INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE AT HARVARD UNIVERSITY

HARVARD School of Engineering and Applied Sciences

Solutions HW: B. Parallel Computing

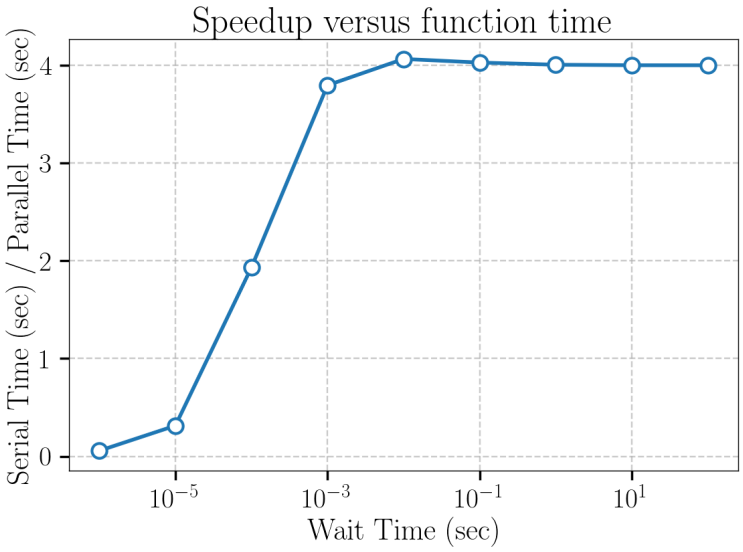
1. Python Multiprocessing Module (20%)

1.1. Count to Ten (10%)

1. There doesn't appear to be any order to when jobs finish. It looks like some processors finish before other processors.
2. We should do our best to balance the computations to try to minimize downtime.
3. A scenario where this would be important is some computation that is sensitive to idle processes.

1.2. How Much Faster? (10%)

1. See accompanying solution file (P12.py).



2.
3. As the computing time increases, the ratio asymptotes to 4. More time spent computing (and less communicating) will result in a 4x speed-up.

4. Yes. Depending on the granularity of the tasks. If a parallel program is spending all it's time communicating, then a serial program that is performing computations could feasibly finish before the parallel program.

2. Performance Optimization (20%)

COMPUTING EXPERIMENT ENVIRONMENT

System: t2.2xlarge at AWS

CPU Specs

Model: Intel(R) Xeon(R) CPU E5-2686 v
Clock: 2.30 GHz
Number of vCPUs: 8
Main Memory: 32 GiB

Operating System

Distro: Ubuntu 16.04 LTS

Compiler

Name: gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9)
Version: 5.4.0 20160609

2.1. Optimization of Basic Codes (10%)

	O0	O1	O2	O3	Ofast	Os
`Dotprod_serial.c` `VECLEN = 100`	1e-06	2e-06	1e-06	1e-06	1e-06	1e-06
`Dotprod_serial.c` `VECLEN = 10000`	3.5e-05	3.5e-05	1.2e-05	1.2e-05	7e-06	1.3e-05
`Dotprod_serial.c` `VECLEN = 1e8`	3.40329	3.40663	1.4515	1.46696	1.37251	1.46542
`jacobi1d.c`	44.9531	22.4818	23.013	23.471	23.0274	22.9424

	2x unrolling	4x unrolling
`Dotprod_serial.c` `VECLEN = 1e8`	3.40308	3.38768
`jacobi1d.c`	26.939	25.3912

2.2. Optimization of Matrix Multiplication (5%)

See an example here

https://github.com/EvanPurkhiser/CS-Matrix-Multiplication/blob/master/report.md

	O0	O3	Loop Unrolling	Blocking	Loop Unrolling + Blocking
Notes			Instructions = 4	Blocksize = 25	Blocksize = 150 Instructions = 2
`seq_mm.c`	17.1956	2.50667	16.3225	11.781	7.51579

Note: The students should provide source code for the combined version of the code with loop unrolling and blocking.

To estimate the optimal size of block remember to use sqrt(M/3), where M is the size of the cache (usually L3 size).

3. Accelerated Computing (20%)

COMPUTING EXPERIMENT ENVIRONMENT

System: g3.4xlarge at AWS

CPU Specs (*not relevant for this exercise because it only uses the GPU part*)

Model: Intel Xeon E5-2686 v4 (Broadwell)
Clock: 2.7 GHz
Number of vCPUs: 16
Main Memory: 122 GiB
EBS Bandwidth: 3.5 Gbps
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 46080K

GPU Specs

Model: NVIDIA Tesla M60 GPU
Number of GPUs: 1
Number of cores per GPU: 2,048
Memory per GPU: 8 GiB
Driver: NVIDIA-SMI 390.12

Operating System

Distro: Ubuntu 16.04 LTS
Kernel: 4.4.0-1047-aws

Compiler

Name: PGI Community Edition
Version: 17.10

CUDA Driver Version: 9010

3.1. Acceleration of Basic Code (15%)

A. SEQUENTIAL EXECUTION

Compilation

```
gcc -DUSE_CLOCK jacobi1d.c timing.c -o jacobi1d
```

Execution

```
./jacobi1d 100000000 100
n: 100000000
nsteps: 100
Elapsed time: 44.3638 s
```

Compilation with -O3

```
gcc -DUSE_CLOCK -O3 jacobi1d.c timing.c -o jacobi1d
```

Execution

```
./jacobi1d 100000000 100
n: 100000000
nsteps: 100
Elapsed time: 23.1418 s
```

B. ACCELERATED (FIRST VERSION)

Code

```
for (sweep = 0; sweep < nsweeps; sweep += 2) {

        /* Old data in u; new data in utmp */
#pragma acc kernels loop independent
        for (i = 1; i < n; ++i)
            utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;

        /* Old data in utmp; new data in u */
#pragma acc kernels loop independent
        for (i = 1; i < n; ++i)
            u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
```

Compile

```
pgcc -DUSE_CLOCK -acc -Minfo jacobi1d_oacc_a.c timing.c -o
jacobi1d_oacc_a

 30, Generating implicit copyin(u[:n+1])
         Generating implicit copyout(utmp[1:n-1])
         Generating implicit copyin(f[1:n-1])
     31, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         31, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     31, FMA (fused multiply-add) instruction(s) generated
     35, Generating implicit copyin(f[1:n-1],utmp[:n+1])
         Generating implicit copyout(u[1:n-1])
     36, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         36, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     36, FMA (fused multiply-add) instruction(s) generated
```

Indicates 128 threads per block

Execute

```
./jacobi1d_oacc_a 100000000 100

n: 100000000
nsteps: 100
Elapsed time: 50.371 s

Accelerator Kernel Timing data
/home/ubuntu/jacobi1d_oacc_a.c
  jacobi  NVIDIA  devicenum=0
    time(us): 23,640,216
    30: compute region reached 50 times
        31: kernel launched 50 times
            grid: [65535]  block: [128]
             device time(us): total=1,028,721 max=20,598 min=20,548
avg=20,574
```

```
            elapsed time(us): total=1,030,884 max=20,639 min=20,588
avg=20,617
    30: data region reached 100 times
        30: data copyin transfers: 4800
             device time(us): total=7,204,952 max=1,566 min=1,027
avg=1,501
        35: data copyout transfers: 2400
             device time(us): total=3,586,598 max=1,520 min=1,032
avg=1,494
    35: compute region reached 50 times
        36: kernel launched 50 times
            grid: [65535]  block: [128]
             device time(us): total=1,028,768 max=20,599 min=20,557
avg=20,575
            elapsed time(us): total=1,030,894 max=20,643 min=20,599
avg=20,617
    35: data region reached 100 times
        35: data copyin transfers: 4800
             device time(us): total=7,204,671 max=1,665 min=1,026
avg=1,500
        38: data copyout transfers: 2400
             device time(us): total=3,586,506 max=1,520 min=1,032
avg=1,494
```

2.2 seconds in processing and 22 seconds in data transfer

C. ACCELERATED (SECOND ENHANCED VERSION)

Code

```
#pragma acc data copy(u[0:n]) create(utmp[0:n]) copyin(f[0:n])

    for (sweep = 0; sweep < nsweeps; sweep += 2) {

        /* Old data in u; new data in utmp */
#pragma acc kernels loop independent
        for (i = 1; i < n; ++i)
            utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;

        /* Old data in utmp; new data in u */
#pragma acc kernels loop independent
        for (i = 1; i < n; ++i)
            u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
```

```
    }
```

Compile

```
pgcc -DUSE_CLOCK -acc -Minfo jacobi1d_oacc_b.c timing.c -o
jacobi1d_oacc_b

     27, Generating copyin(f[:n])
         Generating create(utmp[:n])
         Generating copy(u[:n])
     33, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         33, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     33, FMA (fused multiply-add) instruction(s) generated
     38, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         38, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     38, FMA (fused multiply-add) instruction(s) generated
```

Indicates 128 threads per block

Execute

```
./jacobi1d_oacc_b 100000000 100

n: 100000000
nsteps: 100
Elapsed time: 2.77933 s

Accelerator Kernel Timing data
/home/ubuntu/jacobi1d_oacc_b.c
  jacobi  NVIDIA  devicenum=0
    time(us): 2,367,934
    27: data region reached 2 times
        27: data copyin transfers: 96
             device time(us): total=143,565 max=2,221 min=1,031
avg=1,495
        42: data copyout transfers: 48
```

```
              device time(us): total=71,617 max=1,507 min=1,038
avg=1,492
    32: compute region reached 50 times
        33: kernel launched 50 times
            grid: [65535]  block: [128]
             device time(us): total=1,076,496 max=21,566 min=21,478
avg=21,529
            elapsed time(us): total=1,077,723 max=21,596 min=21,502
avg=21,554
    37: compute region reached 50 times
        38: kernel launched 50 times
            grid: [65535]  block: [128]
             device time(us): total=1,076,256 max=21,555 min=21,488
avg=21,525
            elapsed time(us): total=1,077,523 max=21,600 min=21,525
avg=21,550

num_gangs=65535 num_workers=1 vector_length=128 grid=65535 block=128
```

2.2 seconds in processing and 1.3 in data transfer

C. COMPARE EXECUTION TIME OF BOTH CODES

TABLE

k

	A	B
6	0.66	0.16
7	5.3	0.4
8	50	2.7

D. TUNING CHANGING NUMBER OF THREADS PER BLOCK

About OpenACC tuning

http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf

Code

```
#pragma acc kernels loop independent vector(1024)
```

OPTIMAL FOR 1024 (maximum allowed by PGI compilers)

```
n: 100000000
nsteps: 100
Elapsed time: 2.61042 s

Accelerator Kernel Timing data
/home/ubuntu/jacobi1d_oacc_c.c
  jacobi  NVIDIA  devicenum=0
    time(us): 2,195,183
    27: data region reached 2 times
        27: data copyin transfers: 96
             device time(us): total=145,028 max=1,568 min=1,036
avg=1,510
        42: data copyout transfers: 48
             device time(us): total=71,878 max=1,510 min=1,040
avg=1,497
    32: compute region reached 50 times
        33: kernel launched 50 times
            grid: [65535]  block: [1024]
             device time(us): total=989,300 max=19,800 min=19,768
avg=19,786
            elapsed time(us): total=990,549 max=19,859 min=19,792
avg=19,810
    37: compute region reached 50 times
        38: kernel launched 50 times
            grid: [65535]  block: [1024]
             device time(us): total=988,977 max=19,795 min=19,765
avg=19,779
            elapsed time(us): total=990,182 max=19,854 min=19,787
avg=19,803
```

WHY?

The programmer divides work into threads, threads into thread blocks, and thread blocks into grids. The compute work distributor allocates thread blocks to Streaming Multiprocessors (SMs). Once a thread block is distributed to a SM the resources for the thread block are allocated (warps and shared memory) and threads are divided into groups of 32 threads called warps.

In this particular case, the larger the vector length the lower the number of blocks, and given that each grid has 65,535 blocks, this reduces the management overhead.

100.000.000 / 128 => 781.250 blocks
100.000.000 / 1024 => 97.565 blocks

3.2. Acceleration of Matrix Multiplication (5%)

A. SEQUENTIAL CODE

Sequential optimized code

```
gcc -DUSE_CLOCK -O3 seq_mm.c timing.c  -o seq_mm
```

Execution

```
Elapsed time: 2.4834 s
```

B. ACCELERATED VERSION

Accelerated code

```
#pragma acc kernels loop independent copyout(c) copyin(a,b)
  for (i=0; i<N; i++)
    {
    for(j=0; j<N; j++)
      for (k=0; k<N; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
```

Compile

```
pgcc -DUSE_CLOCK -acc -Minfo seq_mm_oacc.c timing.c -o seq_mm_oacc
```

Execute

```
Elapsed time: 0.913734 s

Accelerator Kernel Timing data
/home/ubuntu/seq_mm_oacc.c
  main  NVIDIA  devicenum=0
    time(us): 216,659
    26: compute region reached 1 time
```

```
        30: kernel launched 1 time
            grid: [12x1500]  block: [128]
             device time(us): total=211,947 max=211,947 min=211,947
avg=211,947
            elapsed time(us): total=211,990 max=211,990 min=211,990
avg=211,990
    26: data region reached 2 times
        26: data copyin transfers: 4
             device time(us): total=3,257 max=1,503 min=122 avg=814
        34: data copyout transfers: 2
             device time(us): total=1,455 max=1,339 min=116 avg=727
```

C. DISCUSSION

See the GPU part is only 0.2, (compared with 2.4 from the optimized sequential execution). However there is an overhead in set-up and overall time is 0.9. We can reduce transfer time by creating the a and b arrays in the GPUs

The system by default assigns 128 threads per block and a 12x1500 block grid that work columnwise with the matrix.

4. Shared-Memory Parallel Processing (20%)

COMPUTING EXPERIMENT ENVIRONMENT

System: t2.2xlarge at AWS

CPU Specs

Model: Intel(R) Xeon(R) CPU E5-2686 v4
Clock: 2.3 GHz
Number of vCPUs: 8
Main Memory: 16 GiB
EBS Bandwidth: 3.5 Gbps
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 46080K

Operating System

Distro: Ubuntu 16.04 LTS
Kernel: 4.4.0-1047-aws

Compiler

Name: gcc
Version: 5.4.1 20160904 (Ubuntu 5.4.1-2ubuntu1~16.04)

4.1. Parallelization of Basic Code (10%)

A. SEQUENTIAL EXECUTION

Compilation with -O3

```
gcc -DUSE_CLOCK -O3 jacobi1d.c timing.c -o jacobi1d
```

Execution

```
./jacobi1d 100000000 100

n: 100000000
nsteps: 100
Elapsed time: 23.2348 s
```

B. PARALLELIZED

Code

```
for (sweep = 0; sweep < nsweeps; sweep += 2) {

#pragma omp parallel shared(u, utmp, f, h2,n) private(i)
{

        /* Old data in u; new data in utmp */
#pragma omp for
        for (i = 1; i < n; ++i)
            utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;

        /* Old data in utmp; new data in u */
#pragma omp for
        for (i = 1; i < n; ++i)
            u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
}
}
```

Compile

```
gcc -DUSE_CLOCK -fopenmp -O3 jacobi1d_omp.c timing.c -o jacobi1d_omp
```

Execute with 4 cores

```
export OMP_NUM_THREADS=4

./jacobi1d_omp 100000000 100

Threads: 4
n: 100000000
nsteps: 100
Elapsed time: 7.05428 s
```

C. PERFORMANCE TABLES

1. Execution times for different problem sizes 10^k, where k=5 to 8

```
Time Parallel (4 cores)

5 0.004
6 0.03
7 0.71
```

```
8 7.05

Time Sequential

5 0.009
6 0.094
7 2.3
8 24.8

Speedup (4 cores)

5 2.25
6 3.13
7 3.24
8 3.52
```

The speedup grows with the size of the problem because the granularity also grows. As problem size increases, the synchronization overhead due to creation/control of threads becomes lower compared with the computing in each time step.

2. Run your codes with ncells of 100,000 on 2-8 cores and produce a speedup plot.

```
Time Parallel (10^8 and growing number of cores)

Cores      Time       Speedup
 1          24          1.00
 2          12.3        1.95
 3          8.7         2.76
 4          7.1         3.38
 5          6.3         3.81
 6          6           4.00
 7          6.02        3.99
 8          7           3.43
```

As problem size per core decreases, the synchronization overhead due to creation/control of threads becomes dominant. **I think this behaviour is because these instances only allow 4**

simultaneous threads. We will see that with MPI on multiple nodes we can continue reducing the execution time.

4.2. Scheduling Policies (10%)

I took the code from

https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_openmp/mandelbrot_openmp.html

Compile

```
gcc -O3 -fopenmp mandelbrot_omp.c -o mandelbrot_omp -lm
```

Execution time

1 0.30
2 0.15
4 0.15
8 0.12

This is because by default schedule is static and assign a chunk of 500/4 contiguous iterations to each thread. Because computing is concentrated in some of the pixels, the distribution of work is unbalanced.

Now if you specify schedule(static,1), chunk is 1, which is the number of contiguous iterations assigned to each thread, and the work gets balanced.

1 0.30
2 0.15
4 0.08
8 0.04

In the first case, the first thread executes iterations 1 to 125, next 126 to 500…. And with chunk equal to 1, the first executes iterations, 1,5,9… the second 2,6,10…

And with dynamic and default chunk (which is one)

1 0.30
2 0.15
4 0.08
8 0.04

5. Distributed-Memory Parallel Processing (20%)

COMPUTING EXPERIMENT ENVIRONMENT

Distributed Memory

Nodes: 2
Type: t2.2xlarge at AWS
Network: Shared 1 Gbit/sec

CPU Specs

Model: Intel(R) Xeon(R) CPU E5-2686 v4
Clock: 2.3 GHz
Number of vCPUs: 8
Main Memory: 16 GiB
EBS Bandwidth: 3.5 Gbps
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 46080K

Operating System

Distro: Ubuntu 16.04 LTS
Kernel: 4.4.0-1047-aws

Compiler

mpicc MPICH version 3.2
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.6)

5.1. Parallelization of Basic Code (10%)

A. SEQUENTIAL EXECUTION

Compilation with -O3

```
gcc -DUSE_CLOCK -O3 jacobi1d.c timing.c -o jacobi1d
```

Execution

```
./jacobi1d 100000000 100

n: 100000000
nsteps: 100
Elapsed time: 19.5775 s
```

B. PARALLELIZED

Code

```
 /* YOUR SOLUTION HERE */

    /* SEND RIGHT */

    if (rank != size-1)

        MPI_Send(&u[n-1], 1, MPI_DOUBLE, rank+1,
                 1, MPI_COMM_WORLD);

    /* RECEIVE FROM RIGHT */

    if (rank != 0)

        MPI_Recv(&u[0], 1, MPI_DOUBLE, rank-1,
                     1, MPI_COMM_WORLD, &status);

    /* SEND LEFT */

    if (rank != 0)

        MPI_Send(&u[1], 1, MPI_DOUBLE, rank-1,
                 1, MPI_COMM_WORLD);

    /* RECEIVE FROM RIGHT */

    if (rank != size-1)

        MPI_Recv(&u[n], 1, MPI_DOUBLE, rank+1,
                     1, MPI_COMM_WORLD, &status);
```

Compile

```
mpicc -O3 jacobi1d_mpi.c -o jacobi1d_mpi
```

Try in one node with 4 processes

```
mpirun -np 4 ./jacobi1d_mpi 100000000 100
Processes: 4
n: 100000000
```

```
nsteps: 100
Elapsed time: 7.81268 s
```

Try in one node with 8 processes

```
mpirun -np 8 ./jacobi1d_mpi 100000000 100
Processes: 8
n: 100000000
nsteps: 100
Elapsed time: 7.28838 s
```

Same behaviour with OpenMP, no improvement from 4 tasks.

Now 8 on two nodes

```
mpirun -np 8 -hosts master,node1 ./jacobi1d_mpi 100000000 100
Processes: 8
n: 100000000
nsteps: 100
Elapsed time: 3.87772 s
```

C. PERFORMANCE TABLES

Should be very similar to those obtained with OpenMP. But in this case because we have two nodes we can achieve 8-grade parallelism.

5.2. Hybrid Parallel Processing (10%)

New Code

```
  for (sweep = 0; sweep < nsweeps; sweep += 2) {

        /* Exchange ghost cells */
        ghost_exchange(u, n, rank, size);
        utmp[0] = u[0];
        utmp[n] = u[n];


        /* Sweep */
#pragma omp parallel for shared(u, utmp, f, h2,n) private(i)
        for (i = 1; i < n; ++i)
            utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;
```

```
        /* Exchange ghost cells */
        ghost_exchange(utmp, n, rank, size);
        u[0] = utmp[0];
        u[n] = utmp[n];

        /* Old data in utmp; new data in u */
#pragma omp parallel for shared(u, utmp, f, h2,n) private(i)
        for (i = 1; i < n; ++i)
            u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
    }
```

Compile

```
mpicc -O3 -fopenmp jacobi1d_mpi_omp.c -o jacobi1d_mpi_omp
```

Evaluate

2 tasks with 4 threads each

```
mpirun -np 2 -hosts master,node1 -genv OMP_NUM_THREADS 4
./jacobi1d_mpi_omp 100000000 100

Processes: 2
n: 100000000
nsteps: 100
Elapsed time: 4.00473 s
```

4 tasks with 2 threads each

```
mpirun -np 4 -hosts master,node1 -genv OMP_NUM_THREADS 2
./jacobi1d_mpi_omp 100000000 100

Processes: 4
n: 100000000
nsteps: 100
Elapsed time: 3.96339 s
```

8 tasks with 1 thread each

```
mpirun -np 8 -hosts master,node1 -genv OMP_NUM_THREADS 1
./jacobi1d_mpi_omp 100000000 100
```

```
Processes: 8
n: 100000000
nsteps: 100
Elapsed time: 3.88561 s
```

Discuss

No difference in performance in this case.