Homework 3

Xinyuan Miao
020033910009
mxinyuan@sjtu.edu.cn

Exercise 5.9

Main Idea

To parallelize the Sieve of Eratosthenes program based upon a functional decomposition of the algorithm, every processor need to identifies some 'seed primes', picking some of the seeds in a interleave fashion for the further sieve. Unlike the domain decomposition, search for the 'seed primes' is not only the first processor's task.
Each processor then do the sieve based on the seed primes belonging to itself and form a local marked table. After that, all local marked tables will be sent to the first processor to do the OR-reduction and finally, the processor-0 calculate the number of the primes.

Execution Results

```
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpicxx -o ex5_9 ex5_9.c
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpirun -np 3 ./ex5_9 10000
Finished!
1229 primes are less than or equal to 10000
Total elapsed time:   0.009555
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpirun -np 3 ./ex5_9 1000000
Finished!
78498 primes are less than or equal to 1000000
Total elapsed time:   0.036033
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpirun -np 4 ./ex5_9 1000000
Finished!
78498 primes are less than or equal to 1000000
Total elapsed time:   0.081872
```

Figure 1: Execution result

Exercise 6.10

Main Idea

Let the first processor execute the function *MPI_Send* while the others execute the function *MPI_Recv*.

Execution Results

In the testing program, we create an array with the size ranging from 100 to 100000000. The output shows that there's no much difference on performance between the two implementation of the broadcast.

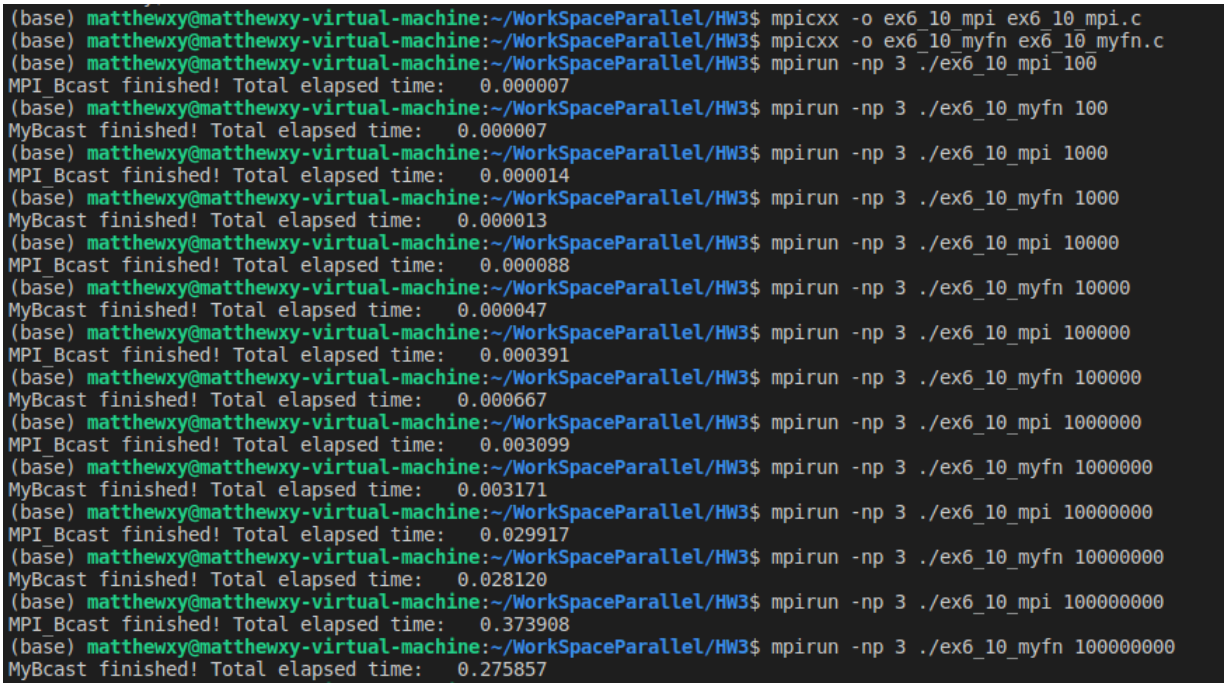

Figure 2: Execution result

Exercise 6.13

Main Idea

Using the domain decomposition, each processor is assigned one task that is to update the status of the elements in several consecutive rows. Specifying 1 denote **alive** and 0 denote **dead**, we can update any element's status based on itself and the current status of the neighbours.
Notice that the update to the elements, some of whose neighbors are in another processor requires the communication among processors. In each iteration, each processor first sends the border elements to the processors that will get use of these elements later. For example, the processor-i will send its first row and last row to the processor-$(i-1)$ (if exists) and the processor-$(i+1)$ (if exists) respectively. Then, each processor wait for the elements it require from other processors. After that, the processor just do the update to the local elements.

Execution Results

We provide a program to generate the binary file containing the matrix that will be used as the input.
We utilize some functions provided by the book such as some *read* and *print* functions and implement the core algorithm by our own *getNewStatus* and *play* functions.

```
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpirun -np 2 ./ex6_13 input_Matrix
Command line: ./ex6_13 <m>
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ gcc ex6_13_genMatrix.c -o ex6_13_genMatrix
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ ./ex6_13_genMatrix
Generation completed!
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpicxx -o ex6_13 utils.c ex6_13.c
utils.c: In function 'void print_row_striped_matrix(void**, MPI_Datatype, int, int, MPI_Comm)':
utils.c:82:35: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
   82 |                     b[i] = b[i-1] + n * datum_size;
      |                            ~~~~~~~^~~~~~~~~~~~~~~
utils.c: In function 'void read_row_striped_matrix(char*, void***, void**, MPI_Datatype, int*, int*, MPI_Comm)':
utils.c:168:14: warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]
  168 |         rptr += *n * datum_size;
      |         ~~~~~^~~~~~~~~~~~~~~~~~
utils.c: In function 'int get_size(MPI_Datatype)':
utils.c:26:1: warning: control reaches end of non-void function [-Wreturn-type]
   26 | }
      | ^
(base) matthewxy@matthewxy-virtual-machine:~/WorkSpaceParallel/HW3$ mpirun -np 2 ./ex6_13 input_Matrix 3 1
Initial state:
     1     0     0     1     1
     0     0     0     1     1
     0     0     0     0     1
     1     1     1     0     1
     0     0     0     0     1

After 1 iteration(s):
     0     0     0     1     1
     0     0     0     0     0
     0     1     1     0     1
     0     1     0     0     1
     0     1     0     1     0

After 2 iteration(s):
     0     0     0     0     0
     0     0     1     0     1
     0     1     1     1     0
     1     1     0     0     1
     0     0     1     0     0

After 3 iteration(s):
     0     0     0     0     0
     0     1     1     0     0
     1     0     0     0     1
     1     0     0     0     0
     0     1     0     0     0

Finished! Total elapsed time:   0.000129
Final state:
     0     0     0     0     0
     0     1     1     0     0
     1     0     0     0     1
     1     0     0     0     0
     0     1     0     0     0
```

Figure 3: Execution result

Exercise 7.11

According to the Amdahl's Lawa, $f = \sigma(n)/(\sigma(n) + \varphi(n))$, f has nothing to do with the number of processors p. However, in Gustafson-Barsis's Law, s denotes the fraction of time spent in the parallel computation performing inherently sequential operations that $s = \sigma(n)/(\sigma(n) + \varphi(n)/p)$ that means s the increase of p lead to the decrease of s. Therefore, when increasing the number of processors p the speedup predicted by Gustafson-Barsis's Law also increases with bound. Since both of the law are derived from the same general formula, the bound of the speedup is the same, say $1/f$.