Lab 2

Dense Matrix Multiplication (OpenMP Implementation), Locality, and Debugging

1. Modify the code from Lab 1 to improve its locality by storing each matrix in a contiguous block of memory as a 1D array such that its rows (or columns) are concatenated. In a row-based 1D matrix, you can then access the value at index `(i,j)` at the array index `i*ncols+j`. Similarly, in a column-based 1D matrix version of the same matrix, you can then access the value at index `(i,j)` at the array index `j*nrows+i`. We call the column-based version of the matrix its transpose.

Deliverable: Provide *DenseMatMult2.cpp* which implements the array-based version of matrix multiplication. Execute the same experiments as in Lab 1 using the updated code and compare the results. Provide the log output of both experiments. Name the files *DenseMatMult.log* and *DenseMatMult2.log*.

2. Use GDB and valgrind to debug your code and verify that you have no illegal memory accesses or dynamically allocated memory that you have not de-allocated at the end of your program.

Deliverable: Provide a *screenshot* of using GDB, either though VSCode or on the command line to verify the execution of your code. Capture the output of valgrind applied to your program when complied with `-g` and using the flags `--leak-check=full --show-reachable=yes`. You can do this by piping the output of stderr to a file, e.g.,

```
valgrind --leak-check=full --show-reachable=yes ./examples 10000 -t 3
2> examples-valgring.log
```

Name the files *DenseMatMult2-GDB.png* and *DenseMatMult2-valgrind.log*.

3. Parallelize both DenseMatMult.cpp and DenseMatMult2.cpp using OpenMP.

Deliverable: Provide *DenseMatMultOmp.cpp* and *DenseMatMult2Omp.cpp* with implement the parallel versions of the programs.

4. Execute the same set of experiments as in Lab 1, using both parallel programs, using 2, 4, 8, and 16 threads.

Deliverable: Provide the log output of all experiments. Name the files *DenseMatMultOmp.log* and *DenseMatMult2Omp.log*. Note that you should execute all the experiments in the same slurm job, which should allocate enough resources for the most demanding execution of your program. You can either call your program multiple times by duplicating its execution line and passing a different number of threads (or optionally setting the number of threads through the appropriate environment variable) or use a bash loop to duplicate the execution for each number of threads.