THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 3

Due: 11:59pm Tuesday, 20 May 2014

This assignment is worth 10% of your final assessment

Task description

In this assignment we will implement a simplified vector processing language that is able to construct vectors and perform vector computations in the C programming language. The aim is to use a variety of parallel programming, algorithmic and code optimisation techniques to achieve peak performance.

You have been provided with thoroughly documented skeleton code that is incomplete and somewhat works quite slowly. Your task is to implement the incomplete functions and then analyse the running time of the operations under various inputs with an aim to make it as fast as possible. Most of vector operations can be improved using parallelisation, other algorithms or perhaps both. It is up to your to determine whether you will focus on tuning certain operations that are dramatically slower than the others, or to focus on each operation equally. Be sure to cache vectors and results where possible.

Fetching the assignment files

To obtain the files required for this assignment, run the following command on **ucpu1**

```
$ /labcommon/comp2129/bin/get-assignment3
```

Hint: Develop and run your code on the undergraduate server to access the included testing tools.

Academic declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Information Technologies, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of IT for the purpose of future plagiarism checking.

Implementation details

The contents of the header file **`vector.h`** are shown below:

```c
#ifndef VECTOR_H_
#define VECTOR_H_

#include <stdint.h>

/* utility functions */

int64_t fast_rand(void);

void set_seed(int64_t value);
void set_length(int64_t value);

/* vector operations */

int64_t* new_vector(void);
int64_t* prime_vector(int64_t start);
int64_t* random_vector(int64_t seed);
int64_t* uniform_vector(int64_t value);
int64_t* sequence_vector(int64_t start, int64_t step);

int64_t* cloned(int64_t* vector);
int64_t* reversed(int64_t* vector);
int64_t* ascending(int64_t* vector);
int64_t* descending(int64_t* vector);

int64_t* scalar_add(int64_t* vector, int64_t scalar);
int64_t* scalar_mul(int64_t* vector, int64_t scalar);

int64_t* vector_add(int64_t* vector_a, int64_t* vector_b);
int64_t* vector_mul(int64_t* vector_a, int64_t* vector_b);

/* compute operations */

int64_t get_sum(int64_t* vector);
int64_t get_mode(int64_t* vector);
int64_t get_median(int64_t* vector);
int64_t get_minimum(int64_t* vector);
int64_t get_maximum(int64_t* vector);
int64_t get_element(int64_t* vector, int64_t index);
int64_t get_frequency(int64_t* vector, int64_t value);

void display(int64_t* vector, int64_t label);

#endif
```

Warning: Ensure that you do not modify the included Makefile.

Program input

The program input consists of three sections. The first defines the program settings. The second defines the vectors. The third defines the computations. The structure of the input is shown below.

You can assume only valid input will be tested. Any commands that use vectors are guaranteed to have them defined beforehand. All vectors and elements will be indexed from 0. All vectors in a test will have the same length defined by $1 \leqslant length \leqslant 1,000,000,000$. Vector elements can have values up to the maximum value of **`int64_t`**. Our tested commands will not cause any integer overflows.

```
>>> Settings

<# cores> <# threads>
<vector length> <# vectors> <# computations>

>>> Vectors :: vector= <function> [args]

vector= random <seed>
vector= primes <start>
vector= uniform <value>
vector= sequence <start> <step>

vector= reversed <vector index>
vector= ascending <vector index>
vector= descending <vector index>

vector= scalar#add <vector index> <value>
vector= scalar#mul <vector index> <value>

vector= vector#add <vector a index> <vector b index>
vector= vector#mul <vector a index> <vector b index>

>>> Computations :: compute <function> [args]

compute sum <vector index>
compute mode <vector index>
compute median <vector index>
compute minimum <vector index>
compute maximum <vector index>
compute frequency <vector index> <value>

compute display <vector index>
compute element <vector index> <element index>
```

Program output

Simply output the result of each computation in the order they are given. Refer to the sample output.

Use the provided format strings to output **`int32_t`** and **`int64_t`**

The Makefile

We have provided you with a Makefile that can be used to compile and test your code.
The included Makefile will only run correctly on the **`ucpu1`** undergraduate server.

- **`$ make`**
  will compile your source files into an executable file **`scaler`**
- **`$ make rain`**
  will compile your program and check whether your program has any memory leaks using the test cases that you define in your **`tests/`** directory. If any test cases require investigation then run valgrind on each test, e.g. **`$ valgrind ./scaler < sample.in`**
- **`$ make test`**
  will compile your program and check the correctness of your program against the input and output files that you created in your **`tests/`** directory. If any test cases require investigation then run sdiff on each test, e.g. **`$ sdiff sample.obs.out sample.exp.out`**
- **`$ make submission`**
  will submit your program to the benchmarking queue and also for marking

Hint: This assignment will be discussed in the Week 9 lecture, make sure you attend!

Marking

We have provided a set of test cases in the included **`tests/`** directory, it is important that you thoroughly test and benchmark your code by extending the set of test cases you are provided with. In this assignment only **valid inputs** are tested, and your program will be checked for memory leaks. If your program leaks memory you will not pass the test case even if the output is correct. In addition to the provided cases we will run your program against a substantial collection of hidden cases.

In this assignment correctness and performance are equally weighted. However, your submission will only receive marks for the performance component if it passes all of the correctness test cases.

5 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will use our own hidden test cases that cover every aspect of the specification. To pass test cases your solution must produce the correct output within the imposed time limit.

5 **marks** are assigned based on the *performance* of your code relative to other students.
This component is tested on a separate machine in a consistent manner. The fastest submission will receive 5 marks, with successively slower solutions receiving lower marks. Any student who is faster than our basic parallel reference implementation will receive at least 2.5 marks.

Warning: Any attempts to deceive the automatic marking will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specification, or your code is unnecessarily or deliberately obfuscated.