Project 2 Report

▼ Section 1. How to compile and execute the program

To compile, I simply use the provided `CMakeList.txt` file, so the commands to compile the program is straightforward.

```
cd project2 && mkdir build && cd build
cmake ..
make -j4
```

To execute, I also modified the provided `sbatch` script to execute the program on the cluster

```
cd /path/to/project2
# If you replace the sbatch.sh with the default one, you may skip this step
mkdir results
sbatch ./src/sbatch.sh
```

▼ Section 2. Analysis of 4 Parallel Computing Models

1. **Memory Locality**

   If a matrix is large enough, each element of the matrix will be brought from the main memory to the CPU's data cache n times. The large amount of data that needs to be transferred between the CPU and the memory (i.e., cache miss) will cause the matrix multiplication to be slower than it needs to be. So one useful property of data caches we can exploit is to **decrease the amount of traffic between the CPU and memory** (i.e., decrease cache miss). The principle is that **if the program has accessed a certain memory location, a second access to the same location happening soon after the original access will be cheap**. This happens because the data we are accessing the second time may still in the data cache. Therefore, several techniques are used to optimize the memory access pattern to fully exploit the cache.

2. **DLP**

   - Memory Locality
   - SIMD
     - Using SIMD in the innermost loop allows us to load multiple elements from the matrix and do the multiplication simultaneously, which fits the task of matrix multiplication well because the nature of matrix multiplication is element-wise vector multiplication.

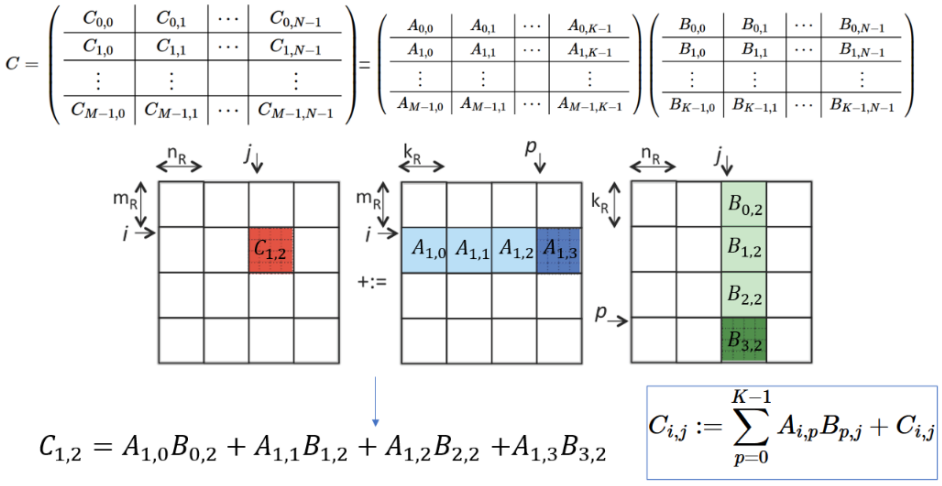
3. **TLP**

   - DLP
   - OpenMP
     - With OpenMP, we can further exploit the thread-level parallelism because now the nested loop will be assigned to multiple threads with partitioned workload to run in parallel. This optimization leverages the fact that the outer loops are independent of each other so it can be divided for different threads.

4. **PLP** (Process Level Parallelism)
    - TLP
    - MPI
        - Using MPI, we are able to push the thread-level parallelism one step further: we initiate multiple processes, each with multiple threads, to carry out the computation. This further decompose the overall task into smaller sub-tasks for each thread in each process, and then merge the results.

▼ Section 3. Optimization techniques

1. **Memory Locality**
    - **Reorder**
        - Converted the innermost loop nest **(i, j, k)** to loop nest **(i, k, j)**. By interchanging the loops over `j` and the loop over `k`, we eliminated the **strided** memory accesses pattern (i.e., cross rows) and only keeps **constant** (`a[i][k]`) and **sequential** (`c[i][j]` and `b[k][j]`) memory access patterns.
    - **Tile**



        - For the large matrix, the cache cannot fit all of its elements, especially for the matrix2 whose elements need to be accessed vertically (column major). By the principle of memory locality, we would like to reuse data already present in the cache as much as possible, so we iterate through the matrix tile by tile (i.e., in smaller blocks) to maximizing the cache line usage. In each iteration, the cache only needs to store the elements in the tile and the elements are also reused for computation.
    - **Pointer**
        - In practice, we also found that using pointer in loop to represent one line instead of directly using the naive operator `Matrix::[]` takes less time.
2. **DLP (SIMD)**

- By applying **SIMD** in the innermost loop, we broadcast one element from matrix1 and load 8 integer elements in a row from matrix2 to the AVX2 register `__m256i` and conduct element-wise multiplication. After that, we update the 8 elements in a row for the result matrix.
- The trick here is that we always use row-wise access pattern for the matrices, which is more cache friendly and is suitable for SIMD instructions.

3. **TLP (OpenMP)**
   - For OpenMP, we use a simple pragma to parallel the nested for loops.

     ```
     #pragma omp parallel for collapse(3) shared(result)
     ```

     Note that we use `collapse(3)` to instruct the OpenMP compiler to collapse the 3 outer loops into one single loop to improve the load balancing of the initiated threads. We also do not explicitly assign the number of threads for this parallel region because we want the program to allocate the number of available threads appropriately during runtime.
4. **PLP (MPI)**
   - In MPI, one optimization technique we use is sending and receiving the each row of the result matrix individually from each process. This is because the Matrix class implement matrix as a 2D array whose rows are not contiguously stored in the memory.
   - Also, we distribute the task in a more balanced manner as suggested in the last project. That is, if there are 5 rows and 3 processes, we assign the tasks as [2, 2, 1] rather than [1, 1, 3], which may hold the execution longer for that single thread assigned with more workload.

▼ Section 4. Performance Analysis

$$Speedup(S) = \frac{\text{Time taken by the sequential execution}}{\text{Time taken by the parallel version with P processors}}$$

$$Efficiency(E) = \frac{\text{Speedup(S)}}{\text{P processors}}$$

On **1024 x 1024** matrix multiplication

	Time (ms)	Processors	Speedup (S)	Efficiency (E)
Naive	8151	1	1	1
Memory Locality	692	1	11.78	11.78
Locality + SIMD	253	1	32.22	32.22
Locality + SIMD + OpenMP	45	32	181.13	5.66
Locality + SIMD + OpenMP + MPI	35	16 x 2	233.00	7.28

On **2048 x 2048** matrix multiplication

(2048 x 2048)	Time (ms)	Processors	Speedup (S)	Efficiency (E)
Naive	89250	1	1	1
Memory Locality	5634	1	15.8	15.8
Locality + SIMD	2085	1	42.81	42.81
Locality + SIMD + OpenMP	207	32	432.46	13.51
Locality + SIMD + OpenMP + MPI	209	16 x 2	427.03	13.34
*MPI (64 threads)	183	64 x 1	487.70	7.62

Grading Criteria

Compared to **1024 x 1024** matrix multiplication **Baseline**

Baseline time(%)	Time (Baseline)	Time (Best)	Percentage(%)
Naive	8165	8151	-
Memory Locality	776	692	89%
Locality + SIMD	273	253	92%
Locality + SIMD + OpenMP	41	45	109%
Locality + SIMD + OpenMP + MPI	33	35	106%

Compared to **2048 x 2048** matrix multiplication **Baseline**

Baseline time(%)	Time (Baseline)	Time (Best)	Percentage(%)
Naive	89250	89250	-
Memory Locality	6509	5634	86%
Locality + SIMD	2720	2085	76%
Locality + SIMD + OpenMP	214	207	96%
Locality + SIMD + OpenMP + MPI	184	209	113%

*the best performances are selected for multi-threads or multi-process programming model

We can safely conclude that all there significant boost for all the 4 tasks, and all of them are comparable to the baseline (≤125%).

We keep the discussion and findings at the last section after the demonstration of the profiling results.

▼ Section 5. Profiling Results

Naive

```
Available samples
34K cpu-cycles:u
8K cache-misses:u
115 page-faults:u
```

```
Samples: 34K of event 'cpu-cycles:u', Event count (approx.): 24291731834
  Children      Self  Command  Shared Object        Symbol
+   73.06%    73.04%  naive    naive                [.] matrix_multiply
+   17.72%    17.72%  naive    naive                [.] Matrix::operator[]
+    6.89%     6.89%  naive    naive                [.] Matrix::operator[]
+    0.78%     0.78%  naive    libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator
```

```
Samples: 8K of event 'cache-misses:u', Event count (approx.): 1105696
  Children      Self  Command  Shared Object        Symbol
+   73.50%    73.50%  naive    naive                [.] Matrix::operator[]
+    9.22%     9.19%  naive    naive                [.] matrix_multiply
+    8.46%     8.46%  naive    libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_
+    4.77%     0.00%  naive    [unknown]            [.] 0x0000000000000006
```

```
Samples: 115  of event 'page-faults:u', Event count (approx.): 10076
  Children      Self  Command  Shared Object     Symbol
+   51.32%    51.32%  naive    naive             [.] matrix_multiply
+   38.24%     0.00%  naive    [unknown]         [.] 0000000000000000
+   28.81%    28.81%  naive    libc-2.17.so      [.] _int_malloc
+    9.43%     9.43%  naive    libc-2.17.so      [.] _int_free
+    5.37%     0.18%  naive    ld-2.17.so        [.] _dl_sysdep_start
+    4.86%     4.86%  naive    naive             [.] Matrix::saveToFile
```

Memory Locality

```
Available samples
3K cpu-cycles:u
1K cache-misses:u
23 page-faults:u
```

```
Samples: 3K of event 'cpu-cycles:u', Event count (approx.): 2575356741
  Children      Self  Command   Shared Object        Symbol
+   77.42%    77.39%  locality  locality             [.] matrix_multiply_locality
+    7.87%     7.87%  locality  libstdc++.so.6.0.19  [.] std::num_get<char, std::i
+    3.92%     0.00%  locality  [unknown]            [.] 0x00401f0fffffff74
```

```
Samples: 1K of event 'cache-misses:u', Event count (approx.): 515300
  Children      Self  Command   Shared Object        Symbol
+   69.14%    69.14%  locality  locality             [.] matrix_multiply_locality
+   13.41%    13.41%  locality  libstdc++.so.6.0.19  [.] std::num_get<char, std::istre
+    8.48%     0.00%  locality  [unknown]            [.] 0x00401f0fffffff74
+    8.48%     0.00%  locality  libstdc++.so.6.0.19  [.] virtual thunk to std::basic_c
+    8.48%     0.00%  locality  [unknown]            [.] 0x0000000000000006
+    8.40%     1.20%  locality  libstdc++.so.6.0.19  [.] std::num_put<char, std::ostre
+    1.88%     1.88%  locality  libstdc++.so.6.0.19  [.] 0x000000000008a5fa
```

```
Samples: 23  of event 'page-faults:u', Event count (approx.): 3521
  Children      Self  Command   Shared Object      Symbol
+   78.47%    78.47%  locality  libc-2.17.so       [.] _int_malloc
+   78.47%     0.00%  locality  [unknown]          [.] 0000000000000000
+   14.83%     0.48%  locality  ld-2.17.so         [.] _dl_sysdep_start
+   10.79%    10.79%  locality  ld-2.17.so         [.] _dl_relocate_object
+   10.79%     0.00%  locality  ld-2.17.so         [.] dl_main
+    6.50%     6.50%  locality  libc-2.17.so       [.] sysmalloc
+    3.55%     3.55%  locality  ld-2.17.so         [.] _dl_count_modids
+    0.65%     0.00%  locality  ld-2.17.so         [.] _dl_start_user
```

SIMD

```
Available samples
1K cpu-cycles:u
880 cache-misses:u
16 page-faults:u
```

```
Samples: 1K of event 'cpu-cycles:u', Event count (approx.): 1174856956
  Children      Self  Command  Shared Object        Symbol
+   53.17%     0.00%  simd     simd                 [.] main
+   52.55%    52.50%  simd     simd                 [.] matrix_multiply_simd
+   15.18%    15.18%  simd     libstdc++.so.6.0.19  [.] std::num_get<char, std::istrea
+    7.92%     0.00%  simd     [unknown]            [.] 0x00401f0fffffff74
+    7.92%     0.00%  simd     libstdc++.so.6.0.19  [.] virtual thunk to std::basic_o
+    7.92%     0.00%  simd     [unknown]            [.] 0x0000000000000006
+    6.67%     0.49%  simd     libstdc++.so.6.0.19  [.] std::num_put<char, std::ostrea
+    6.39%     0.00%  simd     [unknown]            [.] 0000000000000000
+    4.67%     4.67%  simd     libstdc++.so.6.0.19  [.] std::istream::sentry::sentry
```

```
Samples: 880  of event 'cache-misses:u', Event count (approx.): 178628
  Children      Self  Command  Shared Object        Symbol
+   64.88%     0.00%  simd     simd                 [.] main
-   63.13%    63.13%  simd     simd                 [.] matrix_multiply_simd
     main
     matrix_multiply_simd
+   11.97%    11.97%  simd     libstdc++.so.6.0.19  [.] std::num_get<char, std::istre
+    6.47%     0.00%  simd     [unknown]            [.] 0x0000000000000006
+    6.19%     0.00%  simd     [unknown]            [.] 0x00401f0fffffff74
+    6.19%     0.00%  simd     libstdc++.so.6.0.19  [.] virtual thunk to std::basic_o
+    6.03%     0.60%  simd     libstdc++.so.6.0.19  [.] std::num_put<char, std::ostre
+    4.23%     0.00%  simd     ld-2.17.so           [.] _dl_sysdep_start
```

```
Samples: 16  of event 'page-faults:u', Event count (approx.): 3562
  Children      Self  Command  Shared Object      Symbol
+   78.83%    78.83%  simd     libc-2.17.so       [.] _int_malloc
+   78.83%     0.00%  simd     [unknown]          [.] 0000000000000000
+   20.27%    20.27%  simd     ld-2.17.so         [.] memcmp
+   20.27%     0.00%  simd     ld-2.17.so         [.] 0x00007f53accf2fba
+   20.27%     0.00%  simd     ld-2.17.so         [.] _dl_map_object
+    0.87%     0.00%  simd     ld-2.17.so         [.] _dl_start_user
+    0.73%     0.73%  simd     ld-2.17.so         [.] _dl_sysdep_start
     0.14%     0.14%  simd     ld-2.17.so         [.] _dl_start
     0.03%     0.03%  simd     ld-2.17.so         [.] _start
```

OpenMP

```
Available samples
6K cpu-cycles:u
2K cache-misses:u
76 page-faults:u
```

```
Samples: 2K of event 'cache-misses:u', Event count (approx.): 1283457
  Children      Self  Command  Shared Object        Symbol
-   88.08%    88.08%  openmp   openmp               [.] matrix_multiply_openmp
   + 82.14% 0x894855f872ff41e0
   + 5.94% 0x1000000053
+   83.28%     0.00%  openmp   [unknown]            [.] 0x894855f872ff41e0
+   83.26%     0.00%  openmp   libgomp.so.1.0.0     [.] 0x00007f5c780ca405
+    6.99%     0.00%  openmp   [unknown]            [.] 0x0000000000000400
+    5.94%     0.00%  openmp   [unknown]            [.] 0x0000001000000053
+    5.94%     0.00%  openmp   [unknown]            [.] 0x0000000000cc94d0
+    5.94%     0.00%  openmp   [unknown]            [.] 0x00007ffec0b3cc50
+    5.94%     0.00%  openmp   libgomp.so.1.0.0     [.] GOMP_parallel
```

```
Samples: 76  of event 'page-faults:u', Event count (approx.): 3614
  Children      Self  Command  Shared Object        Symbol
+   83.73%     0.00%  openmp   [unknown]            [.] 0000000000000000
+   81.99%    81.99%  openmp   libc-2.17.so         [.] _int_malloc
+    5.40%     0.00%  openmp   ld-2.17.so           [.] _dl_sysdep_start
+    5.40%     0.00%  openmp   ld-2.17.so           [.] dl_main
+    3.51%     2.41%  openmp   ld-2.17.so           [.] _dl_relocate_object
+    2.60%     2.60%  openmp   libpthread-2.17.so   [.] pthread_create@@GLIBC_2.2.5
+    2.60%     0.00%  openmp   [unknown]            [.] 0x0000000000000001
```

MPI

For MPI, I select a process with average performance to demonstrate.

```
Performance counter stats for './build/src/mpi 2 ./matrices/matrix5.txt ./matrices/matrix6.txt ./result
s/mpi1.txt':

     1,023,099,511      cpu-cycles:u
           250,084      cache-misses:u
             5,361      page-faults:u
```

```
        0.414043168 seconds time elapsed

        0.373309000 seconds user
        0.033846000 seconds sys
```

Also, since we cannot use `perf record` to profile the individual MPI ranks and we can only use `perf stat`, **I kept the perf stat results as the profiling data for MPI rather than the perf record data.**

Summary

	Naive	Locality	+SIMD	+OpenMP*	+MPI*
CPU Cycles	34K	3K	1K	6K	1023099K
Cache Miss	8K	1K	880	2K	250K
Page Fault	115	23	16	76	5361

From the summary table, we can observe that there is a significant improvement on the CPU Cycles, Cache miss and Page Fault after optimized with Memory locality and SIMD.

However, the statistics for OpenMP and MPI may be on a different criterion and is not directly comparable with the previous ones which only have 1 process and 1 thread during the execution. OpenMP and MPI perf results seems to **aggregate** all the cycles and cache misses from all the threads, so the number is significantly higher. However, the actual situation is that these processes and threads are running in parallel, especially for the process that are on different cores for MPI.

▼ Section 6. Findings & Discussion

1. **Pointer Access**
    - In practice, we noticed that from the profiling results that operator `Matrix::[]` takes a lot of time, so we change it to pointer to access the matrix values, which significantly improves the execution efficiency.
2. **About Tiled Multiplication**
    - **The choice of *tile size* constant is crucial for good performance**. For different tile sizes, the performance gap can range from 100ms to 1000ms. The most appropriate sizes based on my experiment is from 128 to 256.
    - I also noticed that **different parallel programming models have different tile sizes for their best performance.** I think this is because that the compiler will change the overall structure of code each time when we introduce a new parallel computing optimization techniques, even if the matrix multiplication function stays the same.
3. **MPI vs OpenMP**
    - As shown is the performance metric, there is no significant differences between MPI + OpenMP and OpenMP alone. This may due to some reasons. One reason can be that the communication overhead between the process. Specifically, due to the limitation in the Matrix class implementation, each process needs to send the result matrix **row by row** to the master node rather than send it as a whole because these rows are not contiguous in memory space. Another reason is that the total number of threads across all the processes stays the same, and we run programs in one single node.