Performance Engineering of Software Systems September 8, 2011
Massachusetts Institute of Technology 6.172
Professors Saman Amarasinghe and Charles E. Leiserson Handout 2

Getting Started

Project 0

Out: Thursday, September 8, 2011
Due: 11:59 PM on Monday, September 12, 2011

Last Updated: September 8, 2011

This project introduces the environment and tools you will be using to complete your future project assignments. You will experiment with a simple program for multiplying rectangular matrices. You should use this assignment to familiarize yourself with the tools you will be using throughout the course.

NOTE: While you may have tools that you prefer for developing and debugging C programs, we strongly recommend that you build, test, and debug your program on the cloud machines as instructed. Later projects will rely on specific tools that may not work on your personal machines.

1 Get a CSAIL account

If you already have a CSAIL account, then you may skip this step.

You need to sign up for a CSAIL account before you can do anything else. Do so by 8PM on Thursday, September 8, or else you may not be able to access course machines on Friday. The URL to create an account is

```
% http://inquir.csail.mit.edu/signup
```

Choose the "Computer Architecture" group with "Saman Amarasinghe" as your sponsor. It will probably take a few hours before you can log in (since Prof. Amarasinghe must approve your account-creation request).

2 Course machines

You must connect to the `cloud`N`.csail.mit.edu` machines, for $N = 6, 7, \ldots, 15$, to work on and run code. These machines have working compilers and the CQ client. In order to connect to these machines, you will need to use ssh. If you're on a Mac OS X or Linux machine, this is as simple as opening a terminal and typing

```
% ssh username@cloudN.csail.mit.edu
```

which is exactly the same way you remotely connect to Athena. If you're using Windows, however, you'll need to install some software. The free, open-source, and widely used PuTTY is a good choice.

You can edit your files using editors installed in the course machines, or you can setup your personal machine to access files in the AFS and edit them locally using your favorite editor. Follow the instruction to install Kerberos and OpenAFS at

```
% http://tig.csail.mit.edu/twiki/bin/view/TIG/GetStartedAndConnected
```

While you can edit your files anywhere, **we strongly recommend that you build, test, and debug your program on the cloud machines as instructed.**

3 Running the setup script

Your account will need to be configured to use our tools. The necessary magic incantations can be performed by a setup script. Use ssh to connect to one of the cloud machines, and run the following command:

```
% /afs/csail.mit.edu/proj/courses/6.172/scripts/student-setup
```

You must log out and log back in for the setup to be complete. Everything should then work except for CQ, which you need to wait for staff to setup your account.

4 Version control

We shall use the git distributed version control system. The baseline code for this assignment is in the repository

```
% /afs/csail.mit.edu/proj/courses/6.172/student-repos/project0/username
```

If you haven't used git before, please be aware that it's a little bit different than subversion (svn), which you may have used in 6.005.

To make a clone — a local copy of the repository for your work — on a machine without AFS, type:

```
% git clone ssh://username@login.csail.mit.edu/afs/csail.mit.edu/proj/\
courses/6.172/student-repos/project0/username project0
```

To make a clone on a CSAIL machine or other machine with AFS, type:

```
% git clone /afs/csail.mit.edu/proj/\
courses/6.172/student-repos/project0/username project0
```

Edit the code in the project, and when you're done, commit and push your changes back to the repository:

```
% git commit -am 'Your commit message'
git push
```

For more advanced usage of git, please refer to your favorite search engine.

5 Building and running your code

You can build the code by going to the `project0/matrix_multiply` directory and typing `make`.

```
$ make
icc -O3 -DNDEBUG -Wall -m64 -DBUILD_64   -c -o testbed.o testbed.c
icc -O3 -DNDEBUG -Wall -m64 -DBUILD_64   -c -o ktiming.o ktiming.c
icc -O3 -DNDEBUG -Wall -m64 -DBUILD_64   -c -o matrix_multiply.o matrix_multiply.c
icc -o matrix_multiply testbed.o ktiming.o matrix_multiply.o -lrt
```

You can then run it by typing `./matrix_multiply`. The program should print out something, and then crash with a segmentation fault.

6 Using a debugger

While debugging your program, if you encounter a segmentation fault, bus error, or assertion failure, or if you just want to set a breakpoint, you can use the debugging tool `gdb`.

You can start a debugging session in gdb:

```
% gdb --args ./matrix_multiply
```

This command should give you a `(gdb)` prompt, at which you should type `run` or `r`:

```
% (gdb) run
```

Your program will crash, giving you back a prompt, where you can type `backtrace` or `bt` to get a stack trace:

```
...
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f6ed6cca6e0 (LWP 897)]
0x0000000000401229 in matrix_multiply_run ()
Current language:  auto; currently asm
(gdb) bt
#0  0x0000000000401229 in matrix_multiply_run ()
#1  0x0000000000400eaa in main ()
```

This stack trace says that the program crashes in `matrix_multiply_run`, but doesn't tell any other information about the error. In order to get more information, you need to build a "debug" version of the code. First, you quit gdb by typing `quit` or `q`:

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

Next, you build a "debug" version of the code by typing `make DEBUG=1`:

```
$ make DEBUG=1
icc -g -DDEBUG -O0 -Wall -m64 -DBUILD_64 -c testbed.c -o testbed.o
icc -g -DDEBUG -O0 -Wall -m64 -DBUILD_64 -c ktiming.c -o ktiming.o
icc -g -DDEBUG -O0 -Wall -m64 -DBUILD_64 -c matrix_multiply.c -o matrix_multiply.o
icc -o matrix_multiply testbed.o ktiming.o matrix_multiply.o -lrt
```

The major differences from the optimized build are '`-g`' (add debug symbols to your program) and '`-O0`' (compile without any optimizations). Once you have created a debug build, you can start a debugging session again:

```
$ gdb --args ./matrix\_multiply
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f0f141de6e0 (LWP 942)]
0x0000000000401417 in matrix_multiply_run (A=0x2085270, B=0x20851d0, C=0x2085130)
at matrix_multiply.c:77
77          C->values[i][j] += A->values[i][k] * B->values[k][j];
```

Now, gdb can tell that a segmentation fault occurs at at `matrix_multiply.c` line 77. You can ask gdb to print values using `print` or `p`:

```
(gdb) p A->values[i][k]
$1 = 5 (Note: you might see another number here.)
(gdb) p B->values[k][j]
Cannot access memory at address 0x0
(gdb) p B->values[k]
$2 = (int *) 0x0
(gdb) p k
$3 = 4
```

This suggests that `B->values[4]` is `0x0`, which means B doesn't have row 5. There is something wrong with matrix dimensions.

7 Using assertions

The assert package is a useful tool for catching bugs before your program goes off into the weeds. If you look at `matrix_multiply.c`, you should see some assertions in `matrix_multiply_run` routine that check that the matrices have compatible dimensions. Uncomment these lines and a line to include `assert.h` at the top of the file. Then, build and run the program again using gdb. Make sure that you build using `make DEBUG=1`. You will get

```
(gdb) r
...
Running matrix_multiply_run()...
matrix_multiply: matrix_multiply.c:70:
matrix_multiply_run: Assertion ‘A->cols == B->rows’ failed.
[New Thread 0x7f73b314b6e0 (LWP 1024)]

Program received signal SIGABRT, Aborted.
[Switching to Thread 0x7f73b314b6e0 (LWP 1024)]
0x00007f73b257ded5 in raise () from /lib/libc.so.6
```

Now, gdb tells that "Assertion '`A->cols == B->rows`' failed", which is much better than the former "segmentation fault". You might try printing the value of `A->cols`, and fail. The reason is that gdb is not in the stack frame you want. You can get the stack trace to see which frame you want (#3 in this case), and type `frame 3` or `f 3` to move to frame #3. After that, you can print `A->cols` and `B->cols`.

```
(gdb) bt
#0  0x00007f73b257ded5 in raise () from /lib/libc.so.6
#1  0x00007f73b257f3f3 in abort () from /lib/libc.so.6
#2  0x00007f73b2576dc9 in __assert_fail () from /lib/libc.so.6
#3  0x00000000004013d1 in matrix_multiply_run (A=0x1007270, B=0x10071d0, C=0x1007130)
at matrix_multiply.c:70
#4  0x0000000000400f76 in main (argc=1, argv=0x7fffe64ff3f8) at testbed.c:129
```

```
(gdb) f 3
#3  0x00000000004013d1 in matrix_multiply_run (A=0x1007270, B=0x10071d0, C=0x1007130)
at matrix_multiply.c:70
70   assert(A->cols == B->rows);
(gdb) p A->cols
$1 = 5
(gdb) p B->rows
$2 = 4
```

You should see the values 5 and 4, which indicates that we are multiplying matrices of incompatible dimensions.

You will also see an assertion failure with a line number for the failing assertion without using gdb. Since the extra checks performed by assertions can be expensive, they are disabled for optimized builds, which are the default in our Makefile. As a result, if you make the program without `DEBUG=1`, you will not see an assertion failure.

You should consider sprinkling assertions throughout your code to check important invariants in your program, since they will make your life easier when debugging. In particular, most nontrivial loops and recursive functions should have an assertion of the loop or recursion invariant.

Fix `testbed.c`, which creates the matrices, rebuild your program, and verify that it now works. You should see "Elapsed execution time..." after running

```
% ./matrix_multiply
```

Next, you want to check the result of the multiplication. Run

```
% ./matrix_multiply -p
```

The program will print out the result. However, the result seems to be wrong. You can check the multiplication of zero matrices by running

```
% ./matrix_multiply -pz
```

8 Using a memory checker

Some memory bugs do not crash the program, so gdb cannot tell you where the bug is. You can use the memory checking tool `valgrind` to track these bugs.

```
% valgrind ./matrix_multiply -p
```

Note that you need `-p` since valgrind only detects memory bugs that affect outputs. You should also use a "debug" version to get a good result. This command should print out many lines. The important ones are

```
==1116== Use of uninitialised value of size 8
==1116==    at 0x550C953: (within /lib/libc-2.7.so)
==1116==    by 0x550F7D2: vfprintf (in /lib/libc-2.7.so)
==1116==    by 0x5516CB9: printf (in /lib/libc-2.7.so)
==1116==    by 0x40131E: print_matrix (matrix_multiply.c:57)
==1116==    by 0x40100D: main (testbed.c:138)
```

This means the program uses a value before initializing it. The stack trace indicates that the bug occurs in `testbed.c:138`, which is where the program print out matrix C.

Fix `matrix_multiply.c` to initialize values in matrices before using them. Rebuild your program, and verify that it outputs a correct answer.

9 Memory management

In C, you need to free memory after you are done using it. Valgrind can track memory leaks in the program. Run the same valgrind command, you will see these lines at the very end.

```
==1226== LEAK SUMMARY:
==1226==    definitely lost: 376 bytes in 19 blocks.
==1226==      possibly lost: 0 bytes in 0 blocks.
==1226==    still reachable: 0 bytes in 0 blocks.
==1226==         suppressed: 0 bytes in 0 blocks.
```

This suggests that there are memory leaks in the program. To get more information, you can run valgrind again using `--leak-check=full`

```
% valgrind --leak-check=full ./matrix_multiply -p
```

The trace shows that all leaks are from the creations of matrices A, B and C.

Fix `testbed.c` by freeing these matrices after use. Rebuild your program, and verify that valgrind doesn't complain anything.

In your write-up, paste the output from valgrind showing that there is no error in your program.

10 Using CQ

You share the `cloud`*N*`.csail.mit.edu` machines on which you are developing with your fellow students. Directly running and timing the execution on these machines may result in measurement errors due to interference with the jobs of others. To get an accurate timing measure on a dedicated machine, you can use the CQ utility.

Before you can submit your job to CQ, you must give CQ's servers read access to your project directory. To do this, issue the command

```
% fs sa . username.batch rlidwk
```

Note: You will not be able to use CQ until we have created *username.batch* for you.

To submit a job to CQ, just place `cqsub` in front of the command you usually issue. Try submitting your matrix-multiply implementation with:

```
% cqsub ./matrix_multiply
```

If other jobs are in front of you, CQ will notify you that you are queued. After your job runs and completes, its output will be displayed, for example, as below:

```
$ cqsub ./matrix_multiply
job id is 129. output file name is yod-95.
Wed Aug 31 22:26:34 2011 active
   --- yod-95.stdout ---
Elapsed execution time: 0.000001 sec
   --- yod-95.stderr ---
Setup
Running matrix_multiply_run()...
```

11 Compiler optimizations

To get an idea of the extent to which compiler optimizations can affect the performance of your program, increase the size of all matrices to 1000 by 1000. Rebuild your program in “debug” mode and run it. Rebuild it again with optimizations (just type `make`) and run it. Both versions should print timing information, and you should verify that the optimized version is faster.

Report the execution time of both programs in your write-up.

12 Performance enhancements

Now let’s try one of the techniques from the first lecture. Right now, the inner loop produces a sequential access pattern on `A` and skips through memory on `B`.

Let’s rearrange the loops to produce a better access pattern. First, you should run the program as is with optimizations to get a performance measurement. Next, swap the `j` and `k` loop, so that the inner loop strides sequentially through the rows of the `C` and `B` matrices. Rerun the program, and verify that you have produced a speedup.

Report the execution time of the new program in your write-up.

13 Submission

Submit the write-up as described in Section 9, 11 and 12 to the Stellar website. Then, commit and push your changes back to the repository.

```
% git commit -am ’Done!’
git push
```