Jay Butera
Parallel Matrix Multiplication

This is a review of parallelizing the matrix-matrix multiplication across threads with OpenMP, GPU with OpenACC, and across multiple nodes with OpenMPI.

When parallelizing with one node (CPU or GPU), a multiplication algorithm appropriate for shared memory is best. The single node CPU on Bridges has 35MB of cache which may not fit the entire matrix in memory. A recursive multiplication divides the matrix into four quadrants, each of which can be further divided until the sub matrix fits into the cache.

The following table holds data for matrix multiplication utilizing 8 cores with OpenMP.

Matrix Size	GFLOPS	Time	Speedup
1ME	0.549	3.915	0.75
4ME	0.555	30.990	0.78
16ME	0.521	263.652	0.78

Obviously something is wrong with the parallelization. The code actually decreases in performance as the problem size scales. To parallelize the triply nested multiplication loop, the top level loop is distributed among cores to each process a row. Rows are independent of each other, so all cores can compute using shared memory.

To build and run for OpenMP

```
[jbutera@r001 prgrm4]$ pgcc test_smm.c smm.c -mp -D USE_MP -o tsm
test_smm.c:
smm.c:
[jbutera@r001 prgrm4]$ ./tsm
elapsed time (s): 24.009444
GFLOPS: 0.715546
```

The Karp-Flatt metric can be computed:

$$e = \frac{1/.75 - 1/8}{1 - 1/8} = 1.38$$

OpenACC

For the GPU, both the rows and columns can be parallelized. For the recursive matrix multiplication, once the divided data fits into memory, the block is copied to the GPU and multiplied.

#pragma acc loop independent

The loop pragma is used for the row and column for loops. The following table displays run times and speed up for the GPU utilized code. There is no apparent speedup, there must be a problem in the build step or parallelization portion.

Matrix Size	GFLOPS	Time	Speedup
1ME	0.77	2.80	1.06
4ME	0.78	22.04	1.09
16ME	0.73	188.36	1.10

To build and run for OpenACC

```
[jbutera@br006 prgrm4]$ pgcc test_smm.c smm.c -acc -ta=tesla,cuda8.0 -D USE_ACC -o tacc
test_smm.c:
smm.c:
[jbutera@br006 prgrm4]$ ./tacc
elapsed time (s): 22.224179
GFLOPS: 0.773026
```

OpenMPI

The cannon algorithm is a good choice to parallelize across multiple nodes with OpenMPI. Each node holds a submatrix block of size $(\sqrt{N}, \sqrt{N})$ of matrices A and B to multiply. Processes are managed in a virtual cartesian grid and matrix blocks are passed between neighbors to compute the complete matrix C = AB.

To build for MPI

First navigate to the common.h file and uncomment the static declaration of variables A,B, and C, and comment the pointers.

```
/*
DTYPE A[SIZE][SIZE];
DTYPE B[SIZE][SIZE];
DTYPE C[SIZE][SIZE];
*/

DTYPE** A;//[SIZE][SIZE];
DTYPE** B;//[SIZE][SIZE];
DTYPE** C;//[SIZE][SIZE];
```

The MPI program uses static sizes. Dynamic allocation was used in the shared memory programs to test larger matrices.

Build using the command:

```
[jbutera@br006 prgrm4]$ mpicc matmul.c smm.c -lm
```

Then submit a batch script to utilize a certain number of nodes.

The batch scripts I submitted didn't finish in time so there is no data for the MPI implementation.