Problem set 3, Serial Optimization

TDT4200, Fall 2013

Deadline: 03.10.2013 at 22.00 Contact course staff if you cannot meet the deadline.

Evaluation: Pass/Fail

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

- *yourusername_ps3.pdf*, with answers to the theory questions
- *yourusername_code_ps3.*{*zip* |*tar.gz* |*tar*} containing your modified versions of the files:
  - *fast.c*

General notes: All problem sets are to be done **INDIVIDUALLY**. Code must compile and run on `clustis3.idi.ntnu.no`. You should only make changes to the files indicated. Do not add additional files or thrid party code/libraries.

Part 1, Theory

Problem 1, Optimization

a) Explain why branches in code can be a performance problem.

b) What is spatial and temporal locality?

c) How does caches make use of spatial and temporal locality to increase performance?

Problem 2, Caching

In this problem, we'll examine the code in the provided file *cache.c*. The code contains two functions, both of which perform the same calculations, but using datastructures with different memory layouts.

a) Measure the performance of the two functions (comment out one function to test the other). (Use the time command to measure the running time, e.g. `time ./cache`).

b) Explain why the memory layout of the best performing function is advantageous in this case.

Problem 3, Basic Valgrind

In this problem you should find problems with the program *test01.c* using Valgrind. Write a list of bugs, errors and issues you find in the code.

The following instructions can be helpful to achieve this.

Compile with:

```
gcc-4.6 -g test01.c
```

Run with Valgrind with:

```
valgrind ./a.out
```

More information can be found with:

```
valgrind --leak-check=full ./a.out
```

Problem 4, Intermediate Valgrind

In this problem you should find problems with the program *test02.c* using Valgrind. Write a list of bugs, errors and issues you find in the code.

The following instructions can be helpful to achieve this. The questions are only there to help you find bugs, and should not be answered directly, you should only deliver the list of bugs.

Compile with:

```
gcc-4.6 -g test02.c
```

Run with Valgrind with:

```
valgrind --tool=exp-dhat ./a.out
```

What does the output mean? What about 'Aggregated access counts'? If it is unclear try to add code that access `mem` like `mem[112] = 42;`, and look at the new output of Valgrind.

Problem 5, Intermediate Valgrind

In this problem you should find problems with the program *test03.c* using Valgrind. Write a list of bugs, errors and issues you find in the code.

The following instructions can be helpful to achieve this. The questions are only there to help you find bugs, and should not be answered directly, you should only deliver the list of bugs.

Before running and compiling a quick look at the code is advised. Read the documentation only and make up your mind on how the program works.

Compile with all combinations:

```
gcc-4.6 -g test03.c
gcc-4.6 -g -O3 test03.c
gcc-4.6 -g -Wall test03.c
gcc-4.6 -g -Wall -O3 test03.c
gcc-4.6 -g -Wall -O1 test03.c
```

Note that `-Wall` enables 'all' warning messages except the ones enabled by

`-Wextra`.

Compile with `gcc-4.6 -g` 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
```

Was there any point in the comments GCC made using `-Wall`? Look at the code again and note that it really works, even the `malloc/free` is correct.

Try:

```
./a.out X 1234
./a.out X 12345678
./a.out X 123456789
./a.out X 1234567890
./a.out X "" 1234567890
./a.out X 1234567890 ""
```

Compare:

```
./a.out "" 1234567 qwertyuiopa
./a.out "" 1234567 ASDFGHJKL qwertyuiopa
```

Recompile with `gcc-4.6 -O3 -g` and test:

```
./a.out X 123456789
./a.out X 1234567890
./a.out X "" 1234567890
./a.out X 1234567890 ""
```

Did the program behave the same using `-O3`? Try test with and without Valgrind using `gcc-4.6 -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 1234567890abcdefg
```

What is the output, and why (no searching, just think/guess)? Then try with:

```
valgrind --track-origins=yes --log-file=val.txt ./a.out X 1234567890abcdefg
```

Other things to try:

```
valgrind --log-file=val.txt --malloc-fill='41' ./a.out 1 2
3 4 5
```

It might be easier to see when the errors occur without the `--log-file` option, and link it to what happens in the program.

Problem 6 Advanced Valgrind

In this problem we'll look at the NAS Parallel Benchmarks (`www.nas.nasa.gov/publications/npb.html`) with Valgrind.

As with the previous problems, you should create a list of problems, issues and errors, but only a few are required, and they may be on a higher level than for the previous problems. In addition you should comment on the following questions: Is this benchmark well written? How optimal is the memory usage? Keep in mind that this code is complex, large (in this setting+time), probably production quality and tested by many people.

The following instructions expains how to set up the code. Start by extracting the NAS files:

```
tar -xvf NPB3.3.1.tar.gz
```

Copy the file *make.def* to NPB3.3.1/NPB3.3-SER/config/:

```
cp make.def NPB3.3.1/NPB3.3-SER/config/
```

Make the 'DC' benchmark:

```
cd NPB3.3.1/NPB3.3-SER
make dc CLASS=S
```

Now the binary is in the bin directory, and can be executed with:

```
./bin/dc.S.x
```

You can use one or both of the following approaches to get information about the code. The first might be easier but require that you install KCachegrind or (Qcachegrind on Windows). If you don't want to install this program, you can use the other option. Each option reveals different issues, however.

Option 1, Make a call graph of the 'DC' benchmark

If Valgrind and the viewer are on different systems you might need to manually add source code directories: In settings->configure Kcachegrind->Annotations, add the DC source code folder, eg: /Downloads/NPB3.3.1/NPB3.3-SER/DC

Use the following command:

```
valgrind --tool=callgrind --branch-sim=yes --collect-jumps=yes
--cache-sim=yes --cacheuse=yes --simulate-hwpref=yes --dump-instr=yes
./bin/dc.S.x
```

Run Kcachegrind/Qcachegrind, load the *callgrind.out.xxxxx* file and look at the

code flow. Click around the graph/maps (start at main) and try to find the time consuming parts in the code. What do they perform (look at the function names only)?

Option 2, Test 'DC' with various Valgrind tools

Use the following commands:

```
valgrind ./bin/dc.S.x
valgrind --tool=exp-dhat ./bin/dc.S.x
valgrind --tool=massif ./bin/dc.S.x
```

With the massif tool use `ms_print` to view the output file:

```
ms_print massif.out.12345 | less
```

Find the peak total(B), useful-heap(B) and extra-heap. Use the other options and knowledge you have gained of Valgrind as well.

Part 2, Code

Problem 1, gemm

In this problem you should write a funtion that performs general matrix multiplication as fast as possible. The provided file *fast.c* currently contains a naive implementation, your task is to rewrite the function to improve the performance as much as possible.

Additional neccesary details can be found in the recitation slides for this problem set