Report

How to compile and execute program

All the 4 programs can be directly compiled and executed by using "CMakeLists.txt" and "sbach.sh" files. In the file "CMakeLists.txt", in order to compile all the 4 programs (also the dependency files), the following need to be included in the CMakeList.txt in the src folder

For locality.cpp

```
add_executable(locality
        locality.cpp
        matrix.cpp matrix.hpp)
target_compile_options(locality PRIVATE -O2)
```

For simd.cpp

```
add_executable(simd
        simd.cpp
        matrix.cpp matrix.hpp)
target_compile_options(simd PRIVATE -O2 -mavx2)
```

For openmp.cpp

```
add_executable(openmp
        openmp.cpp
        matrix.cpp matrix.hpp)
target_compile_options(openmp PRIVATE -O2 -fopenmp -mavx2)
target_include_directories(openmp PRIVATE ${OpenMP_CXX_INCLUDE_DIRS})
target_link_libraries(openmp PRIVATE ${OpenMP_CXX_LIBRARIES})
```

For mpi.cpp

```
add_executable(mpi
        mpi.cpp
        matrix.cpp matrix.hpp)
target_compile_options(mpi PRIVATE -O2 -fopenmp -mavx2)
target_include_directories(mpi PRIVATE ${MPI_CXX_INCLUDE_DIRS}
${OpenMP_CXX_INCLUDE_DIRS})
target_link_libraries(mpi ${MPI_LIBRARIES} ${OpenMP_CXX_LIBRARIES})
```

To successfully run the programs, you also need a sbatch file, following should be added to the file

```
 # For running locality.cpp
srun -n 1 --cpus-per-task 1 ${CURRENT_DIR}/../build/src/locality
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt
```

```
# For running simd.cpp
srun -n 1 --cpus-per-task 1 ${CURRENT_DIR}/../build/src/simd
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt

# For running openmp.cpp
for num_cores in 1 2 4 8 16 32
do
  srun -n 1 --cpus-per-task $num_cores ${CURRENT_DIR}/../build/src/openmp
$num_cores ${CURRENT_DIR}/../matrices/matrix5.txt
${CURRENT_DIR}/../matrices/matrix6.txt ${CURRENT_DIR}/../build/result.txt
done

# For running mpi.cpp
srun -n 1 --cpus-per-task 32 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 32
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt

srun -n 2 --cpus-per-task 16 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 16
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt

srun -n 4 --cpus-per-task 8 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 8
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt
srun -n 8 --cpus-per-task 4 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 4
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt

srun -n 16 --cpus-per-task 2 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 2
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt

srun -n 32 --cpus-per-task 1 --mpi=pmi2 ${CURRENT_DIR}/../build/src/mpi 1
${CURRENT_DIR}/../matrices/matrix5.txt ${CURRENT_DIR}/../matrices/matrix6.txt
${CURRENT_DIR}/../build/result.txt
```

1. To compile the code, first configure all the CMake setup by creating a new directory “build” and in that directory execute the command “cmake ..”.
2. After successful configuration, execute the command “make -j<number of threads>” to compile the files.

3. Go to the directory “project2” and execute “sbatch ./src/sbach.sh”, the 4 programs will be executed.
4. The expected output will appear in the result.txt inside the build folder, and the command line output will locate in the file “Project2-Results.txt” in the directory where “sbatch ./src/sbach.sh” is executed.

How does each parallel programming model do computation in parallel

For memory locality, it is not a parallel programming model, but it can largely improve the performance compared with the naive implementation through minimizing cache misses and increasing the number of contiguous memory access.

SIMD

SIMD allows to perform the same operation on the same data elements simultaneously. As a result, when multiply matrixes, use __m256i data type to hold eight 32-bit integer values. This enables processing eight elements of the matrix at a time.

OpenMP

OpenMP allows to use multithread with shared memory to perform calculation in parallel. By using multiple threads through using pragma, each thread will be assigned a different range of loop to do its computation. It allows distribution of work into different threads to boost performance.

MPI

MPI allows communication between multiple processes through receiving and sending messages. Matrices can be partitioned into smaller submatrices, and each process is responsible for computing a subset of the resulting submatrix. The computation is then distributed among the processes, and the results are gathered and combined to form the final result.

Optimizations tried to speed up parallel programs, and how does them work

Memory locality

The program is optimizing matrix multiplication by considering memory locality and avoiding cache misses. I reorder the nested loop, resulting in changing the order of the triple nested loop. By reordering the loops, the memory access patterns are largely improved and increase the data locality. Instead of directly accessing single element in the matrix at a time, I choose to access a single row, which can minimize the cache misses since single row allows memory contiguous access. The order of the loops is k -> i -> j. This ordering allows for better utilization of memory, since I can loop in a row, that is, continuous memory access every time. In the code, the matrix multiplication is performed using three nested loops: k, i, and j. The outer loop iterates over the columns of matrix1 and rows of matrix2, while the inner loops iterate over the rows of matrix1 and columns of matrix2. This allows for efficient traversal of the matrices and accessing the required elements.Inside the innermost loop, the multiplication operation is performed between the corresponding elements of matrix1 and matrix2, and the result is accumulated in the corresponding element of the result matrix.

SIMD

Similar with the memory locality program, I also optimize the memory access and minimize the caches misses through reordering the loop and accessing one row at a time. Other than that, I adopt SIMD to handle 8 elements at a time instead of one element compared to the memory locality optimized program. This also reduce the number of innermost loop iteration. First, I load 8 same elements from matrix 1 to _m256i register and load 8 consecutive elements from matrix 2 to _m256i register. At the same time, the program loads the row of result matrix to store the calculation result in the innermost loop. As a result, it allows processing 8 elements at once, which highly increase the performance.

OpenMP

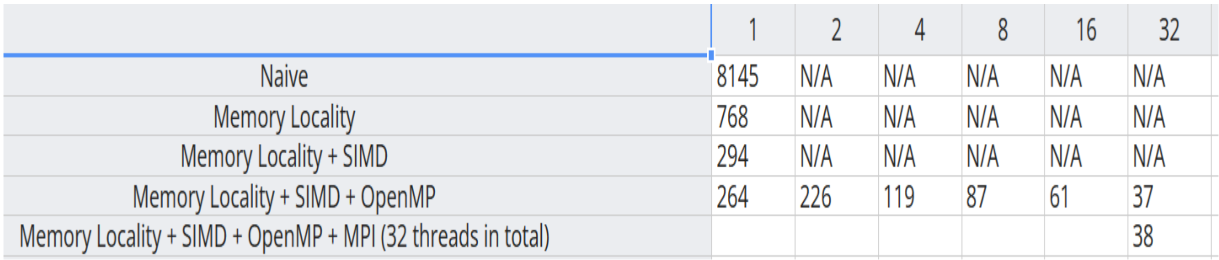
Based on the SIMD implementation, I sightly change the order of the loop in order to let the OpenMP distribute tasks based on i which is the row of the matrix1 and column of the matrix 2. I use loop-level parallelism by using #pragma omp parallel for. This optimization focuses on parallelizing loops by dividing the iterations among multiple threads or processes. Each thread is responsible only a portion of “i”, which means a portion of calculation.

MPI

By using MPI, I divide the matrices to optimize the performance. The matrices are partitioned into smaller submatrices, and each process is assigned a subset of these submatrices. The submatrices are distributed among the processes using MPI communication routines. Each process receives the necessary submatrices from other processes. Each process performs the matrix multiplication operation on its assigned submatrices using SIMD instructions and OpenMP optimization techniques. The computation is done independently on each process. After the computation is complete, the resulting submatrices are collected and combined using MPI communication routines. The final result is obtained by aggregating the submatrices.

Experimental Results and Numerical Analysis

For 1024*1024 matrix multiplication



	1	2	4	8	16	32
Naive	8145	N/A	N/A	N/A	N/A	N/A
Memory Locality	768	N/A	N/A	N/A	N/A	N/A
Memory Locality + SIMD	294	N/A	N/A	N/A	N/A	N/A
Memory Locality + SIMD + OpenMP	264	226	119	87	61	37
Memory Locality + SIMD + OpenMP + MPI (32 threads in total)						38

Notice: I exclude naïve program performance in the graph due to the scale issues of the axis.

Memory locality + SIMD + OpenMP + MPI (performance in milliseconds):

1 process 32 threads: 43

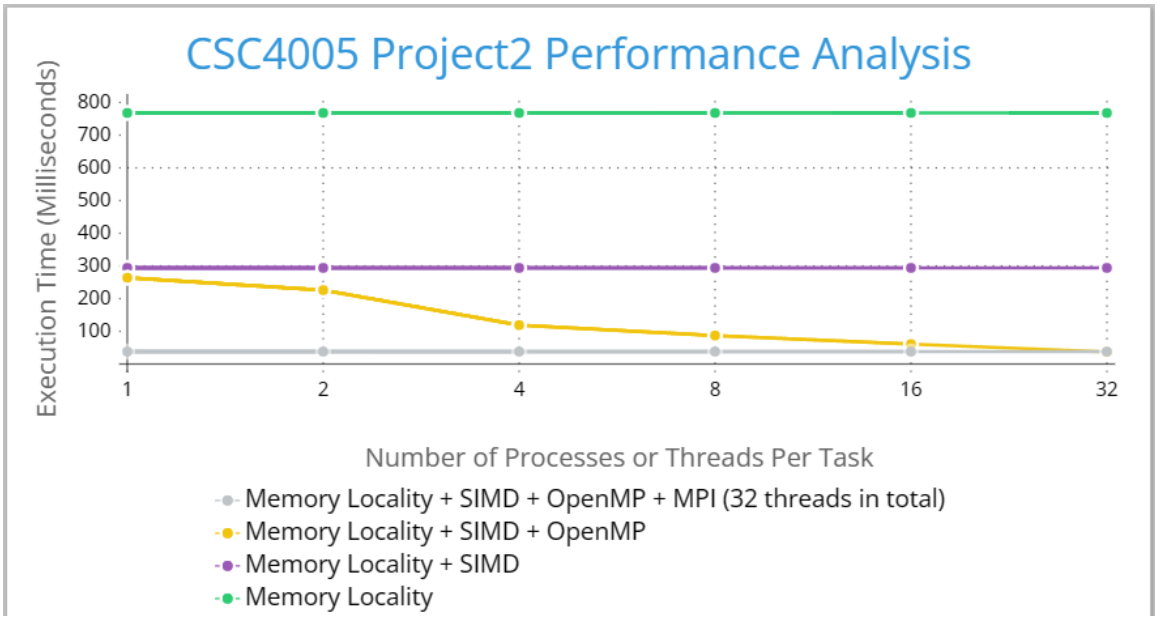
2 processes 16 threads: 45

4 processes 8 threads: 36

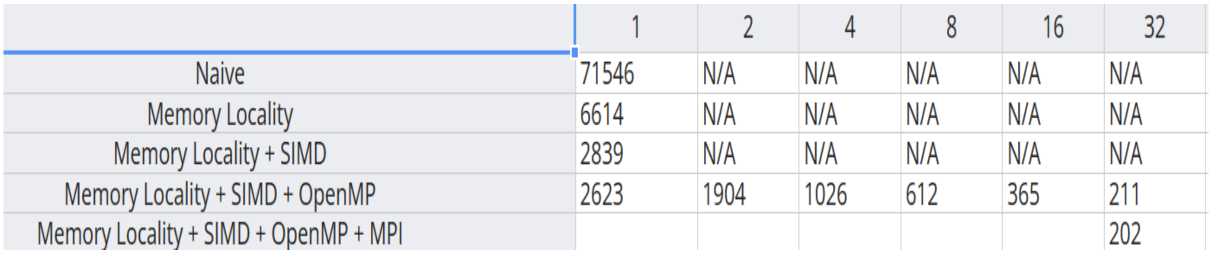
8 processes 4 threads: 36

16 processes 2 threads: 38

32 processes 1 thread: 66



For 2048*2048 matrix multiplication



	1	2	4	8	16	32
Naive	71546	N/A	N/A	N/A	N/A	N/A
Memory Locality	6614	N/A	N/A	N/A	N/A	N/A
Memory Locality + SIMD	2839	N/A	N/A	N/A	N/A	N/A
Memory Locality + SIMD + OpenMP	2623	1904	1026	612	365	211
Memory Locality + SIMD + OpenMP + MPI						202

Memory locality + SIMD + OpenMP + MPI (performance in milliseconds):

1 process 32 threads: 202

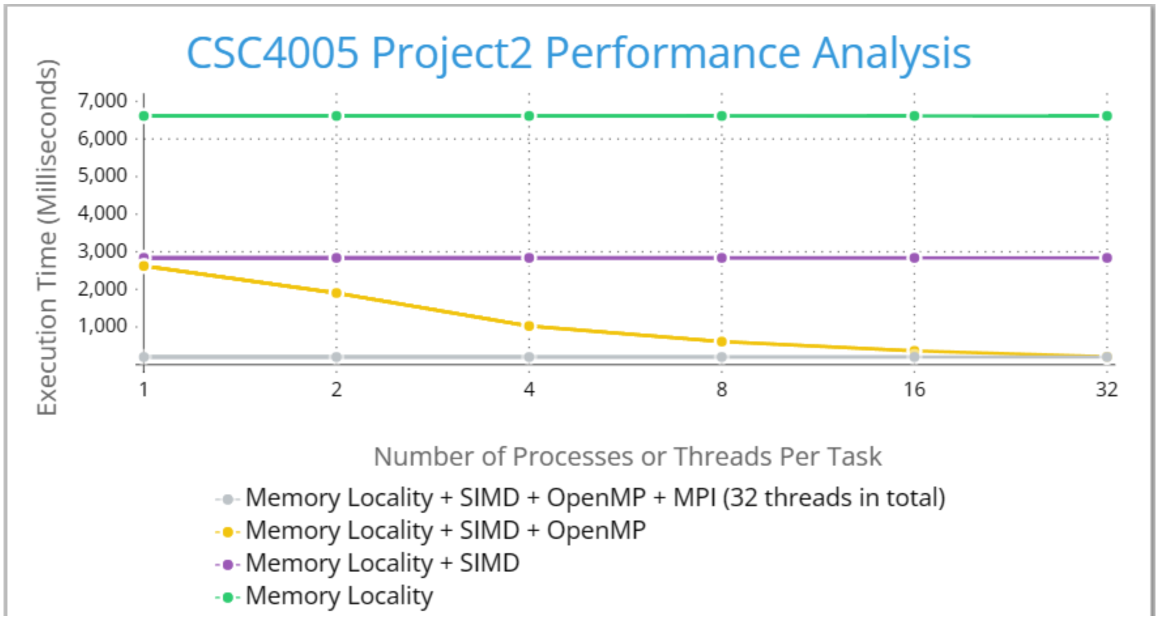
2 processes 16 threads: 226

4 processes 8 threads: 293

8 processes 4 threads: 213

16 processes 2 threads: 245

32 processes 1 thread: 475



Speedup relative to naïve (1024*1024)

Memory locality: 10.60

Memory locality + SIMD: 27.7

Memory locality + SIMD + OpenMP: 30.85, 36.03, 68.45, 93.62, 133.52, 220.14 (1, 2, 4, 8, 16, 32 cores respectively)

Memory locality + SIMD + OpenMP + MPI: 214.34 (32 threads in total: (1) 8 processes – 4 threads and 4 processes – 8 threads both produce this speedup)

Efficiency

Memory locality + SIMD + OpenMP: 30.85%, 18.02%, 17.11%, 11.7%, 8.35%, 6.88%

Memory locality + SIMD + OpenMP + MPI: 6.70%

Speedup relative to naïve (2048*2048)

Memory locality: 10.81

Memory locality + SIMD: 25.20

Memory locality + SIMD + OpenMP: 27.28, 37.58, 69.73, 116.91, 196.02, 339.08

Memory locality + SIMD + OpenMP + MPI: 354.19 (32 threads in total: 1 processes – 32 threads produces this speedup)

Efficiency

Memory locality + SIMD + OpenMP: 27.28%, 18.79%, 17.43%, 14.61%, 12.25%, 10.60%

Memory locality + SIMD + OpenMP + MPI: 11.07%

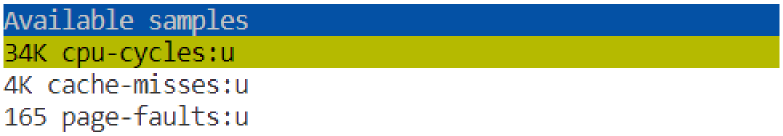
Finding from experimental results

When applying larger data sets, such as 2048*2048 matrix, for OpenMP (shorthand for "Memory locality + SIMD + OpenMP") and MPI (Memory locality + SIMD + OpenMP + MPI) tend to lower speedup compared with relatively smaller data sets, such as 1024*1024 matrix for single core (process or thread). It indicates that it probability reach the limits of single core performance. As a result, while it uses more cores (processes or threads) the speedup is higher compared to 1024*1024, which can prove the limitation of single core performance.

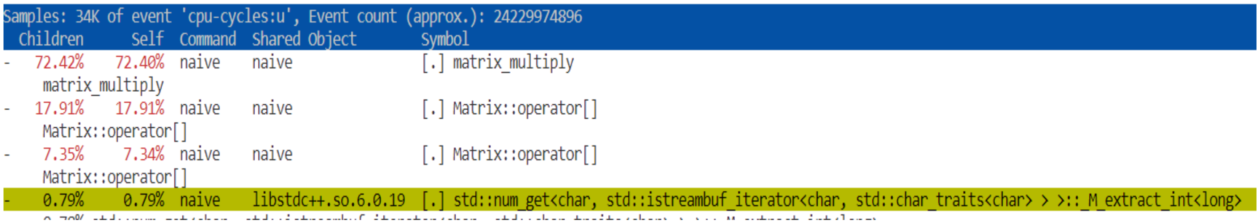
For MPI, the performance of different processes and threads varies a lot at each time I conduct the experiment. For 1024 * 1024 matrix multiplication, sometimes the performance of 32 processes, 1 thread is lower than the performance of 16 processes, 2 threads. It may be due to the communication overhead in MPI. As the number of processes increases, the amount of communication required also increases. With 32 processes, there may be more frequent and larger message exchanges compared to 16 processes, increasing communication overhead. This overhead becomes a bottleneck and limit the overall performance.

Profiling

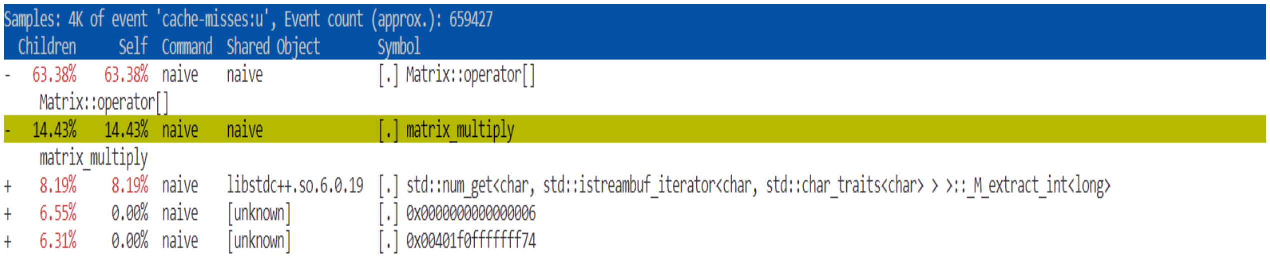
Naïve



```
Available samples
34K cpu-cycles:u
4K cache-misses:u
165 page-faults:u
```



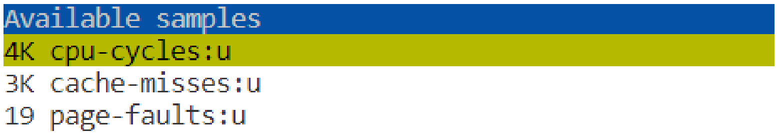
```
Samples: 34K of event 'cpu-cycles:u', Event count (approx.): 24229974896
  Children      Self  Command  Shared Object         Symbol
-   72.42%    72.40%  naive    naive                 [.] matrix_multiply
     matrix_multiply
-   17.91%    17.91%  naive    naive                 [.] Matrix::operator[]
     Matrix::operator[]
-    7.35%     7.34%  naive    naive                 [.] Matrix::operator[]
     Matrix::operator[]
-    0.79%     0.79%  naive    libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
     0.79% std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
```



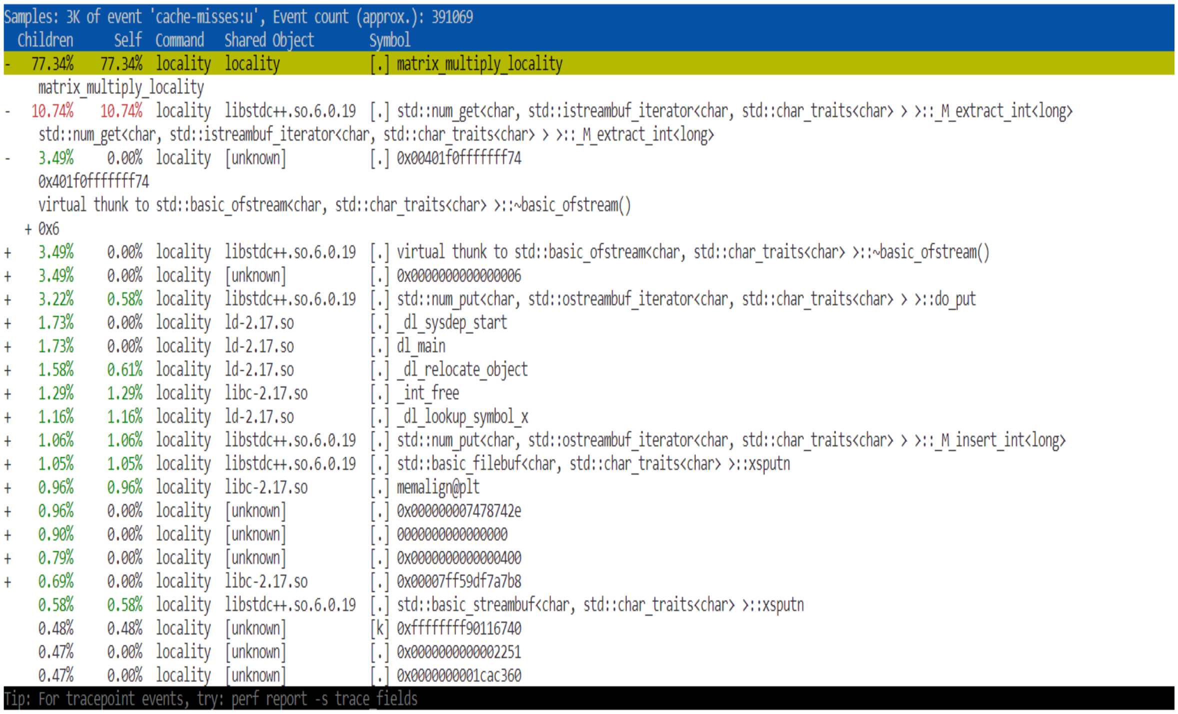
```
Samples: 4K of event 'cache-misses:u', Event count (approx.): 659427
  Children      Self  Command  Shared Object         Symbol
-   63.38%    63.38%  naive    naive                 [.] Matrix::operator[]
     Matrix::operator[]
-   14.43%    14.43%  naive    naive                 [.] matrix_multiply
     matrix_multiply
+    8.19%     8.19%  naive    libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
+    6.55%     0.00%  naive    [unknown]             [.] 0x0000000000000006
+    6.31%     0.00%  naive    [unknown]             [.] 0x00401f0fffffff74
```

As we can see, in the naïve program the CPU cycle is extremely high as is the page-faults. The next two pictures all reveals that cpu did a lot of work inside the matrix_multiply function especially in accessing elements as indicated by Matrix:operator[]. It also indicates that here the cache misses are high. The program wastes a lot of time in accessing the elements from the array.

Memory locality



```
Available samples
4K cpu-cycles:u
3K cache-misses:u
19 page-faults:u
```



```
Samples: 3K of event 'cache-misses:u', Event count (approx.): 391069
  Children      Self  Command   Shared Object         Symbol
-   77.34%    77.34%  locality  locality              [.] matrix_multiply_locality
     matrix_multiply_locality
-   10.74%    10.74%  locality  libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
     std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
-    3.49%     0.00%  locality  [unknown]             [.] 0x00401f0fffffff74
     0x401f0fffffff74
     virtual thunk to std::basic_ofstream<char, std::char_traits<char> >::~basic_ofstream()
   + 0x6
+    3.49%     0.00%  locality  libstdc++.so.6.0.19  [.] virtual thunk to std::basic_ofstream<char, std::char_traits<char> >::~basic_ofstream()
+    3.49%     0.00%  locality  [unknown]             [.] 0x0000000000000006
+    3.22%     0.58%  locality  libstdc++.so.6.0.19  [.] std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::do_put
+    1.73%     0.00%  locality  ld-2.17.so            [.] _dl_sysdep_start
+    1.73%     0.00%  locality  ld-2.17.so            [.] dl_main
+    1.58%     0.61%  locality  ld-2.17.so            [.] _dl_relocate_object
+    1.29%     1.29%  locality  libc-2.17.so          [.] _int_free
+    1.16%     1.16%  locality  ld-2.17.so            [.] _dl_lookup_symbol_x
+    1.06%     1.06%  locality  libstdc++.so.6.0.19  [.] std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>
+    1.05%     1.05%  locality  libstdc++.so.6.0.19  [.] std::basic_filebuf<char, std::char_traits<char> >::xsputn
+    0.96%     0.96%  locality  libc-2.17.so          [.] memalign@plt
+    0.96%     0.00%  locality  [unknown]             [.] 0x000000007478742e
+    0.90%     0.00%  locality  [unknown]             [.] 0000000000000000
+    0.79%     0.00%  locality  [unknown]             [.] 0x0000000000000400
+    0.69%     0.00%  locality  libc-2.17.so          [.] 0x00007ff59df7a7b8
     0.58%     0.58%  locality  libstdc++.so.6.0.19  [.] std::basic_streambuf<char, std::char_traits<char> >::xsputn
     0.48%     0.48%  locality  [unknown]             [k] 0xffffffff90116740
     0.47%     0.00%  locality  [unknown]             [.] 0x0000000000002251
     0.47%     0.00%  locality  [unknown]             [.] 0x000000001cac360
Tip: For tracepoint events, try: perf report -s trace_fields
```

By optimizing the way of accessing elements in the array through memory locality, the CPU cycle reduce significantly by 30k cycles. From the high reduction of page fault, we can notice that the accessing memory patterns has been optimized since there are less page swapping. The cache misses also reduce by 1k.

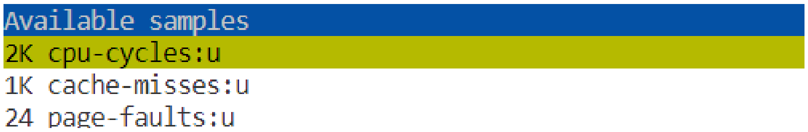
Optimization compared to naïve program:

Number of CPU cycle reduce: 30k

Number of cache misses reduce: 1k

Number of page faults reduce: 146

Memory locality + SIMD



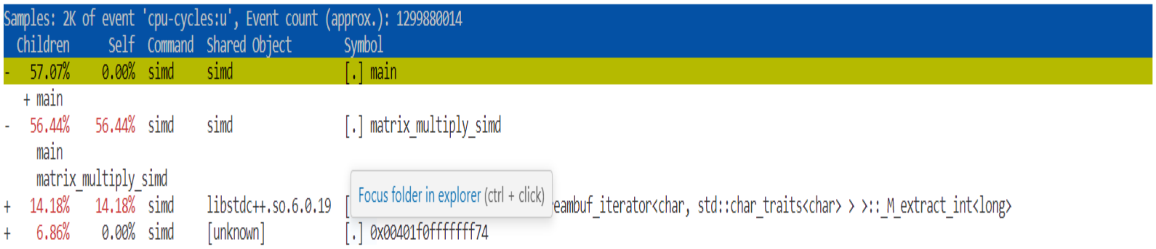
```
Available samples
2K cpu-cycles:u
1K cache-misses:u
24 page-faults:u
```

Optimization compared to naïve program:

Number of CPU cycle reduce: 32k

Number of cache misses reduce: 3k

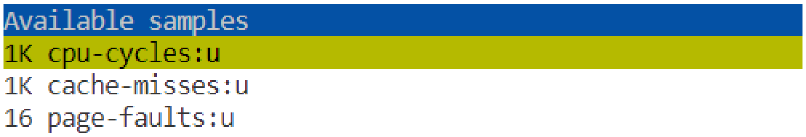
Number of page faults reduce: 141



```
Samples: 2K of event 'cpu-cycles:u', Event count (approx.): 1299880014
  Children      Self  Command  Shared Object        Symbol
-   57.07%     0.00%  simd     simd                 [.] main
   + main
-   56.44%    56.44%  simd     simd                 [.] matrix_multiply_simd
     main
     matrix_multiply_simd
+   14.18%    14.18%  simd     libstdc++.so.6.0.19  [ Focus folder in explorer (ctrl + click) eambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
+    6.86%     0.00%  simd     [unknown]            [.] 0x00401f0fffffff74
```

Memory locality + SIMD + OpenMP

1 thread



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

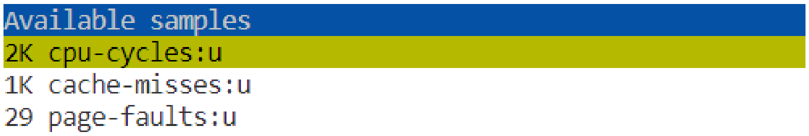
Optimization compared to naïve program:

Number of CPU cycle reduce: 33k

Number of cache misses reduce: 3k

Number of page faults reduce: 139

2 threads



```
Available samples
2K cpu-cycles:u
1K cache-misses:u
29 page-faults:u
```

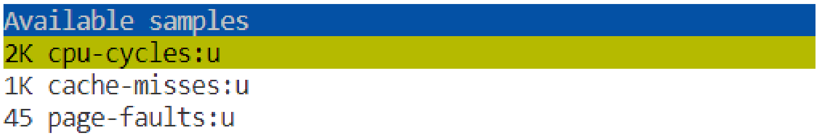
Optimization compared to naïve program:

Number of CPU cycle reduce: 32k

Number of cache misses reduce: 3k

Number of page faults reduce: 136

4 threads



```
Available samples
2K cpu-cycles:u
1K cache-misses:u
45 page-faults:u
```

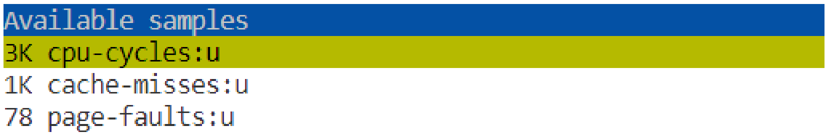
Optimization compared to naïve program:

Number of CPU cycle reduce: 32k

Number of cache misses reduce: 3k

Number of page faults reduce: 120

8 threads



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

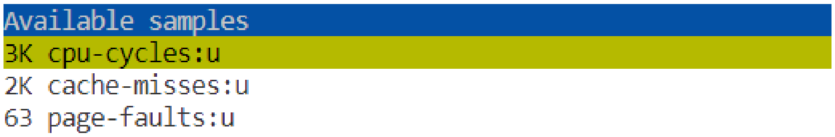
Optimization compared to naïve program:

Number of CPU cycle reduce: 31k

Number of cache misses reduce: 3k

Number of page faults reduce: 87

16 threads



```
Available samples
3K cpu-cycles:u
2K cache-misses:u
63 page-faults:u
```

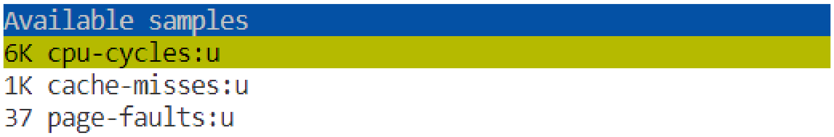
Optimization compared to naïve program:

Number of CPU cycle reduce: 31k

Number of cache misses reduce: 2k

Number of page faults reduce: 102

32 threads



```
Available samples
6K cpu-cycles:u
1K cache-misses:u
37 page-faults:u
```

Optimization compared to naïve program:

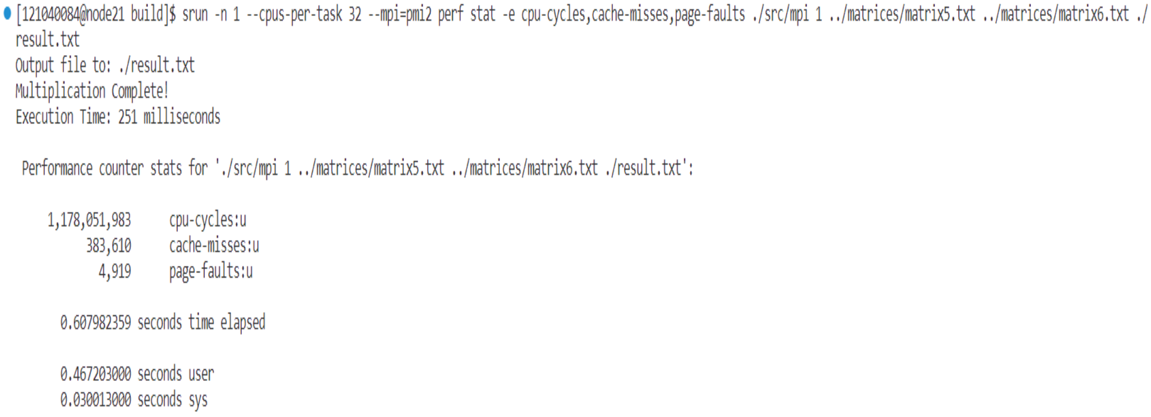
Number of CPU cycle reduce: 28k

Number of cache misses reduce: 3k

Number of page faults reduce: 128

Memory locality + SIMD + OpenMP + MPI

1 process 32 threads



```
● [121040084@node21 build]$ srun -n 1 --cpus-per-task 32 --mpi=pmi2 perf stat -e cpu-cycles,cache-misses,page-faults ./src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./
  result.txt
  Output file to: ./result.txt
  Multiplication Complete!
  Execution Time: 251 milliseconds

   Performance counter stats for './src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./result.txt':

      1,178,051,983      cpu-cycles:u
            383,610      cache-misses:u
              4,919      page-faults:u

        0.607982359 seconds time elapsed

        0.467203000 seconds user
        0.030013000 seconds sys
```

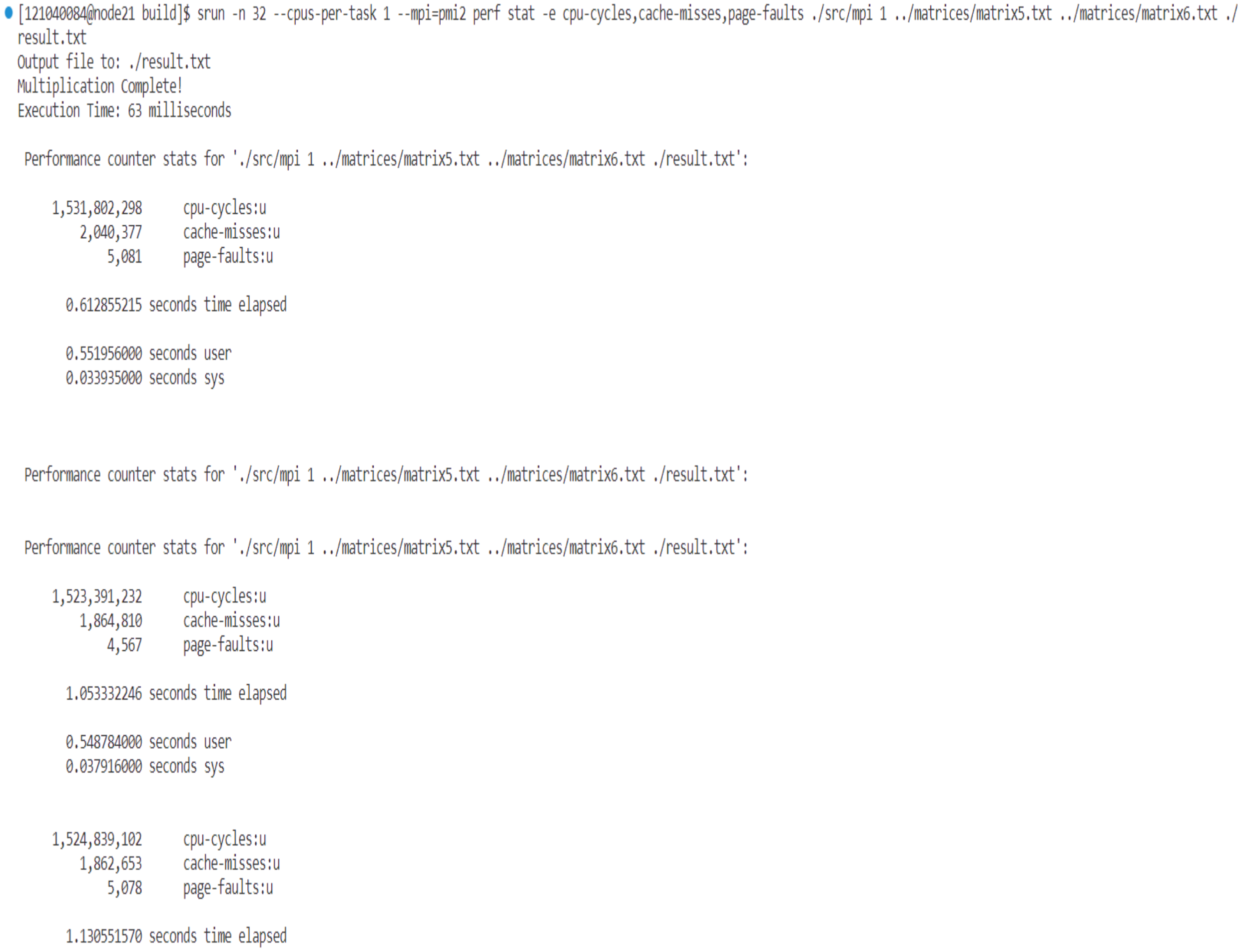
Optimization compared to naïve program:

Number of CPU cycle reduce: no CPU cycle reduce

Number of cache misses reduce: no cache misses reduce

Number of page faults reduce: no page faults reduce

32 processes 1 thread



```
● [121040084@node21 build]$ srun -n 32 --cpus-per-task 1 --mpi=pmi2 perf stat -e cpu-cycles,cache-misses,page-faults ./src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./
  result.txt
  Output file to: ./result.txt
  Multiplication Complete!
  Execution Time: 63 milliseconds

   Performance counter stats for './src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./result.txt':

       1,531,802,298      cpu-cycles:u
           2,040,377      cache-misses:u
               5,081      page-faults:u

         0.612855215 seconds time elapsed

         0.551956000 seconds user
         0.033935000 seconds sys



   Performance counter stats for './src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./result.txt':


   Performance counter stats for './src/mpi 1 ../matrices/matrix5.txt ../matrices/matrix6.txt ./result.txt':

       1,523,391,232      cpu-cycles:u
           1,864,810      cache-misses:u
               4,567      page-faults:u

         1.053332246 seconds time elapsed

         0.548784000 seconds user
         0.037916000 seconds sys


       1,524,839,102      cpu-cycles:u
           1,862,653      cache-misses:u
               5,078      page-faults:u

         1.130551570 seconds time elapsed
```

Interesting finding: page fault increases as threads number increases until the number of threads is 8. If the thread number is more than 8, the number of page fault tends to decrease.

Since MPI has communication between processes the CPU cycles spent at it will increase significantly.