David Tan dwt160030

Program 3 Report in CS6376 Parallel Processing Course

1. Problem Statement:

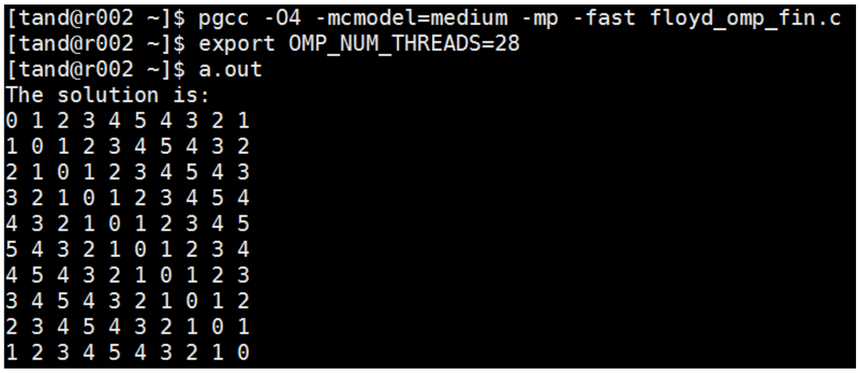
   This program mainly focuses on optimizing the Floyd's Algorithm serial code by using combination of MPI, openMP and openACC.

2. Approach to Solution:

   We need to build five testing cases in this program: openMP, openACC, MPI, openMP + MPI, and openACC + MPI.

   First, I built a serial version of the Floyd algorithm and test it to check if it gives me the correct result. Next, I started to implement the openMP parallel pragma for the first test case; implement the openACC parallel pragma for the second test case, and so on.

   The following image is the test to see if the algorithm run correctly.

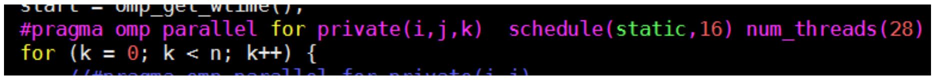


```
[tand@r002 ~]$ pgcc -O4 -mcmodel=medium -mp -fast floyd_omp_fin.c
[tand@r002 ~]$ export OMP_NUM_THREADS=28
[tand@r002 ~]$ a.out
The solution is:
0 1 2 3 4 5 4 3 2 1
1 0 1 2 3 4 5 4 3 2
2 1 0 1 2 3 4 5 4 3
3 2 1 0 1 2 3 4 5 4
4 3 2 1 0 1 2 3 4 5
5 4 3 2 1 0 1 2 3 4
4 5 4 3 2 1 0 1 2 3
3 4 5 4 3 2 1 0 1 2
2 3 4 5 4 3 2 1 0 1
1 2 3 4 5 4 3 2 1 0
```

3. Solution Description:

   The openMP solution simply add a line of pragma before the Floyd algorithm nested loop.

   The performance had the maximum Gflops when N, the input size, is around at 3000.



```
#pragma omp parallel for private(i,j,k)  schedule(static,16) num_threads(28)
for (k = 0; k < n; k++) {
```

   Command line: pgcc -O4 -mcmodel=medium -mp -fast floyd_omp_fin.c

   export OMP_NUM_THREADS=28

	openMP (n = 28)
N=800	0.052919 sec 19.350360Gflops
N=2000	0.539536 sec 29.655111Gflops
N=3000	1.558863 sec 34.640634Gflops
N=4000	3.811736 sec 33.580499Gflops
N=5000	7.984081 sec 31.312307Gflops
N=8000	31.201778 sec 32.818643Gflops

Gustafson-Barsis's Law: $\psi \leq p + (1 - p)s$

The original serial code running time: 2.3 sec (n= 1000)

The final running time result I had after I modified the code: 1.5 sec

Speedup = 2.3/1.5 = 1.53

$1.53 \leq 28 + (-27)*s$

$26.47 \geq 27*s$

$s \leq 0.9453$

Based on the formula, the serial fraction is 94%. Because the Gustafson-Barsis's Law ignores the parallel overhead term, this speedup result might be overestimated.

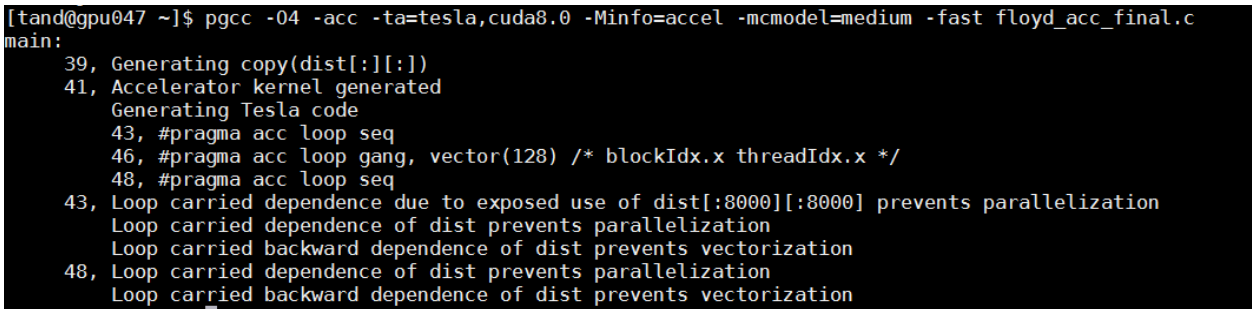
Karp-Flatt Metric: $e = \frac{1/\psi - 1/p}{1 - 1/p}$

e = (1/1.53 – 1/28) / (1 -1/28) = 0.64076

Though Floyd's Algorithm is formed by three nested loops and seems can be paralleled perfectly, there might be data dependence in it, causing it still has a lot portion of serial execution in it.

For the openACC portion. At first I had problem avoiding the dependence problem. But after I searched online, some webpage stated that there is data-dependent and iterative nature of the Floyd's algorithm. I think this may be the reason the Minfo showed there is data dependence to parallel the nested iterations for the Floyd's algorithm, causing the performance is not very good. I did try to use two arrays like the Laplace program, but as it turned out, it run even slower because it computed more nested loops then before.

Command line: pgcc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast floyd_acc_final.c



```
[tand@gpu047 ~]$ pgcc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast floyd_acc_final.c
main:
     39, Generating copy(dist[:][:])
     41, Accelerator kernel generated
         Generating Tesla code
         43, #pragma acc loop seq
         46, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         48, #pragma acc loop seq
     43, Loop carried dependence due to exposed use of dist[:8000][:8000] prevents parallelization
         Loop carried dependence of dist prevents parallelization
         Loop carried backward dependence of dist prevents vectorization
     48, Loop carried dependence of dist prevents parallelization
         Loop carried backward dependence of dist prevents vectorization
```

	openACC
N=800	0.151945sec 6.739281Gflops
N=2000	1.987087 sec 8.051988Gflops
N=3000	4.134310 sec 13.061430Gflops
N=4000	8.553777sec 14.964150Gflops
N=5000	13.490275 sec 18.531868Gflops
N=8000	75.382257sec 13.584098Gflops

For the MPI testing, we used MPI library and its function to help us speedup the program. I used row block distribution for my code. I initialized an array of size N*N, and scatter it to P array of size (N/P)*N (P is the number of nodes I requested.) One thing should be kept in mind is that N must be dividable by P.

Command line: module load mpi/pgi_openmpi

mpicc -O4 -fast floyd_mpi.c

mpirun -np 28 a.out

	MPI(n 28)
N=840	0.042841 sec 27.669942 Gflops
N=2800	0.775554 sec 56.609856 Gflops
N=3080	1.028041 sec 56.842309 Gflops
N=3360	1.611700 sec 47.072105 Gflops
N=3640	1.706929 sec 56.509139 Gflops
N=3920	2.219825 sec 54.271204 Gflops
N=8000	14.314797 sec 24.536289 Gflops

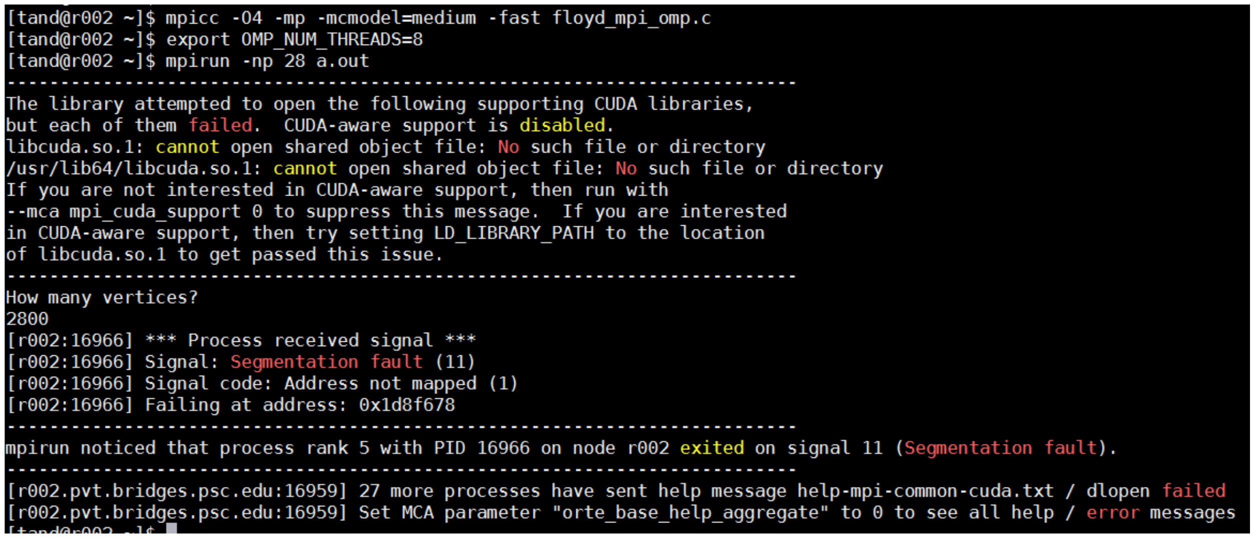
For the MPI openMP hybrid coding, if I export the number of threads greater than 1, the program will show this Segmentation fault. I searched online and still cannot find any solution for it. This is the reason why I did not have the expected increasing performance on MPI + openMP code compared to MPI only code.

Command line: mpicc -O4 -mp -mcmodel=medium -fast floyd_mpi_omp.c

mpirun -np 28 a.out

	MPI + openMP (n 28) floyd_mpi_omp.c
N=840	0.055767 sec 21.256442 Gflops
N=2800	0.760352 sec 57.741678 Gflops
N=3080	1.053150 sec 55.487085 Gflops
N=3360	1.301310 sec 58.299799 Gflops
N=3640	1.688504 sec 57.125768 Gflops
N=3920	2.240346 sec 53.774094 Gflops
N=8000	13.991080 sec 25.103995 Gflops

The following image is the problem I encountered when I tried more than one threads to run the program.



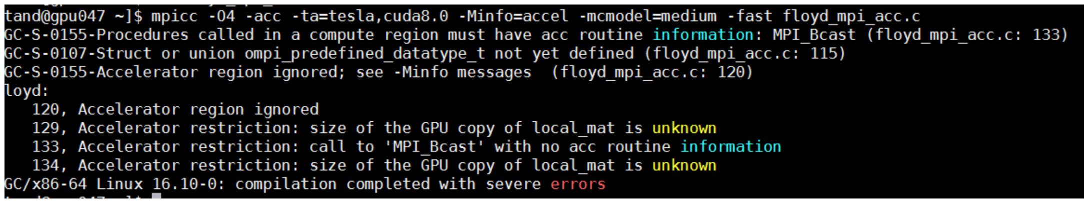
```
[tand@r002 ~]$ mpicc -O4 -mp -mcmodel=medium -fast floyd_mpi_omp.c
[tand@r002 ~]$ export OMP_NUM_THREADS=8
[tand@r002 ~]$ mpirun -np 28 a.out
--------------------------------------------------------------------------
The library attempted to open the following supporting CUDA libraries,
but each of them failed.  CUDA-aware support is disabled.
libcuda.so.1: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directory
If you are not interested in CUDA-aware support, then run with
--mca mpi_cuda_support 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
--------------------------------------------------------------------------
How many vertices?
2800
[r002:16966] *** Process received signal ***
[r002:16966] Signal: Segmentation fault (11)
[r002:16966] Signal code: Address not mapped (1)
[r002:16966] Failing at address: 0x1d8f678
--------------------------------------------------------------------------
mpirun noticed that process rank 5 with PID 16966 on node r002 exited on signal 11 (Segmentation fault).
--------------------------------------------------------------------------
[r002.pvt.bridges.psc.edu:16959] 27 more processes have sent help message help-mpi-common-cuda.txt / dlopen failed
[r002.pvt.bridges.psc.edu:16959] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

For the MPI openACC hybrid coding, I had a hard time combining these two tools together without showing any severe error. After I finally passed the compilation, the Gflops is so small that I cannot run the program with large input size.

Command line:

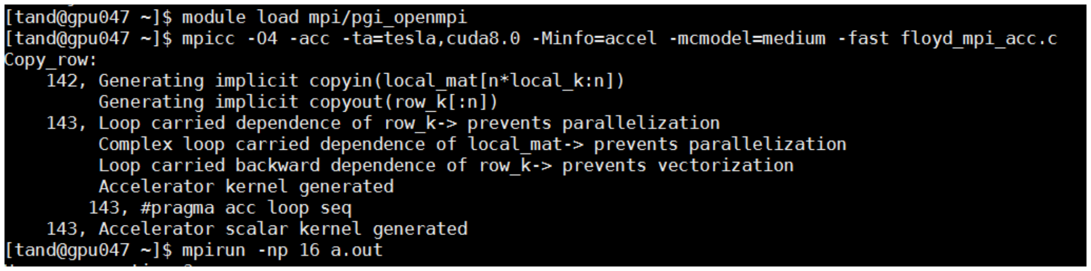
mpicc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast floyd_mpiacc.c

	MPI + openACC (n 16) floyd_mpi_acc.c
N=1600	6.968238 sec 1.175620Gflops
N=3200	9.586810sec 6.836059Gflops
N=4800	25.310623sec 8.738781Gflops
N=6400	49.575703 secsec 10.575503Gflops
N=8000	88.064074sec 11.627897Gflops



```
tand@gpu047 ~]$ mpicc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast floyd_mpi_acc.c
GC-S-0155-Procedures called in a compute region must have acc routine information: MPI_Bcast (floyd_mpi_acc.c: 133)
GC-S-0107-Struct or union ompi_predefined_datatype_t not yet defined (floyd_mpi_acc.c: 115)
GC-S-0155-Accelerator region ignored; see -Minfo messages  (floyd_mpi_acc.c: 120)
loyd:
   120, Accelerator region ignored
   129, Accelerator restriction: size of the GPU copy of local_mat is unknown
   133, Accelerator restriction: call to 'MPI_Bcast' with no acc routine information
   134, Accelerator restriction: size of the GPU copy of local_mat is unknown
GC/x86-64 Linux 16.10-0: compilation completed with severe errors
```

If I add an acc pragma where MPI_Bcast is in it, the compiler will show the above error saying that MPI_Bcast must have acc routine.



```
[tand@gpu047 ~]$ module load mpi/pgi_openmpi
[tand@gpu047 ~]$ mpicc -O4 -acc -ta=tesla,cuda8.0 -Minfo=accel -mcmodel=medium -fast floyd_mpi_acc.c
Copy_row:
    142, Generating implicit copyin(local_mat[n*local_k:n])
         Generating implicit copyout(row_k[:n])
    143, Loop carried dependence of row_k-> prevents parallelization
         Complex loop carried dependence of local_mat-> prevents parallelization
         Loop carried backward dependence of row_k-> prevents vectorization
         Accelerator kernel generated
        143, #pragma acc loop seq
    143, Accelerator scalar kernel generated
[tand@gpu047 ~]$ mpirun -np 16 a.out
```

In this program, we learned how to implement floyd’s algorithm with openMP, openACC and MPI. It was fun trying out the different combinations of the testing cases, but it was also painstaking to find the right and more efficient way to obtain great Gflops performance. I am hoping after the program assignment TA ken can release some sample hybrid code or instructions about how to combine these powerful tools.