Problem set 3, Debugging & Optimization

TDT4200, Fall 2015

Deadline: 24.09.2015 at 32:59. Contact course staff if you cannot meet the deadline.

Evaluation: Pass/Fail

Delivery: Use It's Learning. Deliver exactly two files:

- *yourNTNUusername_ps3.pdf*, with answers to the code questions
- *yourNTNUusername_code_ps3.*{*zip* |*tar.gz* |*tar*} containing your modified versions of the files:
  - `Makefile`
  - `newImageIdea.c`
- Do *not* include any image files.
- Add only the files and not the folder to the archive.

The unmodified ppm.c and ppm.h files can be indluded.

General notes: Code must compile and run on the following systems:

1. `its015-01.idi.ntnu.no`
2. `Problem_set_3` in the *TDT4200_H2015* group on `climb.idi.ntnu.no`.

You should only make changes to the files indicated. Do not add additional files or third party code/libraries.

Part 1, Theory

Problem 1, General Theory

There are no theory questions. Spend the time on the code, you need it!

Part 2, Code

There are two tasks in this assignment. One is debugging the other is optimization. The debugging task is quite easy, but requires some time to perform. It counts for a smaller part of this exercise. The optimization task can take a *long* time to solve in a good way.

Write short answers. Just point at the key problem/answer if possible. The staff does not have time to read long answers. *Sorry.* Quality over quantity.

Problem 1, Debugging

For this task use the files `test01.c`, `test02.c` and `reverseParams.c` from the `debug_handout_ps3.zip` archive. Note that different compilers will behave differently. Your local system might give other bugs, but this is still fine.

a) Find problems with the program 'test01.c':

   i) Compile: 'gcc -g test01.c'
   ii) Run with Valgrind: 'valgrind ./a.out'
   iii) And then try: 'valgrind −−leak-check=full ./a.out'

   Find the bug(s). Only use *one line answers* for each bug you find.

b) *Optional task:* Find problems with the program 'test02.c':

i) Compile: 'gcc -g test02.c'

ii) Run with Valgrind: 'valgrind ./a.out'

iii) And then try: 'valgrind −−leak-check=exp-dhat ./a.out'

Make sense of the output. If it is unclear try to add code that access 'mem' like 'mem[112] = 42;', and look at the new output of Valgrind. Only a short answer is needed (optional task).

c) *Main task:* Find problems with the program 'reverseParams.c'. This exercise consists of two parts - *illustrating* strange bugs and *hunting* them down. Before running and compiling a quick look at the code is advised. Read the code documentation (comments) only and make up your mind on how the program works.

   i) Compile with all combinations:
   ```
   gcc -g reverseParams.c
   gcc -g -O3 reverseParams.c
   gcc -g -Wall reverseParams.c
   gcc -g -Wall -O3 reverseParams.c
   gcc -g -Wall -O1 reverseParams.c
   ```
   Note that -Wall enables 'all' warning messages except the ones enabled by -Wextra.

   ii) Compile with 'gcc -g reverseParams.c' and test:
   ```
   ./a.out 12345
   ./a.out 12345 qwer asd zx c
   ./a.out 1234567890ABC qwer asd zx c
   ./a.out 1234567890ABC "q w e r" asd
   ```
   The code really works.

   iii) Now try this:
   ```
   ./a.out X 1234
   ./a.out X 12345678
   ./a.out X 123456789
   ./a.out X 1234567890
   ./a.out X "" 1234567890
   ./a.out X 1234567890 ""
   ```
   Observe the output.

   iv) Compare:
   ```
   ./a.out "" 1234567 qwertyuiopa
   ./a.out "" 1234567 ASDFGHJKL qwertyuiopa
   ```
   Optional task: How do you explain this behaviour. Short answer please.

   v) Recompile with 'gcc -O3 -g' and again do the tests:
   ```
   ./a.out X 123456789
   ./a.out X 1234567890
   ./a.out X "" 1234567890
   ./a.out X 1234567890 ""
   ```
   Optional task: How did the program behave with the '-O3' flag? Short answer please.

   vi) Try testing *with* and *without* Valgrind using 'gcc -g'. Note that valgrind adds its own messages '==12345== ....' to the output with the original output in between unless you redirect it to a log file:
   ```
   ./a.out X 1234567890abcdefg
   valgrind --log-file=val.txt ./a.out X 1234567890abcdef
   valgrind ./a.out X 1234567890abcdef
   ```

   vii) Other things to try:
   ```
   valgrind --log-file=val.txt --malloc-fill='41' ./a.out 1 2 3 4 5
   ```
   Remember to look at the 'val.txt' text file when you use the log file.

Use the *illustrations* as a guide to find the bug(s). Write a *point list* of bugs, errors, issues and comments. *None* will find all the bugs, just try to list some. *One line* answers are acceptable. Write the relevant line number at the start of each line. Example:
`Line 59:  "No newline at end of file".  And it ends with a comment too!`
If you want you can look at this link to read why this is bad: http://stackoverflow.com/questions/72271/no-newline-at-end-of-file-compiler-warning

Problem 2, Optimization

> "We have just made an amazing new image processing algorithm. This will change everything! Now we need you to optimize the code massively. The algorithm developers need a fast version on their workstations. In addition, the product department believes it will be too slow for our mainstream ARM based devices. I have already told them it you can solve it. You know the presentation to the investors is on next Monday, so get to it."
> – Your boss

This task is real. It is not a toy, example or training task. Solving this in a *very* good way may have real commercial value. For this task use the files `Makefile`, `ppm.c` and `newImageIdea.c` from the `optimize_handout_ps3.zip` archive. You must also download the `flower.ppm` file and include it in the same directory as the code.

The `Makefile` handed out with this Problem set creates an executable named "`newImageIdea`". The program reads the `flower.ppm` image and creates 3 new images: `flower_tiny.ppm`, `flower_small.ppm` and `flower_medium.ppm`. Use a image program to look at the images.

`newImageIdea` implements a naive approach to solving the problem. The code is *bad*. Assume that someone that hardly knows programing wrote the code.

You may modify the given `Makefile` if you think it helps your performance. Make sure that the existing `make newImageIdea` rule still works if you do.

a) Somehow optimize the code significantly *without* using MPI/OpenMP/GPU or parallelization. This is your only program code delivery.

- There are several (simple) changes that will improve performance.
- However, you should think about the following:
  - Caches and access patterns. You may look in the book for hints.
  - Useless code.
  - Branches and data layout/usage.
  - Compiler flags.
  - The amount of operations performed by your program.
- There are many different approches to better performance, and several can be combined.
- Also, when working on real problems the answer can be open for interpretation:
  - Can you solve the problem in a different way?
  - How much precision is needed?
  - Is pixel accuracy needed?
  - Reusing data/math?
- You are allowed to have some minor pixel errors in the final output(s). Each pixel color is in the range of 0-255.
  - A few hundred ± 1 differences is fine.
  - Less than one hundred larger differences is fine.
  - It might be wize to make a small program to chech for pixel errors. Visual verification can be tricky.

b) The code must run on its015-XX.idi.ntnu.no (XX being any of the lab machines in ITS015).

c) The code must run on the CMB system `climb.idi.ntnu.no`. It will be tested for energy efficiency, and must pass an automated test. Do not run on CMB until you have optimized the code. Note: There is a timeout after 90 seconds.

d) If the output from your program is not identical to the handout code you must write this in the answer.

e) Analyze your implementation and report the following: The number of `sum += imageIn->data[...` operations performed by your program a single call of the 'performNewIdeaIteration' function. Use $size = 2$ and $size = 8$ with the `flower.ppm` image as parameters.

Finally, if you make the program really fast you we have a bonus for you. By demonstrating your knowledge here you can *skip* some selected parts in an exercise later on. Other rewards are possible as well.

Additional details can be found in the recitation slides for this Problem set.