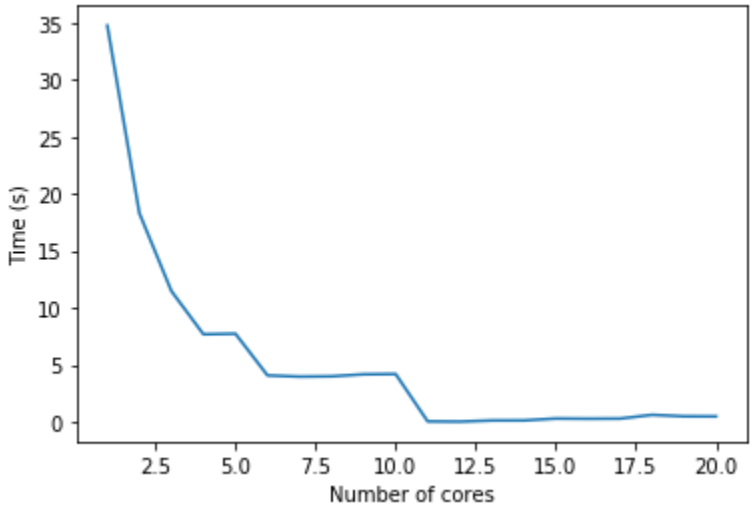
Keertana V. Chidambaram
Assignment #1
Large-Scale Computing for the Social Sciences
MACS 30123, Spring 2020

Note: all the SLURM and .py codes are given in the **Appendix** section towards the end of this document.

Question 1: Clocking CPU parallelism

Figure 1: computation time for 1,000 simulations vs. number of cores used



The speedup is non-linear because on one hand increasing the number of core helps in distributing the parallel work-load of the code to different processors. But, the serial version of the code has to be performed nevertheless and hence you cannot cut speed after a point. Also, in this particular example, there is some communication between the cores (see gather function in Appendix Q1 (Python file)), so increasing cores would mean that more computational work has to be done for communication.

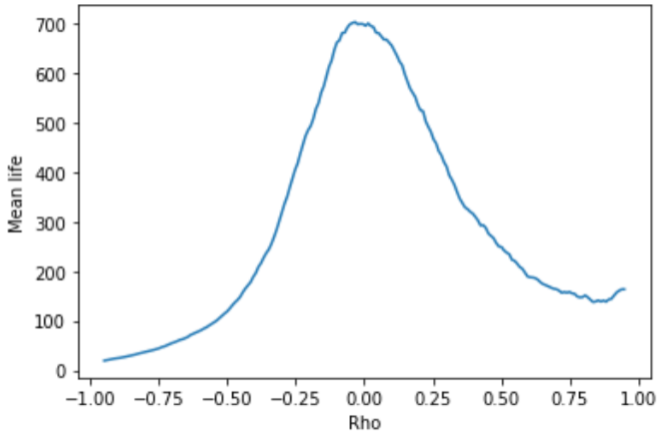
Question 2: Clocking GPU parallelism

The code took 1.80 seconds. This was one simulation of S = 1,000 and T = 4,160. While it took 25 seconds to run 100 of the same simulation in a CPU (0.25 sec per simulation). The difference in performance is because of the data communication required. In the GPU case, we need to initialize the random shock, boundaries, and result array and pass necessary information back and forth. This leads to a larger compute time for GPU as compared to CPU in this application.

Question 3: Embarrassingly parallel processing: grid search

For MPI it took 19 seconds to run the program and optimal rho value = 0.033. With OMP, it took about 45 seconds and optimal rho found was -0.033. MPI is faster, probably because how I parallelized the OMP code wass not optimal (party due to lack of flexibility for OMP and partly because of my superficial understanding of how to write OMP codes); also, there is some data communication which is why I think MPI was faster in my case.

Figure 2: Average period to first negative vs rho value (as run from my collab notebook OMP)



The optimal rho value is the closest to zero in both cases, so I believe the optimal rho will be 0, corresponding average period to negative health is approx 704 periods.

Question 4: More sophisticated parallelism: minimizer

It took 10 seconds to find optimum rho (rho = -0.051) for MPI. and it took about 15 seconds for OMP to find the optimum (rho = 0.0421). Again optimum rho is very close to zero and optimum period is about 710 periods. The computation time is definitely better for both OMP and MPI as compared to question 3. This is because the in Q3 we perform a naive exhaustive grid search whereas in Q4 when we use the optimize function, the search is done in a much more strategic manner and it doesn't need exhaustive search.

Appendix

Code: Q1 (Python file)

```
from mpi4py import MPI
import numpy as np
import scipy.stats as sts
import time

def sim_health_ind_parallel(n_runs, t0):
        # Get comm, rank, size of communicator
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        size = comm.Get_size()

        # Set parameters
        np.random.seed(25)
        S = 1000
        rho = 0.5
        mu = 3.0
        sigma = 1.0
        z_0 = mu
        T = int(4160)

        # Evenly distribute number of simulation runs across processes
        N = int(n_runs/size)

        # Simulate N simulations
        z_mat = np.zeros((N, T, S))
```

```
        for n_ind in range(N):
                z_mat[n_ind, 0, :] = z_0
                eps_mat = sts.norm.rvs(loc=0, scale=sigma, size=(T, S))
                for s_ind in range(S):
                        z_tm1 = z_0
                        for t_ind in range(T):
                                e_t = eps_mat[t_ind, s_ind]
                                z_t = rho * z_tm1 + (1 - rho) * mu + e_t
                                z_mat[n_ind, t_ind, s_ind] = z_t
                                z_tm1 = z_t

        comm.gather(z_mat, root=0)

        if rank == 0:
                # End time
                time_elapsed = time.time() - t0
                print("Number of Runs = ", n_runs, " time = ", time_elapsed, " cores = ", size)
        return

def main():
        sim_health_ind_parallel(n_runs=100, t0=time.time())

if __name__ == '__main__':
        main()
```

Code: Q1 (SLURM file)

```
#!/bin/bash

#SBATCH --job-name=health_mpi
#SBATCH --partition=broadwl
#SBATCH --constraint=fdr

module load mpi4py/3.0.1a0_py3

for i in {1..20}
do
        mpirun -n $i python ./health_sim.py > test${i}.out
done

wait
```

Code: Q2 (Python file)

```
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
import pyopencl.clrandom as clrand
import pyopencl.tools as cltools
from pyopencl.scan import GenericScanKernel
import matplotlib.pyplot as plt
import time

def sim_health_ind_parallel(t0):
        # Set up OpenCL context and command queue
        ctx = cl.create_some_context()
        queue = cl.CommandQueue(ctx)

        # Set parameters
        rho = 0.5
        mu = 3.0
        sigma = 1.0
        S = 1000
        T = int(4160)
        np.random.seed(25)

        # Generate eps matrix Normal Random Numbers on GPU of dim S*T
        rand_gen = clrand.PhiloxGenerator(ctx)
        ran = rand_gen.normal(queue, (T*S), np.float32, mu=0, sigma=sigma)
```

```
        # Establish boundaries for each simulated walk (i.e. start and end)
        # Necessary so that we perform scan only within rand walks and not between
        seg_boundaries = [1] + [0]*(T-1)
        seg_boundaries = np.array(seg_boundaries, dtype=np.uint8)
        seg_boundary_flags = np.tile(seg_boundaries, int(S))
        seg_boundary_flags = cl_array.to_device(queue, seg_boundary_flags)

        # GPU: Define Segmented Scan Kernel, scanning simulations: f(n-1) + f(n)
        prefix_sum = GenericScanKernel(ctx, np.float32,
                arguments="__global float *ran, __global char *segflags, "
                "__global float *out",
                input_expr="segflags[i] == 1 ? 3: ran[i]",
                scan_expr="across_seg_boundary ? b : 0.5*a+b", neutral="0",
                is_segment_start_expr="segflags[i]",
                output_statement="out[i] = item",
                options=[])

        # Allocate space for result of kernel on device
        dev_result = cl_array.empty_like(ran)

        # Enqueue and Run Scan Kernel
        prefix_sum(ran, seg_boundary_flags, dev_result)

        # Get results back on CPU to plot and do final calcs, just as in Lab 1
        z_mat = dev_result.get().reshape(S, T) + 3
        time_elapsed = time.time() - t0
        print("Time = ", time_elapsed)
    return

def main():
        t0 = time.time()
        sim_health_ind_parallel(t0)

if __name__ == '__main__':
        main()
```

Code: Q2 (SLURM file)

```
#!/bin/bash
#SBATCH --job-name=gpu   # job name
#SBATCH --output=gpu.out # output log file
#SBATCH --error=gpu.err  # error file
#SBATCH --time=00:05:00  # 5 minutes of wall time
#SBATCH --nodes=1        # 1 GPU node
#SBATCH --partition=gpu2 # GPU2 partition
#SBATCH --ntasks=1       # 1 CPU core to drive GPU
#SBATCH --gres=gpu:1     # Request 1 GPU

module load cuda
module load mpi4py/3.0.1a0_py3

python ./health_omp.py > ./health_omp.out
```

Code: Q3 (Python file - MPI)

```
# Question 3
from mpi4py import MPI
import numpy as np
import scipy.stats as sts
import time

# Set model parameters
mu = 3.0
sigma = 1.0
z_0 = mu
S = 1000
T = int(4160)
np.random.seed(25)
```

```
def health_sim_single(eps_mat, rho):
        first_neg_arr = []
        for s_ind in range(S):
                z_tm1 = z_0
                for t_ind in range(T):
                        e_t = eps_mat[t_ind, s_ind]
                        z_t = rho * z_tm1 + (1 - rho) * mu + e_t
                        if z_t < 0:
                                first_neg_arr.append(t_ind)
                        break
                        z_tm1 = z_t
        first_neg_avg = np.mean(first_neg_arr)
        return first_neg_avg

def findMinParam(param1, param2, datatype):
        if param1['max'] > param2['max']:
                return param1
        else:
                return param2
def health_sim_optimizer(eps_mat, rho_mat, t0):
        # Get comm, rank, size of communicator
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        size = comm.Get_size()
        start_ind = int(rank * len(eps_mat)/size)
        end_ind = int(start_ind + len(eps_mat)/size)
        rho_new = rho_mat[start_ind:end_ind]
        opt_param = {'max': 0, 'rho':None}
        for rho in rho_new:
                first_neg = health_sim_single(eps_mat, rho)
                if first_neg > opt_param['max']:
                        opt_param['max'] = first_neg
                        opt_param['rho'] = rho
        counterSumOp = MPI.Op.Create(findMinParam, commute=True)
        if rank == 0:
                max_param = comm.allreduce(opt_param, op=counterSumOp)
                print('max param = ', max_param, 'size = ', size)
                print('time = ', time.time()-t0)
        return
def main():
        t0 = time.time()
        eps_mat = sts.norm.rvs(loc=0, scale=sigma, size=(T, S))
        rho_mat = np.linspace(-0.95, 0.95, 200)
        health_sim_optimizer(eps_mat, rho_mat, t0)
if __name__ == '__main__':      main()
```

Code: Q3 (Python file - OMP)

```
# Question 3
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array
import pyopencl.clrandom as clrand
import pyopencl.tools as cltools from pyopencl.scan
import GenericScanKernel
import matplotlib.pyplot as plt
import time from pyopencl.reduction
import ReductionKernel from pyopencl.elementwise
import ElementwiseKernel

def sim_health_ind_parallel(t0):
        # Set up OpenCL context and command queue
        ctx = cl.create_some_context()
        queue = cl.CommandQueue(ctx)

        # Set parameters
        rho_mat = np.linspace(-0.95, 0.95, 200)
        mu = 3.0
        sigma = 1.0
```

```
        S = 1000
        T = int(4160)
        np.random.seed(25)

        # Generate eps matrix Normal Random Numbers on GPU of dim S*T
        rand_gen = clrand.PhiloxGenerator(ctx)
        ran = rand_gen.normal(queue, (T*S), np.float32, mu=0, sigma=sigma)
        opt_mean = 0
        opt_rho = None

        for rho in rho_mat:
                seg_boundaries = [1] + [0]*(T-1)
                seg_boundaries = np.array(seg_boundaries, dtype=np.uint8)
                seg_boundary_flags = np.tile(seg_boundaries, int(S))
                seg_boundary_flags = cl_array.to_device(queue, seg_boundary_flags)
                prefix_sum = GenericScanKernel(ctx, np.float32,
                arguments="__global float *ran, __global char *segflags, "
                        "__global float *out, __global float r",
                        input_expr="segflags[i] == 1 ? 0: ran[i]",
                        scan_expr="across_seg_boundary ? b : r*a+b", neutral="0",
                        is_segment_start_expr="segflags[i]",
                        output_statement="out[i] = item+3",
                        options=[])
                dev_result = cl_array.empty_like(ran)
                prefix_sum(ran, seg_boundary_flags, dev_result, rho)
                z_mat = dev_result.get().reshape((S, T))
                periods = np.argmax(z_mat<0, axis=1)
                mean = np.mean(periods)
                if mean > opt_mean:
                        opt_mean = mean
                        opt_rho = rho
        time_elapsed = time.time() - t0
        print("Time = ", time_elapsed)
        print("Rho =", opt_rho )
        print("Mean=", opt_mean)
        return

def main():
        t0 = time.time()
        sim_health_ind_parallel(t0)

if __name__ == '__main__':
        main()
```

Code: Q3 (SLURM file)

```
#!/bin/bash

#SBATCH --job-name=grids_mpi
#SBATCH --partition=broadwl
#SBATCH --constraint=fdr

module load mpi4py/3.0.1a0_py3

mpirun -n 100 python ./grids_mpi.py > grids_mpi.out



#!/bin/bash
#SBATCH --job-name=gpu   # job name
#SBATCH --output=gpu.out # output log file
#SBATCH --error=gpu.err  # error file
#SBATCH --time=00:05:00  # 5 minutes of wall time
#SBATCH --nodes=1        # 1 GPU node
#SBATCH --partition=gpu2 # GPU2 partition
#SBATCH --ntasks=1       # 1 CPU core to drive GPU
#SBATCH --gres=gpu:1     # Request 1 GPU

module load cuda
module load mpi4py/3.0.1a0_py3
```

```
python ./grids_omp.py > ./grids_omp.out
```

Code: Q4 (Python file - MPI)

```
# Question 4
from mpi4py import MPI
import numpy as np
import scipy.stats as sts
import time from scipy.optimize
import minimize_scalar

# Set model parameters
mu = 3.0
sigma = 1.0
z_0 = mu
S = 1000
T = int(4160)
np.random.seed(25)

def fun(rho):
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        size = comm.Get_size()
        first_neg = []
        S_new = int(S/size)
        for s_ind in range(S_new):
                eps_mat = sts.norm.rvs(loc=0, scale=sigma , size=(T))
                z_tm1 = z_0
                for t_ind in range (T):
                        e_t = eps_mat[t_ind]
                        z_t = rho * z_tm1 + (1 - rho) * mu + e_t
                        if z_t < 0:
                                first_neg.append(z_t)
                                break
                        z_tm1 = z_t
        if rank == 0:
                comm. gather(first_neg , root=0)
        return -np.mean(first_neg)

def main():
        t0 = time.time()
        res = minimize_scalar(fun, bounds=(-0.95, 0.95), method='bounded')
        print(res.x)
        print(res.fun)
        time_elapsed = time.time() - t0
        print(" time = ", time_elapsed)

if __name__ == '__main__':
        main()
```

Code: Q4 (Python file - OMP)

```
# Question 4
import numpy as np
import scipy.stats as sts
import time from scipy.optimize
import minimize_scalar

# Set model parameters
mu = 3.0
sigma = 1.0
z_0 = mu
S = 1000
T = int(4160)
np.random.seed(25)

def fun(rho):
        ctx = cl.create_some_context()
```

```
        queue = cl.CommandQueue(ctx)
        rand_gen = clrand.PhiloxGenerator(ctx)
        ran = rand_gen.normal(queue, (T*S), np.float32, mu=0, sigma=sigma)

        seg_boundaries = [1] + [0]*(T-1)
        seg_boundaries = np.array(seg_boundaries, dtype=np.uint8)
        seg_boundary_flags = np.tile(seg_boundaries, int(S))
        seg_boundary_flags = cl_array.to_device(queue, seg_boundary_flags)

        prefix_sum = GenericScanKernel(ctx, np.float32,
        arguments="__global float *ran, __global char *segflags, "
                "__global float *out, __global float r",
                input_expr="segflags[i] == 1 ? 0: ran[i]",
                scan_expr="across_seg_boundary ? b : r*a+b", neutral="0",
                is_segment_start_expr="segflags[i]",
                output_statement="out[i] = item+3",
                options=[])
        dev_result = cl_array.empty_like(ran)
        prefix_sum(ran, seg_boundary_flags, dev_result, rho)
        z_mat = dev_result.get().reshape((S, T))
        periods = np.argmax(z_mat<0, axis=1)
        mean = np.mean(periods)
        return -mean

def main():
        t0 = time.time()
        res = minimize_scalar(fun, bounds=(-0.95, 0.95), method='bounded')
        print(res.x)
        time_elapsed = time.time() - t0
        print(" time = ", time_elapsed)

if __name__ == '__main__':
        main()
```