Week6 Lab2 Report

111062117, Hsiang-Sheng Huang

April 16, 2025

Matrix Multiplication Profiler

Observations

Execute the three files directly to measure the execution time.

```
scteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ time ./mat_mul_a

real    2m11.101s
user    12m58.101s
sys     0m0.178s
scteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ time ./mat_mul_b

real    0m21.059s
user    2m5.668s
sys     0m0.101s
scteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ time ./mat_mul_c

real    0m3.258s
user    0m18.975s
sys     0m0.085s
```

Running APS

Run APS to observe the elapsed time and memory usage.

```
^Cscteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ aps --result-dir=aps_result-a -c omp ./mat_mul_a
| Summary information
|--------------------------------------------------------------------
  Application                 : mat_mul_a
  Report creation date        : 2025-04-16 15:34:51
  Used statistics             : /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_result-a/scteam01a
|
| Your application might underutilize the available logical CPU cores
| because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel(R) VTune(TM
) Profiler to discover why the CPU is underutilized.
|
  Elapsed Time:                         113.02 s
 Memory Footprint:
 Resident:        60.00 MB
 Virtual:        216.00 MB

Graphical representation of this data is available in the HTML report: /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_report_20250416_153644.html
```

MinMax visible: false
Suggestion visible: true
Application: *mat_mul_a*
Report creation date: *2025-04-16 15:34:51*

113.02s

Elapsed Time

Your application might underutilize the available logical CPU cores

because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel® VTune™ Profiler to discover why the CPU is underutilized.

Current run Target Tuning Potential

Memory Footprint

Resident
60MB

Virtual
216MB

```
scteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ aps --result-dir=aps_result-b -c omp ./mat_mul_b
  | Summary information
  |---------------------------------------------------------------------
    Application                   : mat_mul_b
    Report creation date          : 2025-04-16 15:37:32
    Used statistics               : /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_result-b/scteam01a
  |
  | Your application might underutilize the available logical CPU cores
  | because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel(R) VTune(TM
  ) Profiler to discover why the CPU is underutilized.
  |
    Elapsed Time:                           20.12 s
   Memory Footprint:
   Resident:         76.00 MB
   Virtual:         232.00 MB

  Graphical representation of this data is available in the HTML report: /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_report_20250416_153752.html
```

MinMax visible: false
Suggestion visible: true
Application: *mat_mul_b*
Report creation date: *2025-04-16 15:37:32*

20.12s

Elapsed Time

Your application might underutilize the available logical CPU cores

because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel® VTune™ Profiler to discover why the CPU is underutilized.

Current run Target Tuning Potential

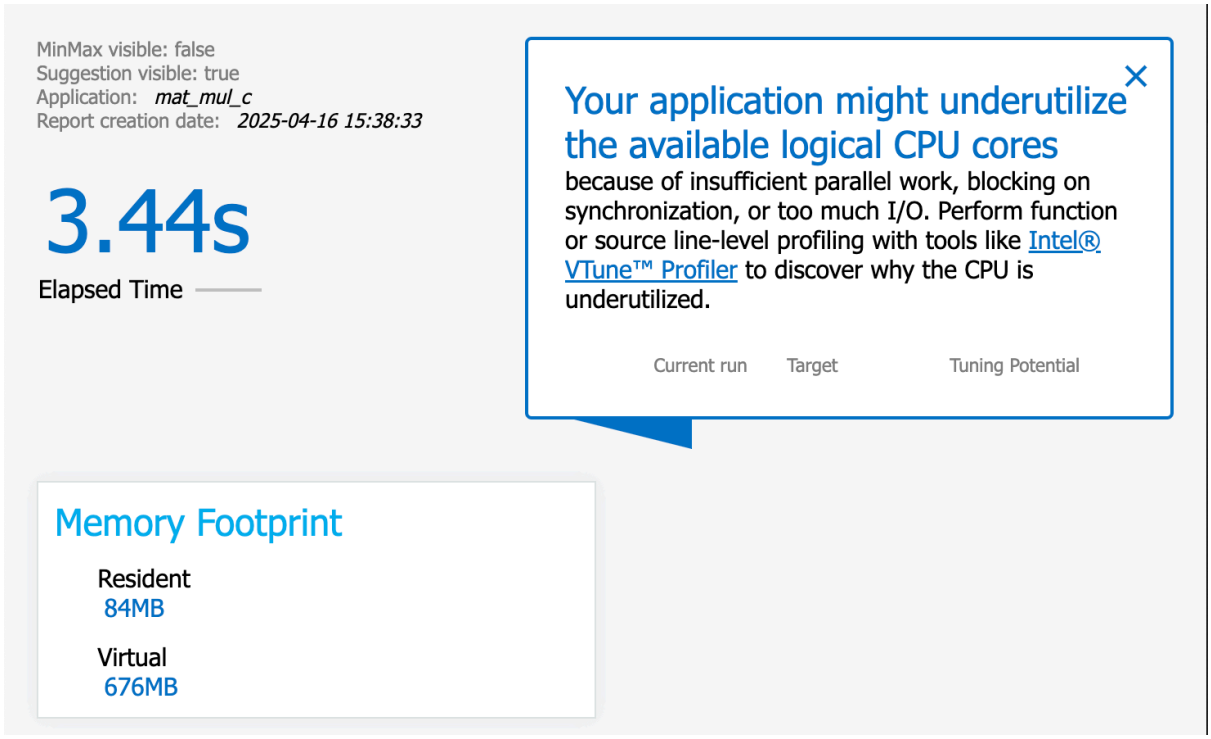
Memory Footprint

Resident
76MB

Virtual
232MB

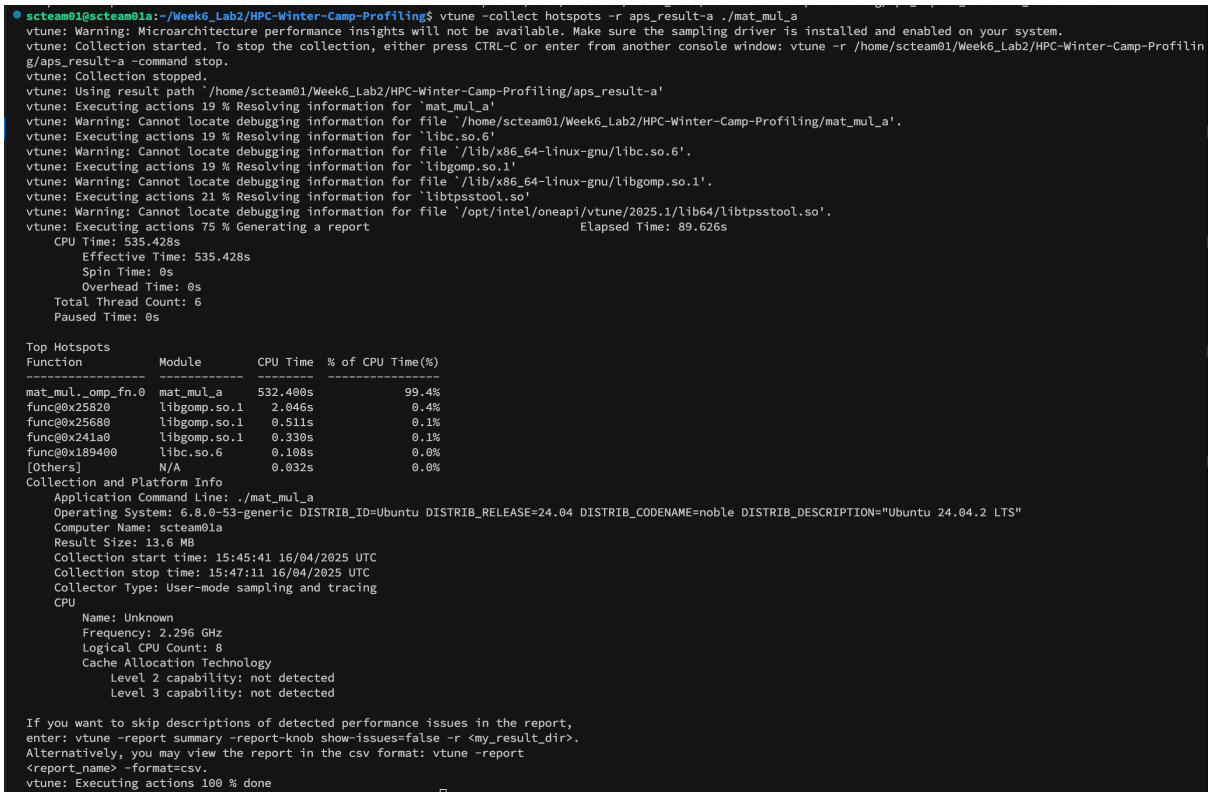
```
scteam01@scteam01a:~/Week6_Lab2/HPC-Winter-Camp-Profiling$ aps --result-dir=aps_result-c -c omp ./mat_mul_c
  | Summary information
  |---------------------------------------------------------------------
    Application                 : mat_mul_c
    Report creation date        : 2025-04-16 15:38:33
    Used statistics             : /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_result-c/scteam01a
  |
  | Your application might underutilize the available logical CPU cores
  | because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel(R) VTune(TM
  ) Profiler to discover why the CPU is underutilized.
  |
    Elapsed Time:                           3.44 s
   Memory Footprint:
   Resident:         84.00 MB
   Virtual:         676.00 MB

  Graphical representation of this data is available in the HTML report: /home/scteam01/Week6_Lab2/HPC-Winter-Camp-Profiling/aps_report_20250416_153837.html
```



Vtune

Run VTune Hotspots to identify the most time-consuming functions or sections.



MPI

Screenshot of Successful execution

I wrote a shell script to test cases with $n = (512, 1024, 2048)$. The shell script is as follows:

```
#!/bin/bash

# Clean and compile all necessary files
make clean
make

# Matrix sizes to test
Ns=(512 1024 2048)

# MPI configuration
NP=4       # number of processes
NX=2       # number of blocks in x-direction
NY=2       # number of blocks in y-direction

# Run for each matrix size
for N in "${Ns[@]}"
do
    echo "========== Running for N = $N =========="

    echo "[1] Generating A.out and B.out"
    ./generator "$N"

    echo "[2] Running MPI matrix multiplication"
    START=$(date +%s.%N)
```

```
        mpirun -np $NP ./matrix-mpi "$N" "$NX" "$NY"
        END=$(date +%s.%N)
        RUNTIME=$(echo "$END - $START" | bc)
        echo "    Total execution time: $RUNTIME seconds"

        # Optionally, calculate FLOPS based on theoretical operations (2 * N^3)
        FLOPS=$(echo "scale=2; 2 * $N * $N * $N / $RUNTIME" | bc)
        GFLOPS=$(echo "scale=2; $FLOPS / 1e9" | bc)
        echo "    Calculated performance: $FLOPS FLOPS"
        echo "    Calculated performance: $GFLOPS GFLOPS"

        echo "[3] Comparing results using float-diff"
        if [ -f C.out ]; then
            ./float-diff C.out output.out
            STATUS=$?
            if [ $STATUS -eq 0 ]; then
                echo "    Output is correct (no significant difference)"
            else
                echo "    Output mismatch detected"
            fi
        else
            echo "    C.out not found, skipping comparison"
        fi

        echo ""
    done
```

And the result is as follows:

```
scteam01@scteam01a:~/Week6_Lab2/MPI$ ./run.sh
rm -f generator matrix matrix-mpi float-diff
g++ -Wall -O2 generator.cpp -o generator
mpicxx -Wall -O2 matrix-mpi.cpp -o matrix-mpi
mpicxx -Wall -O2 float-diff.cpp -o float-diff
========== Running for N = 512 ==========
[1] Generating A.out and B.out
Wrote 262144 elements to C.out
Generated matrices A.in, B.in, and C.out with size 512 x 512
[2] Running MPI matrix multiplication
    Total execution time: .537902443 seconds
    Calculated performance: 499041154.19 FLOPS
    Calculated performance: .49 GFLOPS
[3] Comparing results using float-diff
Files are identical within tolerance 1e-06. Max difference: 0
    ✅ Output is correct (no significant difference)

========== Running for N = 1024 ==========
[1] Generating A.out and B.out
Wrote 1048576 elements to C.out
Generated matrices A.in, B.in, and C.out with size 1024 x 1024
[2] Running MPI matrix multiplication
    Total execution time: 1.462748615 seconds
    Calculated performance: 1468115317.95 FLOPS
    Calculated performance: 1.46 GFLOPS
[3] Comparing results using float-diff
Files are identical within tolerance 1e-06. Max difference: 0
    ✅ Output is correct (no significant difference)

========== Running for N = 2048 ==========
[1] Generating A.out and B.out
Wrote 4194304 elements to C.out
Generated matrices A.in, B.in, and C.out with size 2048 x 2048
[2] Running MPI matrix multiplication
    Total execution time: 20.460039692 seconds
    Calculated performance: 839679171.82 FLOPS
    Calculated performance: .83 GFLOPS
[3] Comparing results using float-diff
Files are identical within tolerance 1e-06. Max difference: 0
    ✅ Output is correct (no significant difference)
```

Understanding / Modifying / Optimizing the code

generator.cpp

This file generates the input matrices A and B with random values for testing.
Key functionality:

- Creates two matrices of size $N \times N$ with random float values
- Writes these matrices to A.out and B.out files
- Also computes the sequential matrix multiplication result and saves to C.out for verification

matrix-mpi.cpp

This file implements the parallel matrix multiplication using MPI.
Key components:

- MPI initialization and process configuration (rank, size)
- Data distribution: divides matrices into blocks according to NX and NY parameters
- Communication pattern between processes for exchanging matrix blocks
- Local computation of partial results
- Gathering final results to the root process

Optimization Strategies:

- **Block-Cyclic Distribution**: Instead of simple block distribution, implementing a block-cyclic distribution could improve load balancing, especially for larger matrices.
- **Overlapping Communication and Computation**: Using non-blocking MPI operations (e.g., MPI_Isend, MPI_Irecv) to overlap communication with computation.
- **Hierarchical Parallelism**: Combining MPI with OpenMP for hybrid parallelism - using MPI across nodes and OpenMP threads within each node.
- **Algorithmic Improvements**: Implementing Cannon's algorithm or SUMMA (Scalable Universal Matrix Multiplication Algorithm) which are designed to minimize communication overhead.
- **Memory Optimization**: Using local memory efficiently through cache blocking techniques to improve spatial and temporal locality.
- **Communication Reduction**: Reorganizing the algorithm to minimize the total volume of data exchanged between processes.

float-diff.cpp

This utility compares the output matrices to verify correctness.
Key functionality:

- Reads two matrix files and compares their values
- Allows for small floating-point differences (epsilon tolerance)
- Returns success (0) if matrices match within tolerance

```
Performance counter stats for './matrix-mpi 512 2 2':

          157.62 msec task-clock                       #    0.322 CPUs utilized
             273      context-switches                 #    1.732 K/sec
               4      cpu-migrations                   #   25.377 /sec
           7,211      page-faults                      #   45.749 K/sec
     360,375,672      cycles                           #    2.286 GHz
     133,195,375      stalled-cycles-frontend          #   36.96% frontend cycles idle
     354,654,416      instructions                     #    0.98  insn per cycle
                                                  #    0.38  stalled cycles per insn
      57,642,418      branches                         #  365.702 M/sec
       2,247,102      branch-misses                    #    3.90% of all branches

     0.489452940 seconds time elapsed

     0.067868000 seconds user
     0.094485000 seconds sys
```

Profile results

The result is as follows:
The profiling results provide valuable insights into the performance characteristics of our MPI implementation. The key observations are:

- **CPU Utilization**: At 0.322 CPUs utilized, our implementation shows relatively low CPU efficiency, suggesting potential for better parallelism or resource usage.
- **Cache and Memory Behavior**: The high number of page faults (7,211) indicates frequent memory access outside the working set, which could be a performance bottleneck.
- **Pipeline Efficiency**: With 36.96% frontend cycles idle and a instructions per cycle (IPC) ratio of 0.98, the processor is spending significant time waiting, likely due to memory access latency.
- **Branch Prediction**: The 3.90% branch misprediction rate is reasonable but could be improved for better instruction flow.
- **Overall Performance**: The total execution time of 0.489 seconds for a 512×512 matrix multiplication demonstrates the effectiveness of parallelization, but the metrics suggest room for optimization, particularly in memory access patterns and CPU utilization.

These metrics suggest that optimizing memory locality and reducing communication overhead would likely yield significant performance improvements for larger matrix sizes.

Computational Performance (GFLOPS)

Theoretical FLOPS Calculation

For matrix multiplication of two $N \times N$ matrices, the operation count can be derived as follows:

- For each element in the result matrix, we compute the dot product of a row from matrix A and a column from matrix B
- Each dot product requires N multiplications and $N - 1$ additions
- Therefore, each element requires approximately $2N$ floating-point operations (for large N)
- With N^2 elements in the result matrix, the total operation count is approximately $2N^3$

This is why we use $2N^3$ floating-point operations in our GFLOPS calculation:

$$\begin{aligned}\text{GFLOPS} &= \frac{\text{Total Operations}}{\text{Execution Time (seconds)}} \\ &= \frac{2N^3 \text{ operations}}{\text{Execution Time (seconds)}} \\ &= \frac{2N^3}{10^9} \cdot \text{GFLOPS}\end{aligned}$$

Matrix Size (N)	**Execution Time (s)**	**GFLOPS**
512	0.5379	0.49
1024	1.4627	1.46
2048	20.4600	0.83

Table 1: Computational performance for different matrix sizes