Lab 7: performance analysis & (more) debugging
Performance question

Add matrices row by row

```
for (i=0; i<N; i++){
    for (j=0; j<M; j++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

Add matrices column by column

```
for (j=0; j<M; j++){
    for (i=0; i<N; i++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
```
many ways to time a program!

- Unix 'time' command
  - Example: time ./my_program
  - Measures execution time of entire program run

- Unix gettimeofday system call
  - Code into program
  - Can time execution of certain portions of the program
gettimeofday

(there are lots of Unix system calls, which do lots of different things)
gettimeofday example

fawcett:330 childs$ cat timings.C
#include <sys/time.h>
#include <stdio.h>

int main()
{
    int num_iterations = 100000000;
    int count = 0;
    struct timeval startTime;
    gettimeofday(&startTime, 0);
    gettimeofday(&endTime, 0);
    double seconds = double(endTime.tv_sec - startTime.tv_sec) +
                     double(endTime.tv_usec - startTime.tv_usec) / 1000000.;
    printf("done executing, took %f\n", seconds);
}
gettimeofday example

fawcett:330 child$ cat timings.C
#include <sys/time.h>
#include <stdio.h>

int main()
{
    int num_iterations = 100000000;
    int count = 0;
    struct timeval startTime;
    gettimeofday(&startTime, 0);
    gettimeofday(&startTime, 0);
    for (int i = 0; i < num_iterations; i++)
    {
        count += i;
    }
    struct timeval endTime;
    gettimeofday(&endTime, 0);
    double seconds = double(endTime.tv_sec - startTime.tv_sec) +
                     double(endTime.tv_usec - startTime.tv_usec) / 1000000.;
    printf("done executing, took %f\n", seconds);
}

fawcett:330 child$ g++ -o2 timings.C
fawcett:330 child$ ./a.out
done executing, took 0.000000
fawcett:330 child$
```c
#include <sys/time.h>
#include <stdio.h>

int main()
{
    int num_iterations = 100000000;
    int count = 0;
    struct timeval startTime;
    gettimeofday(&startTime, 0);
    for (int i = 0 ; i < num_iterations ; i++)
        count += i;
    printf("Count was %d\n", count);  /* NEW LINE OF CODE */
    struct timeval endTime;
    gettimeofday(&endTime, 0);
    double seconds = double(endTime.tv_sec - startTime.tv_sec) +
    double(endTime.tv_usec - startTime.tv_usec) / 1000000.;
    printf("done executing, took %f\n", seconds);
}
```

gmtimeofday example
gettimeofday example

fawcett:330 childs$ cat timings2.C
#include <sys/time.h>
#include <stdio.h>

int LoopFunction(int iteration, int &count)
{
    count += iteration;
}

int main()
{
    int num_iterations = 100000000;
    int count = 0;
    struct timeval startTime;
    gettimeofday(&startTime, 0);
    for (int i = 0; i < num_iterations; i++)
    {
        LoopFunction(i, count);
        /* No longer need this: printf("Count was %d\n", count); */
        struct timeval endTime;
        gettimeofday(&endTime, 0);
        double seconds = double(endTime.tv_sec - startTime.tv_sec) +
            double(endTime.tv_usec - startTime.tv_usec) / 1000000.;
        printf("done executing, took %f\n", seconds);
    }
    fawcett:330 childs$ g++ -02 timings2.C
    fawcett:330 childs$ ./a.out
done executing, took 0.213101
More performance analysis tools

- gprof: old program ... I’m struggling to get it to work
- PAPI: library used widely to capture things like L1 cache misses, stalls, etc
- TAU: full performance analysis infrastructure – made right here at UO
Matrix addition example

– DEMO
Matrix addition example

- Time Cost
  - Strategy A
  - Strategy B
  - Analyzing whether Strategy A or B is more efficient

The reason I am so inefficient
Performance analysis

There will be an assignment for you to add timing information to your image pipeline project.
#1 complaint I hear from employers

• “students can’t debug”
  – If you can debug, you can progress on their projects (even if slowly)
  – If not, they will have to hand hold you

• Think about your debugging approach.
  – How would you describe it during an interview?

This lecture describes how I would answer that question
Debugging Strategy

• (#1) Figure out _where_ the error is occurring
• (#2) Figure out _why_ the error is occurring
• (#3) Form a plan to fix the error
**Terrible debugging strategy**

- fix it by repeatedly making random changes
  - typically, code is pretty close to working in the first place
  - each random change creates a new problem that then has to be debugged
  - code ends up as a complete mess

This is a “bugging” strategy.

Always make sure you feel confident about what the problem is before you start changing code.
Debugging as the scientific method

• Debugging involves the scientific method:
  – You have a hypothesis
    • (“the bug is because of this”)
  – You form an experiment
    • (“I will test my theory by examining the state of some variables”)
  – You confirm or deny the hypothesis
    • (“the state was OK, my hypothesis was wrong”)
    • (“the state was bad, my hypothesis was right”)
Backups

• The “scientific method” of debugging – which is good – can leave your code as a mess

• My recommendation:
  – when you have a bug, immediately make a copy of your program
  – apply the scientific method-style of debugging until you understand the problem and how to fix
  – then go back to your original program and make the fix there
Debugging Overview

• To me, effective debugging is about two things:
  – Challenging your own assumptions
  – Divide-and-conquer to find the problem
Challenging Assumptions

• you thought the code would work and it doesn’t
  – so something you did is wrong, and you have to figure out what
• I find students are often turning under the wrong stones, since there are some stones they don’t see
• the way to find these “hidden stones” is to start with the bad outcome and search backwards
  – why is this pointer NULL? why is this value bad?
Divide-and-Conquer

• There are lots of things that could be wrong with your program
  – This is a search problem!!

• Divide-and-Conquer: try to focus on hypotheses that will eliminate half the possibilities, no matter what the outcome is
Divide-and-Conquer

• “Halving” hypotheses:
  – The problem is in this module
  – The problem is in this function

• Non-halving hypotheses:
  – The problem is in this line of code
  – The problem is with this variable

As you divide the space smaller and smaller, you will eventually end up hypothesis that are small in scope
Good practice / Poor practice

• Good practice:
  – Write a few lines of code and test that it does what you want

• Poor practice:
  – Write a bunch of code and compile it all at the end

Why is it better to write smaller portions of code at a time?
Why is it better to write smaller portions of code at a time?

• If you have one bug
  – it is easier to figure out where that bug is
    • searching through tens of lines of code, not hundreds

• If you have many bugs
  – this is a disaster and you end up chasing your tail
    • and you are still searching through hundreds of lines of code, not tens

The extra effort of modifying the main program to test your new code pays off ten-fold (WAG)
Final thought: always initialize your variables!!

• Many crashes in your HW assignments due to uninitialized pointers.
  – If you assign it to NULL, then at least you know that is set to something not valid
  – Otherwise, you see an address and think it might have valid memory

• Initialize non-pointers too!

• Classic point that employers look for.

This practice becomes increasingly essential when you work in teams on large projects.
Print statements

• Print statements (cerr, printf):
  – the world’s most used debugger
  – very successful SW professionals are able to debug large SW projects using only print statements

• Pitfalls:
  – output buffering
  – too many print statements
Pitfall #1: output buffering

- output is sometimes buffered, and buffered output is dropped when a program crashes
- if you don’t see a print statement
  - your program may have crashed before it got to that print statement
  - your program may have gotten to that print statement, but crashed before it got outputted

`cerr: automatically flushes & mostly prevents this problem`

`cerr << "*(NULL) = " << *NULL << endl; // still doesn’t work`
Output buffering and cerr

• cerr: automatically flushes & mostly prevents output buffering problem

• Exception:
  – cerr << "*(NULL) = " << *NULL << endl;
    • (crashes before anything is printed out)

• Work-around:
  – cerr << "*(NULL) = ";
  – cerr << *NULL << endl;
Pitfall #2: millions of print statements

```c
void make_black(unsigned char *b, int width, int height, int buffer_size)
{
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            int pixel_index = j*width+i;
            int buffer_index = 3*pixel_index;
            cerr << "About to write to index"
                 << buffer_index << endl;
            b[buffer_index+0] = 0;
            b[buffer_index+1] = 0;
            b[buffer_index+2] = 0;
        }
    }
}
```

This will result in millions of print statements ... hard to debug.
Reducing print statements

```c
void make_black(unsigned char *b, int width, int height,
                 int buffer_size)
{
    for (int i = 0 ; i < width ; i++)
    {
        for (int j = 0 ; j < height ; j++)
        {
            int pixel_index = j*width+i;
            int buffer_index = 3*pixel_index;
            if (buffer_index < 0 || buffer_index >= buffer_size)
            {
                cerr << "About to write to index"
                    << buffer_index << endl;
                exit(EXIT_FAILURE);
            }
            b[buffer_index+0] = 0;
            b[buffer_index+1] = 0;
            b[buffer_index+2] = 0;
        }
    }
}
```
Make it easy on yourself...

#include <iostream>

using std::cerr;
using std::endl;

#define PL cerr << "(PL): " << __FILE__ << ": " << __LINE__ << endl;

int main()
{
    PL
    int width = 100, height = 100;
    PL
    int buffer_size = width*height;
    PL
    unsigned char *b = new unsigned char[3*buffer_size];
    PL
    for (int i = 0; i < width; i++)
    {
        PL
        for (int j = 0; j < height; j++)
        {

128-223-223-73-wireless:330 hank$ ./a.out
(PL): big_print.C: 10
(PL): big_print.C: 12
(PL): big_print.C: 14
(PL): big_print.C: 16
(PL): big_print.C: 19
About to write to index10200
Make it easy on yourself ...

- If it takes you two minutes to type in the commands to compile, execute, and inspect the output for your program, then you probably won't test it enough.
- Remember stuff like 'alias', ssh config, scp, shell scripting, etc ...
- If you can run tests quickly, you'll do them more often and save yourself tons of debugging time!
Debuggers

- Allow you to set breakpoints
- Allow you to inspect variables
- Show you where a program crashed
Debuggers

• gdb:
  – See Lab 1 notes for tips on using gdb!

• valgrind:
  – See Lab 1 notes for tips on using valgrind to find memory errors (segfault)!
  – TIL: it's pronounced val - grin - ed
Valgrind: a memory checker

hank@ix: ~ 14$ valgrind a.out
==13623== Memcheck, a memory error detector
==13623== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13623== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13623== Command: a.out
==13623==
==13623== Invalid read of size 4
==13623== at 0x4004C4: main (myprogram.C:5)
==13623== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==13623==
==13623== Process terminating with default action of signal 11 (SIGSEGV)
==13623== Access not within mapped region at address 0x0
==13623== at 0x4004C4: main (myprogram.C:5)
==13623== If you believe this happened as a result of a stack
==13623== overflow in your program's main thread (unlikely but
==13623== possible), you can try to increase the size of the
==13623== main thread stack using the --main-stacksize= flag.
==13623== The main thread stack size used in this run was 8388608.
==13623==
==13623== HEAP SUMMARY:
==13623== in use at exit: 0 bytes in 0 blocks
==13623== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13623==
==13623== All heap blocks were freed -- no leaks are possible
==13623==
==13623== For counts of detected and suppressed errors, rerun with: -v
==13623== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
Segmentation fault (core dumped)
Valgrind

• There will be an assignment to have a memory error-free and memory leak-free program. Use valgrind!
Valgrind: a memory (leak) checker

• To see details, call valgrind with the flag:
  --leak-check-full=yes

• Might be really slow!

• Tells you the number of heap allocations and frees ... should be the same

• Definitely lost = you goofed

• Indirectly lost = fix the definitely lost parts

• Possibly lost = nonstandard situations

• Still reachable = just need to free
Valgrind: memory leak checker

- Issue you may run into:
Valgrind: memory leak checker

• Issue you may run into: