized val
==3512==    at 0x4004D0: main (test.c:4)
...
```

Valgrind

```
[sbrandt@localhost t]$ cat test2.c
#include <stdlib.h>
int main() {
    int *a = (int*)malloc(10*sizeof(int));
    a[10] = 4;
    return 0;
}
[sbrandt@localhost t]$ valgrind ./a.out
...
==3482== Invalid write of size 4
==3482==    at 0x4004E2: main (in /home/sbrandt/c/t/a.out)
==3482==  Address 0x4c0c068 is 0 bytes after a block of size 4
==3482==    at 0x4A0515D: malloc (vg_replace_malloc.c:195)
==3482==    by 0x4004D5: main (in /home/sbrandt/c/t/a.out)
...
```

Valgrind

We fix test2.c like this:

```
#include <stdlib.h>
int main() {
    int *a = (int*)malloc(10*sizeof(int));
    a[9] = 4;
    return 0;
}
```

Can Valgrind tell us anything more about our program?

Valgrind

```
sbrandt@sbrandt-think examples$ valgrind --leak-check=full ./a.out
==15928== Memcheck, a memory error detector
==15928== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==15928== Command: ./a.out
==15928==
==15928== HEAP SUMMARY:
==15928==     in use at exit: 40 bytes in 1 blocks
==15928==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==15928==
==15928== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==15928==    at 0x4A06409: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-l
==15928==    by 0x400541: main (in /home/sbrandt/repos/sci-comp-2013/Module-C/examp
==15928==
==15928== LEAK SUMMARY:
==15928==    definitely lost: 40 bytes in 1 blocks
==15928==    indirectly lost: 0 bytes in 0 blocks
==15928==      possibly lost: 0 bytes in 0 blocks
==15928==    still reachable: 0 bytes in 0 blocks
==15928==         suppressed: 0 bytes in 0 blocks
==15928==
==15928== For counts of detected and suppressed errors, rerun with: -v
==15928== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

LSU

Helgrind

Helgrind

Useful for debugging threads. Prone to false positives.

```
[sbrandt@localhost]$ cat race.c
#include <stdio.h>
#include <pthread.h>
int i = 0;
void *task(void *v) {
    for(int j=0;j<100000;j++) i++;
    return 0;
}
int main() {
    const int N = 10; pthread_t p[N];
    for(int i=0;i<N;i++)
        pthread_create(&p[i],0,task,0);
    for(int i=0;i<N;i++)
        pthread_join(p[i],0);
    printf("i=%d\n",i);
}
```

Helgrind

We can see that we have a problem on line 5.

```
[sbrandt@localhost]$ gcc -std=c99 -g race.c -lpthread
[sbrandt@localhost]$ valgrind --tool=helgrind \
    --log-file=val.out ./a.out 2>&1 | grep race.c
i=1000000
sbrandt@localhost$ grep race.c val.out|sort -u
==20857==    at 0x4005D5: task (race.c:5)
==20857==    at 0x4005DE: task (race.c:5)
==20857==    by 0x40067A: main (race.c:11)
```

Helgrind

Here's one way we can fix it, using an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
int i = 0;
void *task(void *v) {
    for(int j=0;j<100000;j++)
        __sync_add_and_fetch(&i,1);
    return 0;
}
int main() {
    const int N = 10; pthread_t p[N];
    for(int i=0;i<N;i++)
        pthread_create(&p[i],0,task,0);
    for(int i=0;i<N;i++)
        pthread_join(p[i],0);
    printf("i=%d\n",i);
}
```

Helgrind

Now everything is fixed!

```
[sbrandt@localhost]$ gcc -std=c99 -g race.c -lpthread
[sbrandt@localhost]$ valgrind --tool=helgrind \
    --log-file=val.out ./a.out 2>&1 | grep race.c
i=1000000
sbrandt@localhost$ grep race.c val.out|sort -u
```

Note that Helgrind works well with pthreads, but doesn't work well with other forms of thread parallelism.

Clang

clang

- clang stands for C-language compiler. Based on the llvm (Low Level virtual machine) project.
- download it here
  `http://clang.llvm.org/get_started.html#build`
- performs details source code analysis

clang

Just using the clang compiler can help you find new bugs

```
[sbrandt@localhost t]$ cat test3.c
int main() {
    int b[1];
    b[2] = 3;
}
[sbrandt@localhost t]$ clang test3.c
test3.c:3:5: warning: array index of '2' indexes past the end
      (that contains 1 element) [-Warray-bounds]
    b[2] = 3;
    ^ ~

test3.c:2:5: note: array 'b' declared here
    int b[1];
    ^

1 warning generated.
```

clang

Another way to discover problems is with -fsanitize=undefined-trap -fsanitize-undefined-trap-on-error.

```
[sbrandt@localhost t]$ cat test4.c
void foo(int n,int v) {
    int f[2];
    f[n] = v;
}
int main() {
    foo(3,3);
    return 0;
}
[sbrandt@localhost t]$ clang -g -fsanitize=undefined-trap \
    -fsanitize-undefined-trap-on-error test4.c
[sbrandt@localhost t]$ gdb -eval-command=run ./a.out
Program received signal SIGILL, Illegal instruction.
0x000000000040049c in foo (n=3, v=3) at test4.c:3
3       f[n] = v;
```

clang

Integer overflow problems can be discovered with -ftrapv.

```
[sbrandt@localhost t]$ cat test5.c
#include <limits.h>
#include <stdio.h>
void foo(int n) {
    if(n > n+1) printf("overflow\n");
}
int main() {
    foo(INT_MAX);
}
[sbrandt@localhost t]$ clang -g -ftrapv test5.c
[sbrandt@localhost t]$ gdb -eval-command=run ./a.out
Program received signal SIGILL, Illegal instruction.
0x0000000004004f3 in foo (n=2147483647) at test5.c:4
4       if(n > n+1) printf("overflow\n");
```

Misc.

Misc.

- gfortran -fbounds-check - generates bounds checking code for fortran. This flag is *silently ignored* for gcc and g++.
- Cactus debug build - automatically detects indexes out of bounds in grid functions, plus does other things
- In general, using a different compiler or architecture can be helpful in exposing bugs or problems in code.
- `#include <assert.h>` is very helpful. Use -DNDEBUG to make assertions go away.

Misc.

```
[sbrandt@localhost ~]$ cat here.h
#ifndef HERE_H_
#define HERE_H_
#include <stdio.h>
#define HERE printf("%s,%d\n",__FILE__,__LINE__);
#define VARN(X) printf("%s=%d\n",#X,X);
#define VAR(X) printf("%s=%d ",#X,X);
#endif
```

Performance Utilities

Performance Utilities

- gprof
- Coverage Testing
- Memusage
- Profile-Guided Optimization
- hpctoolkit
- perfexpert

Gprof

gprof

Gprof provides basic, low-level profiling of source code. Helps you find your performance hot spots and target your optimization efforts. To use, build your code with `-pg`.

```
g++ -pg -g -o wave wave.cpp
```

When you run your code, a file named `gmon.out` is generated. To view the results, use the `gprof` command.

```
gprof wave gmon.out -p
```

gprof

```
sbrandt@sbrandt-think examples$ gprof wave gmon.out -p
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.27      0.01     0.01     4000     2.51     2.51  deriv(double*, double*)
  0.00      0.01     0.00    19702     0.00     0.00  x(int)
  0.00      0.01     0.00     8000     0.00     0.00  bound(double*)
  0.00      0.01     0.00     1000     0.00    10.03  update(int)
  0.00      0.01     0.00      200     0.00     0.00  print_minmax(int)
  0.00      0.01     0.00      200     0.00     0.00  print(double*, int)
  0.00      0.01     0.00      100     0.00     0.00  sq(double)
  0.00      0.01     0.00        1     0.00     0.00  _GLOBAL__sub_I_u
  0.00      0.01     0.00        1     0.00     0.00  __static_initialization_and_d
  0.00      0.01     0.00        1     0.00     0.00  init()
```

Who knew that `deriv` was our most expensive routine?

Coverage Testing

Coverage Testing

Coverage testing builds in automatic instrumentation of your code to determine which lines are being run and how often. It can help you construct more thorough tests.

```
[sbrandt@localhost t]$ gcc -ftest-coverage -fprofile-arcs test2.c
[sbrandt@localhost t]$ ./a.out
[sbrandt@localhost t]$ ls -t | head -3
test2.gcda
a.out
test2.gcno
```

Coverage Testing

```
[sbrandt@localhost t]$ gcov test2.gcda
File 'test2.c'
Lines executed:100.00% of 4
test2.c:creating 'test2.c.gcov'

[sbrandt@localhost t]$ cat test2.c.gcov
        -:    0:Source:test2.c
        -:    0:Graph:test2.gcno
        -:    0:Data:test2.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdlib.h>
        1:    2:int main() {
        1:    3:    int *a = (int*)malloc(10*sizeof(int));
        1:    4:    a[10] = 4;
        1:    5:    return 0;
        -:    6:}
```

Memusage

Memusage

Memusage is a utility which can help you monitor your heap and stack usage.

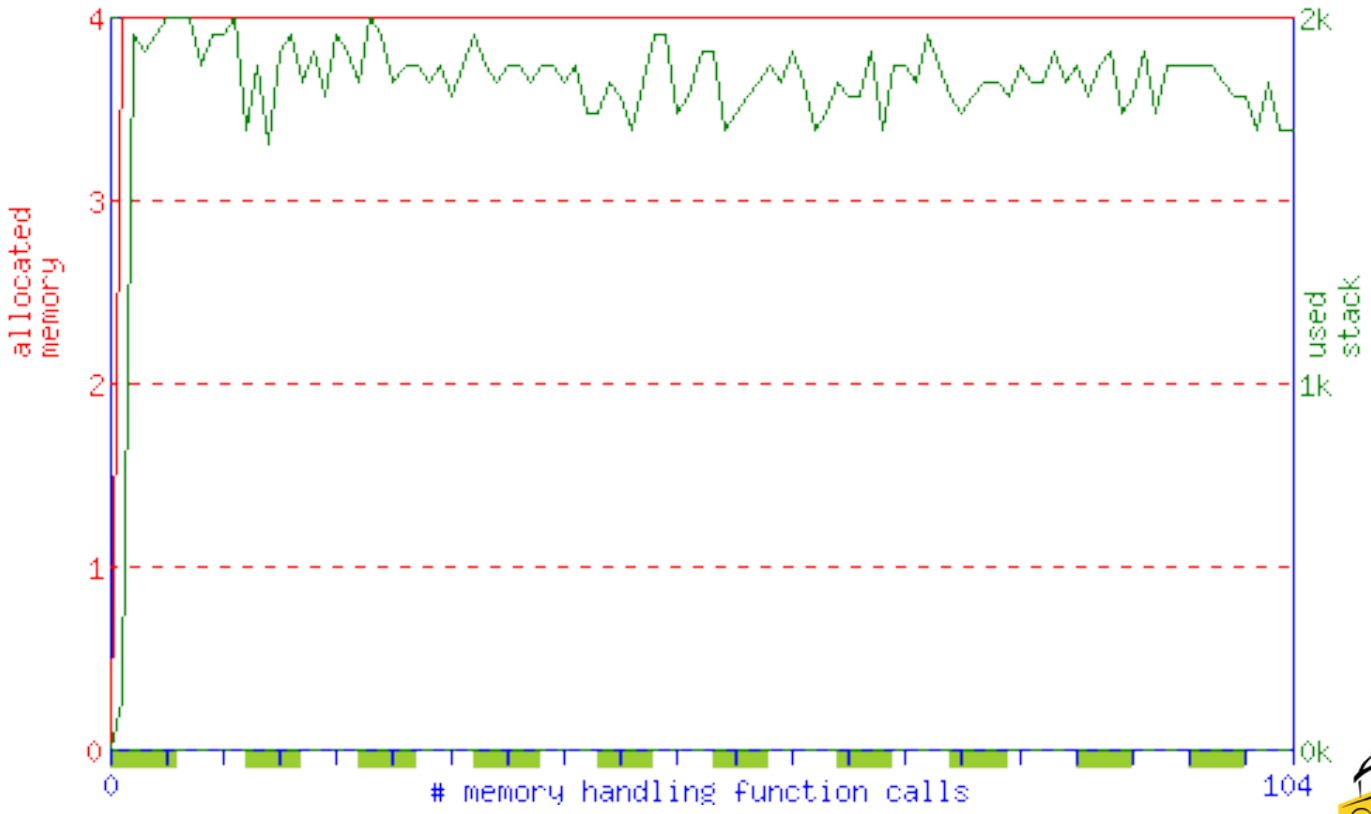
```
sbrandt@localhost$ cat fib.c
#include <stdio.h>

int fib(int n) {
    if(n < 2)
        return n;
    return fib(n-1)+fib(n-2);
}

int main() {
    int *n = new int(35);
    printf("fib(%d)=%d\n",*n,fib(*n));
    return 0;
}
sbrandt@localhost$ g++ fib.c -lmemusage
```

Memusage

```
sbrandt@localhost$ MEMUSAGE_OUTPUT=mem.out \
    LD_PRELOAD=/usr/lib64/libmemusage.so ./a.out
fib(35)=9227465

Memory usage summary: heap total: 4, heap peak: 4, stack peak: 2976
         total calls   total memory   failed calls
 malloc|             1              4              0
realloc|             0              0              0  (nomove:0, dec:0
 calloc|             0              0              0
   free|             2              0
Histogram for block sizes:
    0-15              1 100% ==========================================
```

Memusage

```
sbrandt@localhost$ memusagestat -o mem.png mem.out -x 500 -y 300
```



Profile-Guided Optimization

Profile Guided Optimization

- Run your code and see how fast it is
- Compile with -fprofile-generate
- Run your code (it will be slightly slower)
- Compile again with -fprofile-use
- Run your code again, and see how much faster it is
- Works well if you have lots of if's

Profile Guided Optimization

```
#include <stdlib.h>
void bubble(int *a,int n) {
    for(int j=0;j<n-1;j++)
        for(int i=0;i<n-1;i++)
            if(a[i] > a[i+1]) {
                int s = a[i];
                a[i] = a[i+1];
                a[i+1] = s;
            }
}
int main() {
    const int n = 60000;
    int *a = new int[n];
    for(int i=0;i<n;i++)
        a[i] = rand();
    bubble(a,n);
    return 0;
}
```

Profile Guided Optimization

- Speedup is approximately 6% for Bubble Sort on mike (6.44 to 6.08)
- Can be used to speed up Firefox and gcc
- Used extensively by the Java Virtual Machine
- With the Intel compilers there's -prof-gen and -prof-use
- Actually slows down!

HPC Toolkit

HPC Toolkit

To demonstrate we will use the HPC Toolkit on Arete (arete.cct.lsu.edu). We'll run Cactus for 200 steps, then run the viewer.

- Compile with debug mode:

  ```
  sim build --debug --thornlist ../WaveDemo.th --optionlist local.cfg
  ```

- Run with hpcrun: `mpiexec -np 2 hpcrun -e PAPI_TOT_CYC:1000000 -e PAPI_L2_DCM:100000 \`

  `-e PAPI_FP_OPS:500000 -e PAPI_TLB_DM:100000 ./exe/cactus_sim-debug ./pars/WaveDemo.par`

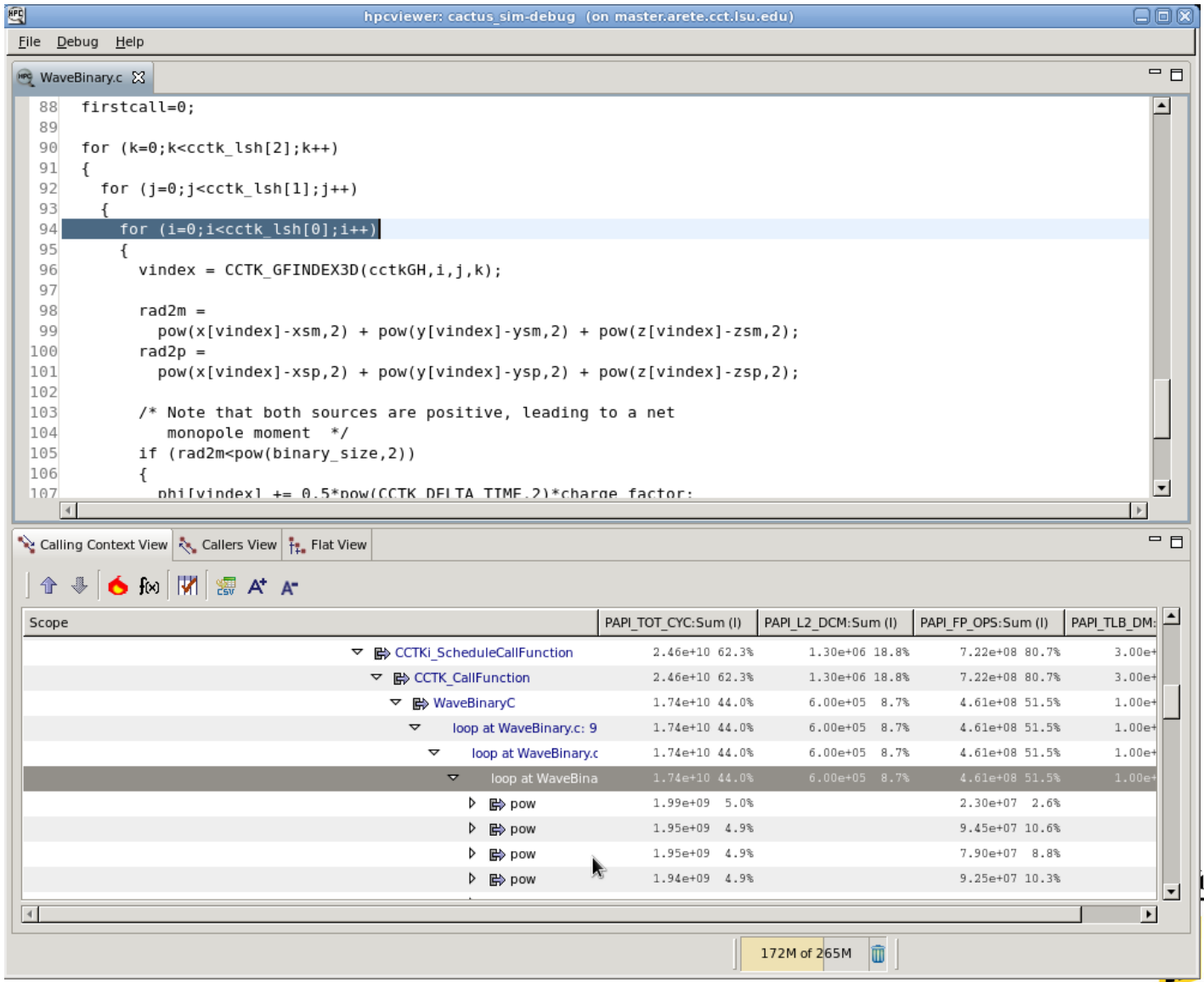
- Postprocess with hpcstruct: `hpcstruct ./exe/cactus_sim-debug`

- Postprocess with hpcprof:

  ```
  hpcprof -S ./cactus_sim-debug.hpcstruct -I ./ hpctoolkit-cactus_sim-debug-measurements/
  ```

- View: `hpcviewer hpctoolkit-cactus_sim-debug-database/`

HPC Toolkit


hpcviewer: cactus_sim-debug (on master.arete.cct.lsu.edu)
File Debug Help
WaveBinary.c
88 firstcall=0;
89
90 for (k=0;k<cctk_lsh[2];k++)
91 {
92 for (j=0;j<cctk_lsh[1];j++)
93 {
94 for (i=0;i<cctk_lsh[0];i++)
95 {
96 vindex = CCTK_GFINDEX3D(cctkGH,i,j,k);
97
98 rad2m =
99 pow(x[vindex]-xsm,2) + pow(y[vindex]-ysm,2) + pow(z[vindex]-zsm,2);
100 rad2p =
101 pow(x[vindex]-xsp,2) + pow(y[vindex]-ysp,2) + pow(z[vindex]-zsp,2);
102
103 /* Note that both sources are positive, leading to a net
104 monopole moment */
105 if (rad2m<pow(binary_size,2))
106 {
Calling Context View
Callers View
Flat View
Scope
PAPI_TOT_CYC:Sum (I)
PAPI_L2_DCM:Sum (I)
PAPI_FP_OPS:Sum (I)
PAPI_TLB_DM:
CCTKi_ScheduleCallFunction 2.46e+10 62.3% 1.30e+06 18.8% 7.22e+08 80.7% 3.00e+
CCTK_CallFunction 2.46e+10 62.3% 1.30e+06 18.8% 7.22e+08 80.7% 3.00e+
WaveBinaryC 1.74e+10 44.0% 6.00e+05 8.7% 4.61e+08 51.5% 1.00e+
loop at WaveBinary.c: 9 1.74e+10 44.0% 6.00e+05 8.7% 4.61e+08 51.5% 1.00e+
loop at WaveBinary.c 1.74e+10 44.0% 6.00e+05 8.7% 4.61e+08 51.5% 1.00e+
loop at WaveBina 1.74e+10 44.0% 6.00e+05 8.7% 4.61e+08 51.5% 1.00e+
pow 1.99e+09 5.0% 2.30e+07 2.6%
pow 1.95e+09 4.9% 9.45e+07 10.6%
pow 1.95e+09 4.9% 7.90e+07 8.8%
pow 1.94e+09 4.9% 9.25e+07 10.3%
172M of 265M

Perfexpert

perfexpert

This is a tool from TACC that attempts to turn the output of HPC Toolkit into something easier to understand.

- Home Page: http://www.tacc.utexas.edu/perfexpert/
- Automatically runs a sequence of experiments
- Shows information about the most significant sections of code and rate loops on how well they perform.
- AutoSCOPE attempts to give concrete advice on how to improve problem areas of code.

perfexpert

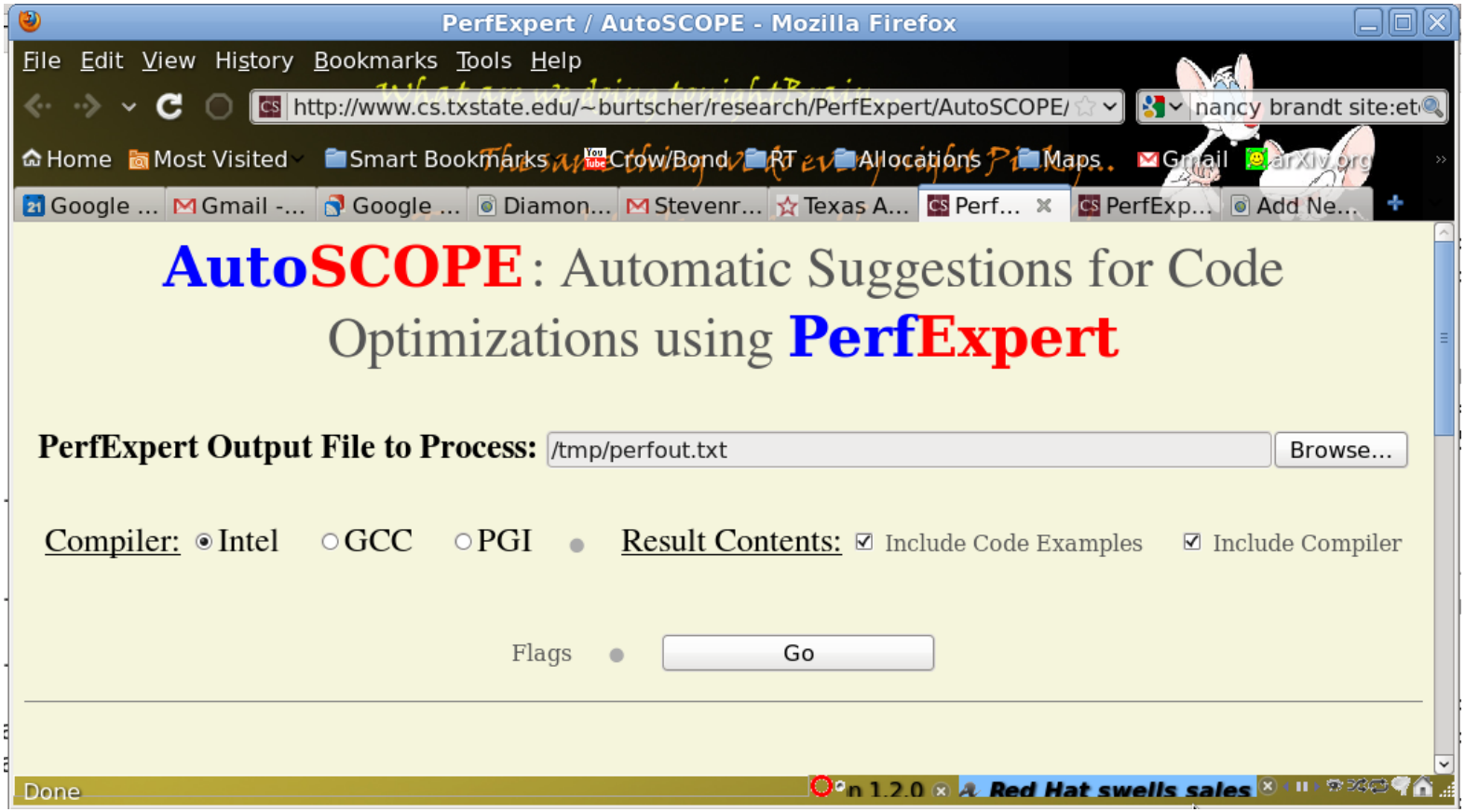
Running Perfexpert has fewer steps than running HPC Toolkit. First you run experiments, then you run the filter on the output (experiment.xml). When you run the regular perfexpert tool, you specify what level of analysis you need. Below we choose to ignore diagnostics for loops that take less than 10% of the run time.

- `[sbrandt@master Cactus]$ perfexpert_run_exp ./exe/cactus_sim ./pars/WaveDemo.par`
- `[sbrandt@master Cactus]$ perfexpert 0.1 ./experiment.xml > perfout.txt`

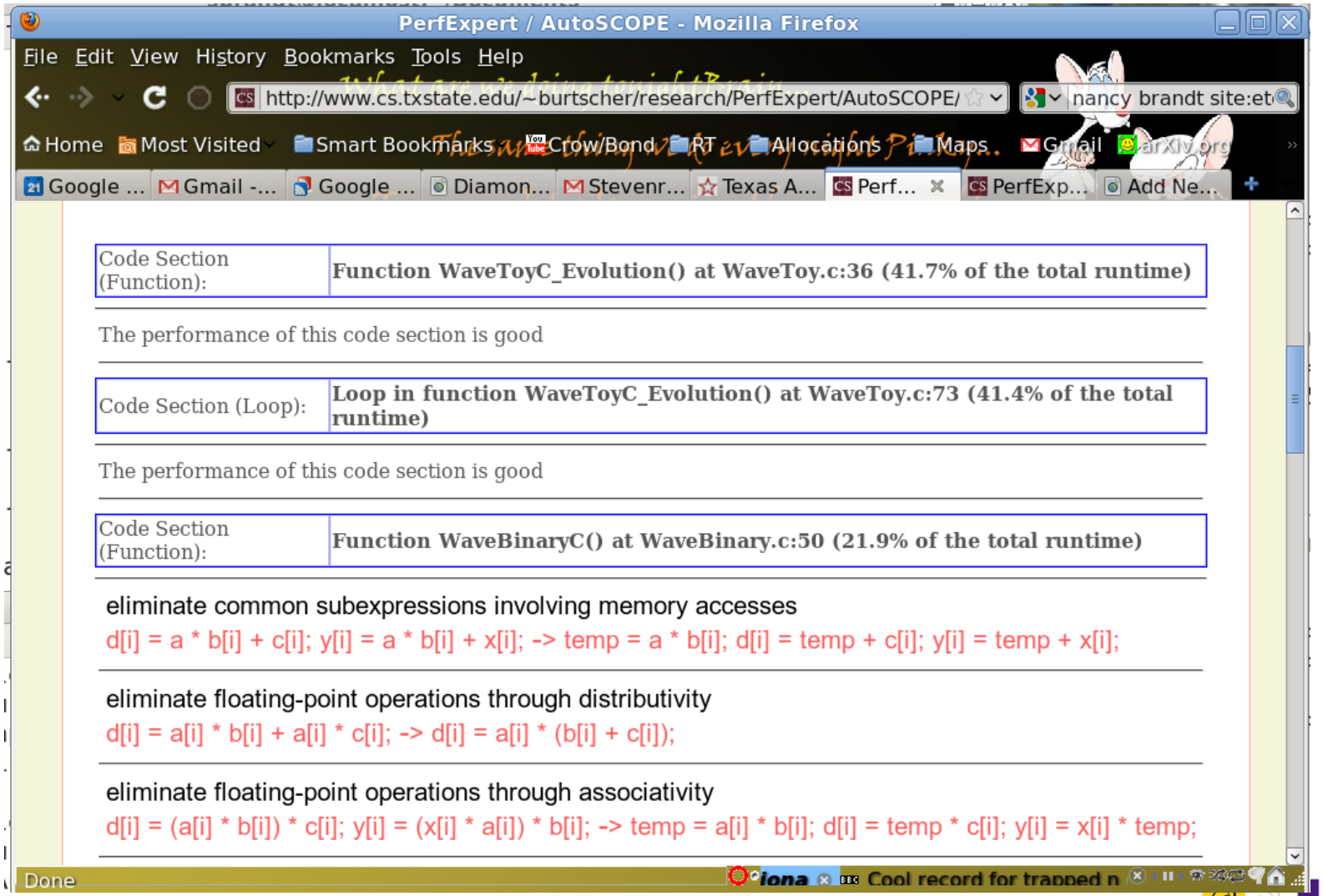
perfexpert

```
Total running time for "./experiment.xml" is 25.236 sec

Function WaveToyC_Evolution() at WaveToy.c:36 (41.7% of the total runtime)
================================================================================
ratio to total instrns       %  0.........25..........50.........75........100
   - floating point      :   22 **********
   - data accesses       :   24 ***********
* GFLOPS (% max)         :   10 *****
--------------------------------------------------------------------------------
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                :  0.6 >>>>>>>>>>
upper bound estimates
* data accesses          :  1.1 >>>>>>>>>>>>>>>>>>>>>>
   - L1d hits            :  0.9 >>>>>>>>>>>>>>>>>>
   - L2d hits            :  0.1 >>
   - L2d misses          :  0.2 >>>
* instruction accesses   :  0.6 >>>>>>>>>>
   - L1i hits            :  0.6 >>>>>>>>>>
   - L2i hits            :  0.0 >
   - L2i misses          :  0.0 >
* data TLB              :  0.0 >
* instruction TLB       :  0.0 >
* branch instructions    :  0.0 >
   - correctly predicted :  0.0 >
   - mispredicted        :  0.0 >
* floating-point instr   :  0.6 >>>>>>>>>>>>
   - fast FP instr       :  0.6 >>>>>>>>>>>>
   - slow FP instr       :  0.0 >

Loop in function WaveToyC_Evolution() at WaveToy.c:73 (41.4% of the total runtime)
================================================================================
```

perfexpert


PerfExpert / AutoSCOPE - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.cs.txstate.edu/~burtscher/research/PerfExpert/AutoSCOPE/
nancy brandt site:et
Home Most Visited Smart Bookmarks Crow/Bond RT Allocations Maps Gmail arXiv.org
Google ... Gmail -... Google ... Diamon... Stevenr... Texas A... Perf... PerfExp... Add Ne...
AutoSCOPE: Automatic Suggestions for Code Optimizations using PerfExpert
PerfExpert Output File to Process: /tmp/perfout.txt Browse...
Compiler: Intel GCC PGI
Result Contents: Include Code Examples Include Compiler
Flags
Go
Done
Red Hat swells sales

perfexpert


PerfExpert / AutoSCOPE - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.cs.txstate.edu/~burtscher/research/PerfExpert/AutoSCOPE/
nancy brandt site:et
Home Most Visited Smart Bookmarks Ctow/Bond RT Allocations Maps Gmail arXiv.org
Google ... Gmail -... Google ... Diamon... Stevenr... Texas A... Perf... PerfExp... Add Ne...
Code Section (Function): Function WaveToyC_Evolution() at WaveToy.c:36 (41.7% of the total runtime)
The performance of this code section is good
Code Section (Loop): Loop in function WaveToyC_Evolution() at WaveToy.c:73 (41.4% of the total runtime)
The performance of this code section is good
Code Section (Function): Function WaveBinaryC() at WaveBinary.c:50 (21.9% of the total runtime)
eliminate common subexpressions involving memory accesses
d[i] = a * b[i] + c[i]; y[i] = a * b[i] + x[i]; -> temp = a * b[i]; d[i] = temp + c[i]; y[i] = temp + x[i];
eliminate floating-point operations through distributivity
d[i] = a[i] * b[i] + a[i] * c[i]; -> d[i] = a[i] * (b[i] + c[i]);
eliminate floating-point operations through associativity
d[i] = (a[i] * b[i]) * c[i]; y[i] = (x[i] * a[i]) * b[i]; -> temp = a[i] * b[i]; d[i] = temp * c[i]; y[i] = x[i] * temp;
Done

Homework

The following questions are based on the wave.cpp and qsort.c programs in the public SVN repository.

1. Debug wave.cpp using Valgrind. What problems did you find?
2. Use profile guided optimization to speed up qsort.c. What speedup do you get?