Asynchronous Elephant

### Due Date: November 21, 2014 @ midnight Submission Subject: “Elephant”

How do you eat an elephant?  *One bite at a time*!

## General Submission Criteria:

* See Lab 0 for the General Submission Criteria!
* Make a directory in your repository: lab7
* Include all of your Lab7 work within the lab7 directory

## Overview:

In this lab, you will create a program that performs perform matrix addition. Within this program, however, you will perform the operation in blocks (aka bites). Moreover, you will read the blocks from a text file asynchronously.

## Executable Names:

**matrix\_gen**: a program that creates a file of integers that represents a NxN matrix.

**matrix\_add:** a program that performs matrix addition on a given matrix, and emits the amount of time to perform the operation on standard error.

NOTE: The file format of the text file for your matrix needs to be understood by both the matrix\_gen and matrix\_add programs.

Usage: matrix\_gen size >matrix\_file.size

Generates a matrix of size “*size* x *size”* that contains a random set of integers from -100 .. 100.

Usage: matrix\_add size blocks < matrix\_file.size-1 >matrix\_file.size-2

Performs matrix addition on an integer matrix with a random scalar value.

The operation is performed on the matrix in bites of size “*size* x *size*”.

## Makefile Targets:

all: (default)

matrix\_gen:

matrix\_add:

clean:

## General Program Layout:

The general layout for the matrix\_add is provided below. Note that within this code the read and write of each operation is performed asynchronously. Within this code layout, you don’t receive any benefit from using asynchronous I/O.

|  |  |
| --- | --- |
| matrix\_add: block, size, scalar  for i = 1 .. size  for j = 1 .. size  block[i][j] += scalar;  end for  end for  main: size, blocks  start\_time = get current time;  scalar = random number;  block\_size = size / blocks;   |  | | --- | | for x = 1 .. blocks  for y = 1 .. blocks  async read request matrix[x, y]  block = async read return matrix[x, y]  matrix\_add(block, block\_size, scaler)  async write request block  async write return block  end for  end for |   end\_time = get the current time  emit end\_time - start\_time |

Once you have your programming working with the above code layout, you should enhance your project to move the asynchronous request and return calls further apart. (Note that you can collapse the x & y loop into one, so that the program layout conforms to the revised layout with asynchronous operations. This means that your block size will be 1 x (“block\_size” \* “block\_size”). To effectively perform this step, you should consider operating in a pipeline fashion. Within the pipeline, you will have three (3) blocks that are being managed by your program.

1. **last**: the last block that was processed and now needs to be written to the filesystem
2. **current**: the current block being processed by the matrix\_add procedure
3. **next**: the next block to be read in from the input file.

You can both pipeline the read operation and the write operation. To reduce confusion, the following layout provides you with the operation semantics of only performing the read operation in a pipeline fashion.

|  |  |
| --- | --- |
| /\* First prime the pump by reading the first block \*/  async read request current  async read return current  for current = 0 .. ( (block\_size \* block\_size ) - 2) {  /\* Note the blocks are number zero relative \*/   |  | | --- | | last = current - 1;  next - current + 1;  async read request **next**  /\* see aio\_read(2) \*/  matrix\_add(current, block\_size, scalar)  async write request **last** /\* see aio\_write(2) \*/  async write return **last**  /\* see aio\_return(2) \*/  memcpy current → last /\* see memcpy(3) \*/  async read return **next**  /\* see aio\_return(2) \*/  memcpy next → current |   /\* Last drain the pump by handling the last block \*/  matrix\_add(current, block\_size, scalar)  async write request **last**  async write return **last** |

To get further performance improvements, you can move the “async write return” call to the end of the loop. Greater performance improvements -- at the cost of code clarity -- can be achieved by pipeline this call.

**Running some tests:**

Now that your program works, you can execute it with various block sizes. First, obtain the size of a page and the maximum range of integer. Based upon this information, you can calculate the number of integers per page.

$ getconf PAGE\_SIZE

4096

$ getconf UINT\_MAX

4294967295

2^32 = 4294967295

32 / 8 = 4 bytes per integer

4096 / 4 = 1024 integers per page. (1K)

Now run your program with various block sizes, and you will see the performance impact on demand paging. Here I have selected a matrix size of 10K x 10K (10240 x 10240)

$ matrix\_add 10240 20 < matrix\_file.size-1 > matrix\_file.size-2

# hence block is ½ a page

$ matrix\_add 10240 10 < matrix\_file.size-1 > matrix\_file.size-2

# hence a block is 1 page

$ matrix\_add 10240 5 < matrix\_file.size-1 > matrix\_file.size-2

# hence a block is 2 pages

$ matrix\_add 10240 2 < matrix\_file.size-1 > matrix\_file.size-2

# hence a block is 5 pages

## 

## Caveats and Exceptions:

## Checkout out the input values to ensure that everything divides evenly. If it does not, you can emit an error message and exit the program.

## See Also:

http://en.wikipedia.org/wiki/Matrix\_(mathematics)#Addition.2C\_scalar\_multiplication\_and\_transposition