Parallel and Distributed Programming

TP1 – Optimizing Memory Access

Malak Kably

January 2026

1 Introduction

This report presents the work carried out for the first laboratory assignment (TP1) of the *Parallel and Distributed Programming* course. Below, the responses and experimental results for each exercise are presented and analyzed.

2 Exercise 1: Impact of Memory Access Stride

2.1 Objective

The objective of this exercise is to study the impact of memory access stride on program performance. The execution time and memory bandwidth are measured for different stride values in order to observe the effect of data locality and cache behavior.

2.2 Experimental Setup

The given C program was implemented in an Ubuntu virtual machine in order to evaluate the impact of memory access stride on performance. The source file `stride.c` allocates an array of doubles, initializes all elements to 1.0, and performs a summation while accessing the array with increasing stride values.

The program was compiled using `gcc` with two different optimization levels:

```
gcc -O0 -o stride stride.c
gcc -O2 -o stride stride.c
```

The executable was then run for stride values ranging from 1 to 20. For each execution, the program reports the execution time (in milliseconds) and the achieved memory bandwidth (in MB/s). The outputs were saved into two separate files, `results_O0.txt` and `results_O2.txt`, corresponding to the two compilation modes.

The collected data were processed and visualized using a Jupyter Notebook. The notebook and the raw result files are provided in the `/ex1` directory, located in the same folder as this report.

2.3 Results

Figure 1 shows the comparison between the `-O0` and `-O2` versions in terms of execution time and memory bandwidth for different stride values.

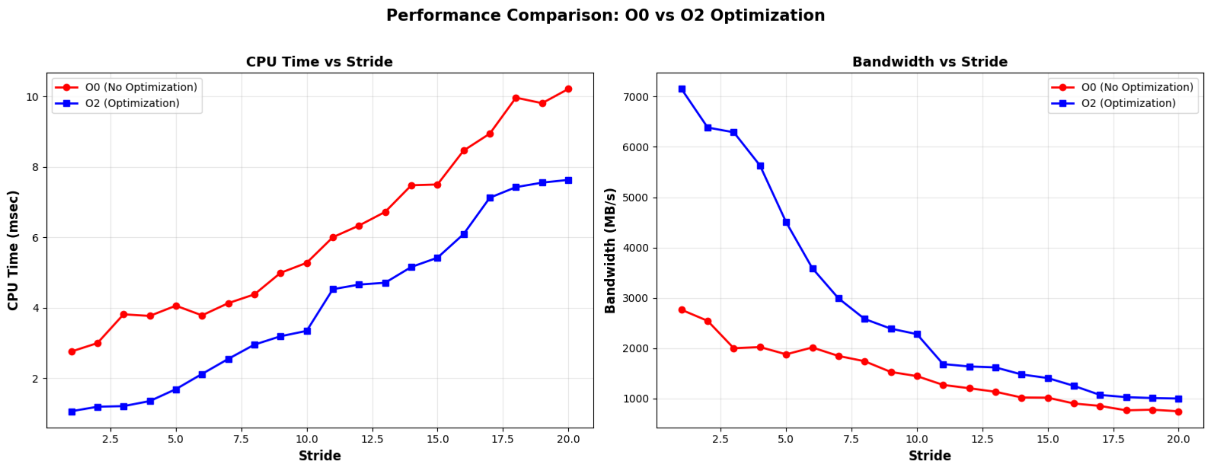


Figure 1: CPU time and memory bandwidth as a function of stride for `-O0` and `-O2` compilations

2.4 Analysis

The results show that when the stride increases, the execution time increases while the memory bandwidth decreases. For small stride values, especially stride equal to one, memory accesses are contiguous, which allows efficient use of cache lines and good spatial locality.

When the stride becomes larger, memory accesses are more spread out. As a result, fewer useful elements are loaded per cache line and the number of cache misses increases. This leads to more accesses to main memory, which explains the observed performance degradation.

The version compiled with `-O2` consistently performs better than the `-O0` version. This improvement comes from compiler optimizations such as loop unrolling and vectorization, which reduce overhead and better exploit the memory hierarchy. However, even with optimizations enabled, large stride values still have a negative impact, confirming the importance of data locality for performance.

3 Exercise 2: Optimizing Matrix Multiplication

3.1 Objective

The objective of this exercise is to study the impact of loop ordering on the performance of matrix multiplication. By changing the order of the nested loops, the goal is to improve cache usage and reduce execution time.

3.2 Methodology

Two versions of a matrix multiplication program were implemented in C. The first version uses the standard `ijk` loop order, while the second version uses the `ikj` loop order to improve data locality. The corresponding source files are provided in the `ex2` directory as `mxm_ijk.c` and `mxm_ikj.c`.

Both programs were compiled and executed using two different optimization levels:

```
gcc -O0 mxm_ijk.c -o mxm_ijk
gcc -O0 mxm_ikj.c -o mxm_ikj
gcc -O2 mxm_ijk.c -o mxm_ijk_O2
gcc -O2 mxm_ikj.c -o mxm_ikj_O2
```

The execution time of each version was measured and redirected to text files for later analysis. The result files are stored in the `ex2` folder as: `result_ijk_O0.txt`, `result_ikj_O0.txt`, `result_ijk_O2.txt`, and `result_ikj_O2.txt`.

3.2.1 Standard Matrix Multiplication (ijk)

The first implementation follows the classical triple-nested loop order `ijk`, where the output matrix element $C[i][j]$ is computed by iterating over the index k. This version directly reflects the mathematical definition of matrix multiplication but does not optimize memory access patterns.

```c
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

3.2.2 Optimized Loop Order (ikj)

In the optimized version, the loop order is modified to `ikj`. This change does not affect the correctness of the computation but significantly improves data locality by accessing matrix `B` in a row-wise manner.

```c
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

3.3 Results

The measured execution times clearly show the impact of both loop ordering and compiler optimizations. Without compiler optimizations, the standard `ijk` implementation required **42.32 seconds**, while the reordered `ikj` version reduced the execution time to **11.88 seconds**.

When compiler optimizations were enabled using `-O2`, execution times decreased significantly for both versions. The `ijk` version achieved an execution time of **12.86 seconds**, whereas the `ikj` version reached the best performance with a time of only **3.62 seconds**.

Overall, the `ikj` loop order consistently outperforms the standard `ijk` version, and the combination of loop reordering and compiler optimizations provides the lowest execution time.

3.4 Analysis

The results confirm that loop ordering has a major impact on the performance of matrix multiplication. In the standard `ijk` implementation, the matrix `B` is accessed column-wise, which leads to poor spatial locality and frequent cache misses. This explains the high execution time of **42.32 seconds** observed without compiler optimizations.

By switching to the `ikj` loop order, accesses to matrix `B` become row-wise, matching the memory layout in memory. This significantly improves cache line utilization and reduces memory access overhead, resulting in a reduced execution time of **11.88 seconds** without optimizations.

Compiler optimizations further improve performance in both cases. With `-O2`, the execution time of the `ijk` version drops to **12.86 seconds**, while the `ikj` version achieves the best result at **3.62 seconds**. These results show that while compiler optimizations are important, choosing a cache-friendly loop order is a key factor for achieving high performance.

4 Exercise 3: Matrix Multiplication per Block

4.1 Objective

The objective of this exercise is to improve the performance of matrix multiplication by applying the blocking (tiling) technique. By processing matrices in submatrices of size $B \times B$, the goal is to increase cache reuse, reduce memory traffic, and determine the optimal block size that provides the best performance.

4.2 Methodology

A blocked version of matrix multiplication was implemented in C. Instead of computing individual matrix elements, the algorithm processes submatrices (blocks) and fully computes each block before moving to the next one. This approach improves data locality and cache utilization.

The initial implementation was written to test the correctness of the blocked algorithm. The code was then modified to allow the block size to be changed at compile time using a preprocessor macro. This made it possible to evaluate the impact of different block sizes without modifying the source code.

The source file used for this exercise is `mxm_bloc.c`, located in the `ex3` directory.

4.2.1 Blocked Matrix Multiplication

The following code snippet illustrates the blocked matrix multiplication algorithm used in this exercise:

```c
for (int ii = 0; ii < N; ii += BS)
    for (int kk = 0; kk < N; kk += BS)
        for (int jj = 0; jj < N; jj += BS)
            for (int i = ii; i < min(ii + BS, N); i++)
                for (int k = kk; k < min(kk + BS, N); k++)
                    for (int j = jj; j < min(jj + BS, N); j++)
                        C[i][j] += A[i][k] * B[k][j];
```

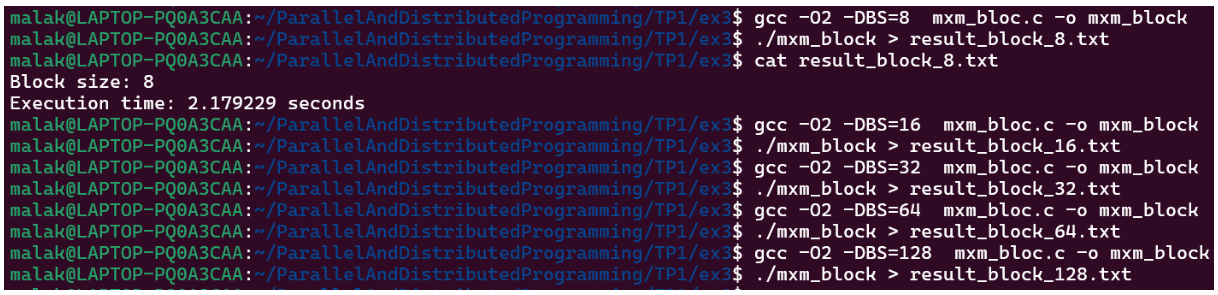
This structure ensures that matrix elements are accessed in blocks of size $B \times B$, allowing data loaded into the cache to be reused multiple times before eviction.

4.3 Experimental Setup

The program was compiled using the `-O2` optimization level, as required by the exercise. Different block sizes were tested by defining the block size at compile time using the `-DBS` flag:

```
gcc -O2 -DBS=8   mxm_bloc.c -o mxm_block
gcc -O2 -DBS=16  mxm_bloc.c -o mxm_block
gcc -O2 -DBS=32  mxm_bloc.c -o mxm_block
gcc -O2 -DBS=64  mxm_bloc.c -o mxm_block
gcc -O2 -DBS=128 mxm_bloc.c -o mxm_block
```

Each executable was run and its output redirected to a text file for analysis. All result files are stored in the `ex3` directory.



```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ gcc -O2 -DBS=8  mxm_bloc.c -o mxm_block
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ ./mxm_block > result_block_8.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ cat result_block_8.txt
Block size: 8
Execution time: 2.179229 seconds
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ gcc -O2 -DBS=16  mxm_bloc.c -o mxm_block
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ ./mxm_block > result_block_16.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ gcc -O2 -DBS=32  mxm_bloc.c -o mxm_block
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ ./mxm_block > result_block_32.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ gcc -O2 -DBS=64  mxm_bloc.c -o mxm_block
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ ./mxm_block > result_block_64.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ gcc -O2 -DBS=128  mxm_bloc.c -o mxm_block
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex3$ ./mxm_block > result_block_128.txt
```

Figure 2: Compilation and execution of the blocked matrix multiplication program for different block sizes

4.4 Results

The execution times obtained for different block sizes are summarized below:

Block size B	Execution time (s)
8	2.18
16	1.85
32	3.60
64	2.93
128	2.58

The lowest execution time was obtained with a block size of $B = 16$.

4.5 Analysis

The results show that blocking significantly affects performance. For very small block sizes, the overhead introduced by additional loop levels limits performance gains. For large block sizes, the working set no longer fits efficiently into the cache, leading to increased cache misses and higher execution times.

The block size $B = 16$ provides the best performance, as it represents a good balance between computation and memory access. At this block size, data reuse within the cache is maximized while avoiding excessive overhead or cache pollution.

These results confirm that choosing an appropriate block size is critical for achieving high performance in matrix multiplication and highlight the importance of cache-aware algorithm design.

5 Exercise 4: Memory Management and Debugging with Valgrind

5.1 Objective

The objective of this exercise is to analyze dynamic memory management in a C program and detect memory leaks using Valgrind. The program allocates, initializes, prints, and duplicates an array, while intentionally introducing memory leaks to be detected and fixed.

5.2 Methodology

The provided C program was implemented in the file `memory_debug.c`, located in the `ex4` directory. The program dynamically allocates an integer array, initializes its values, prints the content, and creates a duplicate of the array using a second dynamic allocation.

Initially, the program does not free all allocated memory, which results in memory leaks. The program was compiled with debugging symbols to enable detailed memory analysis using Valgrind.

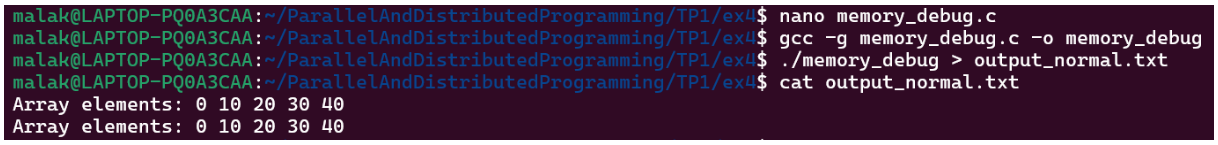
5.3 Compilation and Execution

The program was compiled using the following command:

```
gcc -g memory_debug.c -o memory_debug
```

The executable was first run normally to verify correct functionality, and the output was redirected to a file:

```
./memory_debug > output_normal.txt
```



```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex4$ nano memory_debug.c
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex4$ gcc -g memory_debug.c -o memory_debug
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex4$ ./memory_debug > output_normal.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP1/ex4$ cat output_normal.txt
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
```

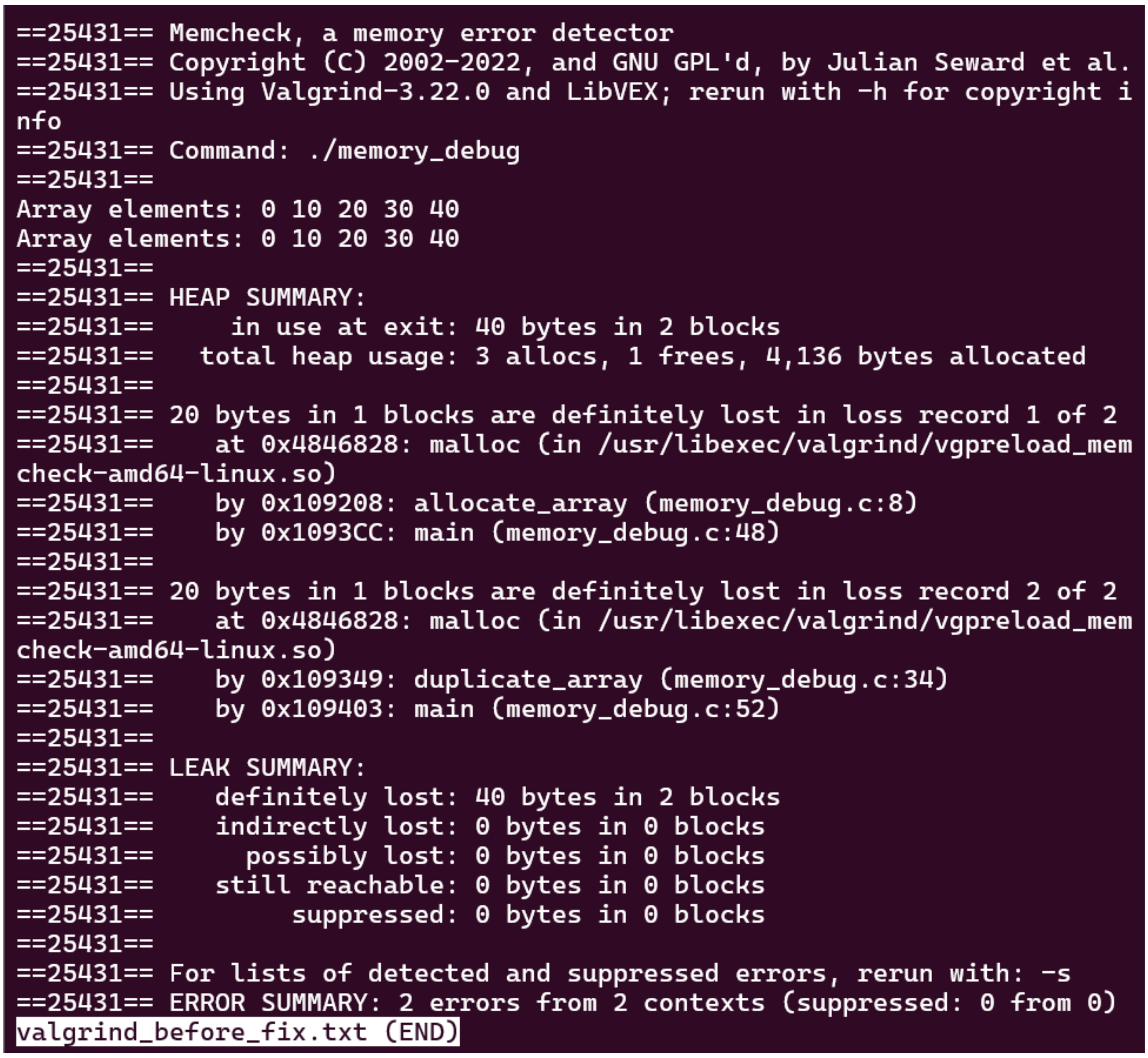
Figure 3: Compilation and normal execution of the program before memory leak analysis

5.4 Memory Leak Detection with Valgrind

Valgrind's Memcheck tool was then used to analyze the program and detect memory leaks. The output of Valgrind was redirected to a text file for analysis:

```
valgrind --leak-check=full --track-origins=yes ./memory_debug > valgrind_before_fix.txt 2>&1
```

The Valgrind report indicates that two memory blocks were *definitely lost*, corresponding to the dynamically allocated array and its duplicate. This confirms the presence of memory leaks caused by missing calls to `free()`.



```
==25431== Memcheck, a memory error detector
==25431== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==25431== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright i
nfo
==25431== Command: ./memory_debug
==25431==
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==25431==
==25431== HEAP SUMMARY:
==25431==     in use at exit: 40 bytes in 2 blocks
==25431==   total heap usage: 3 allocs, 1 frees, 4,136 bytes allocated
==25431==
==25431== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==25431==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_mem
check-amd64-linux.so)
==25431==    by 0x109208: allocate_array (memory_debug.c:8)
==25431==    by 0x1093CC: main (memory_debug.c:48)
==25431==
==25431== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==25431==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_mem
check-amd64-linux.so)
==25431==    by 0x109349: duplicate_array (memory_debug.c:34)
==25431==    by 0x109403: main (memory_debug.c:52)
==25431==
==25431== LEAK SUMMARY:
==25431==    definitely lost: 40 bytes in 2 blocks
==25431==    indirectly lost: 0 bytes in 0 blocks
==25431==      possibly lost: 0 bytes in 0 blocks
==25431==    still reachable: 0 bytes in 0 blocks
==25431==         suppressed: 0 bytes in 0 blocks
==25431==
==25431== For lists of detected and suppressed errors, rerun with: -s
==25431== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
valgrind_before_fix.txt (END)
```

Figure 4: Valgrind output before fixing the memory leaks, showing definitely lost memory blocks

As shown in Figure 4, Valgrind reports two memory blocks that are definitely lost, confirming the presence of memory leaks.

5.5 Fixing the Memory Leaks

To fix the detected memory leaks, the `free_memory` function was updated to properly deallocate memory. Additionally, both dynamically allocated arrays were freed in the `main` function.

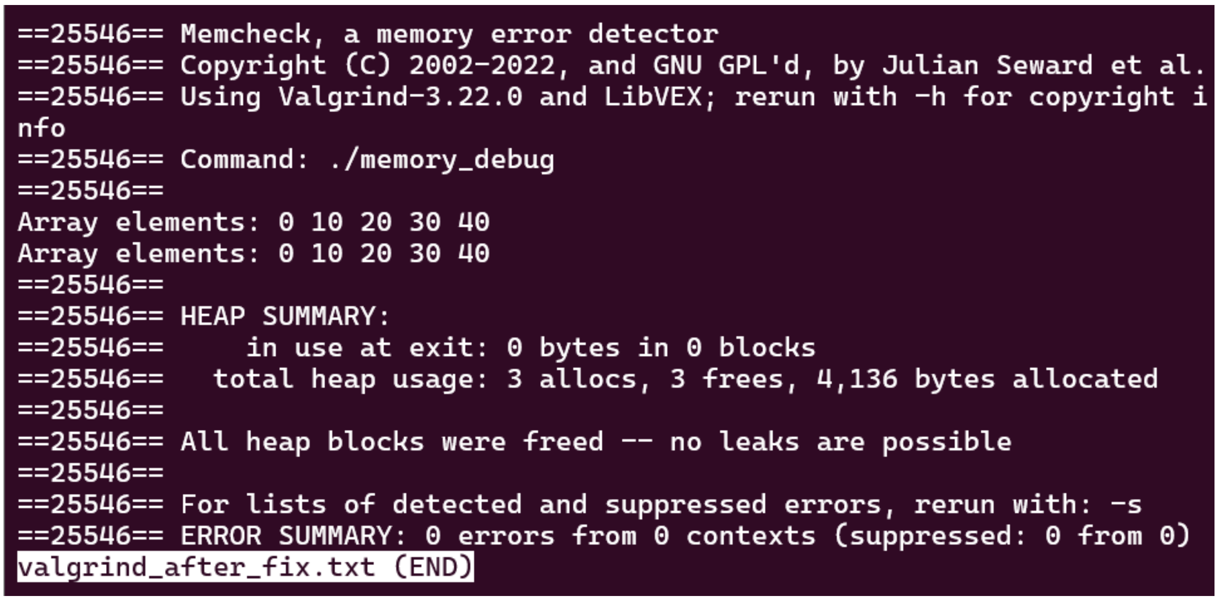
After applying these changes, the program was recompiled and analyzed again using Valgrind:

```
gcc -g memory_debug.c -o memory_debug
valgrind --leak-check=full --track-origins=yes ./memory_debug > valgrind_after_fix.txt 2>&1
```

5.6 Verification and Analysis

The second Valgrind report shows that no memory blocks are lost, with all allocated memory properly freed before program termination. This confirms that the memory leaks were successfully fixed.

All source files, executables, and Valgrind outputs related to this exercise are available in the `ex4` directory.



```
==25546== Memcheck, a memory error detector
==25546== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==25546== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright i
nfo
==25546== Command: ./memory_debug
==25546==
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==25546==
==25546== HEAP SUMMARY:
==25546==     in use at exit: 0 bytes in 0 blocks
==25546==   total heap usage: 3 allocs, 3 frees, 4,136 bytes allocated
==25546==
==25546== All heap blocks were freed -- no leaks are possible
==25546==
==25546== For lists of detected and suppressed errors, rerun with: -s
==25546== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
valgrind_after_fix.txt (END)
```

Figure 5: Valgrind output after fixing the memory leaks, confirming that all allocated memory was freed

Figure 5 confirms that all heap blocks were freed and no memory leaks remain after applying the fix.

6 Exercise 5: HPL Benchmark

6.1 Objective

The objective of this exercise is to evaluate the performance of my machine using the HPL (High Performance Linpack) benchmark. I studied how the matrix size (`N`) and the block size (`NB`) influence execution time, achieved performance (GFLOP/s), and efficiency. All experiments were executed using a single MPI process, as required in the exercise.

6.2 Experimental Setup

I used HPL version 2.3 and ran all tests on my local machine. The configuration was set with a single process grid ($P = 1$, $Q = 1$), meaning that only one CPU core was used.

Several matrix sizes were tested ($N = 1000, 5000, 10000, 20000$). For each matrix size, I varied the block size (`NB`) and collected the execution time and performance. Only runs that passed the residual check were considered valid.

All configuration files, scripts, CSV files, and plots related to this exercise are stored in the `ex5` folder.

6.3 Data Collection and Processing

Running HPL manually for each combination of N and `NB` would be tedious. To automate the process, I wrote a shell script that runs HPL for different values, extracts the execution time and GFLOP/s from the output, and stores the results in a CSV file.

A simplified version of the script is shown below:

```
echo "N,NB,time_s,gflops,status" > hpl_results.csv

for N in 1000 5000 10000 20000; do
  for NB in 1 2 4 8 16 32 64 128 256; do
    ./xhpl > hpl_tmp.out
    TIME=$(grep WR00C2R4 hpl_tmp.out | awk '{print $6}')
    GFLOPS=$(grep WR00C2R4 hpl_tmp.out | awk '{print $7}')
    echo "$N,$NB,$TIME,$GFLOPS,PASSED" >> hpl_results.csv
  done
done
```

After collecting the data, I used a Python script to read the CSV file and generate plots. The script computes performance, execution time, and efficiency as a function of the block size.

6.4 Results

Figure 6 shows the achieved performance (GFLOP/s) as a function of the block size. For all matrix sizes, performance increases with the block size until reaching an optimal range. Beyond this point, performance stabilizes or slightly decreases.

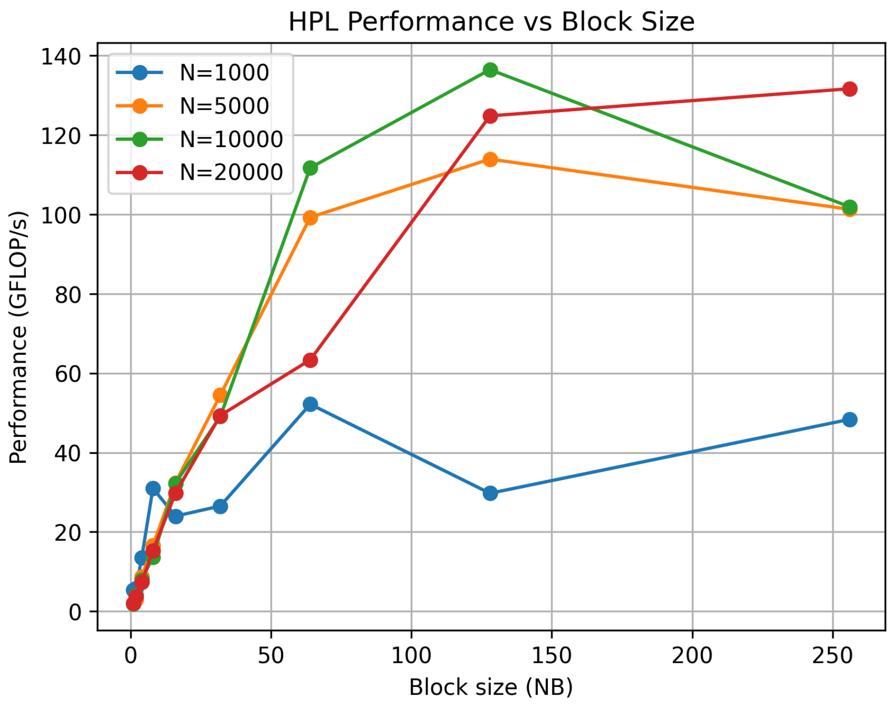


Figure 6: HPL performance (GFLOP/s) versus block size for different matrix sizes

Figure 7 presents the execution time. As expected, larger matrices require more time. However, increasing the block size significantly reduces execution time until an optimal value is reached.

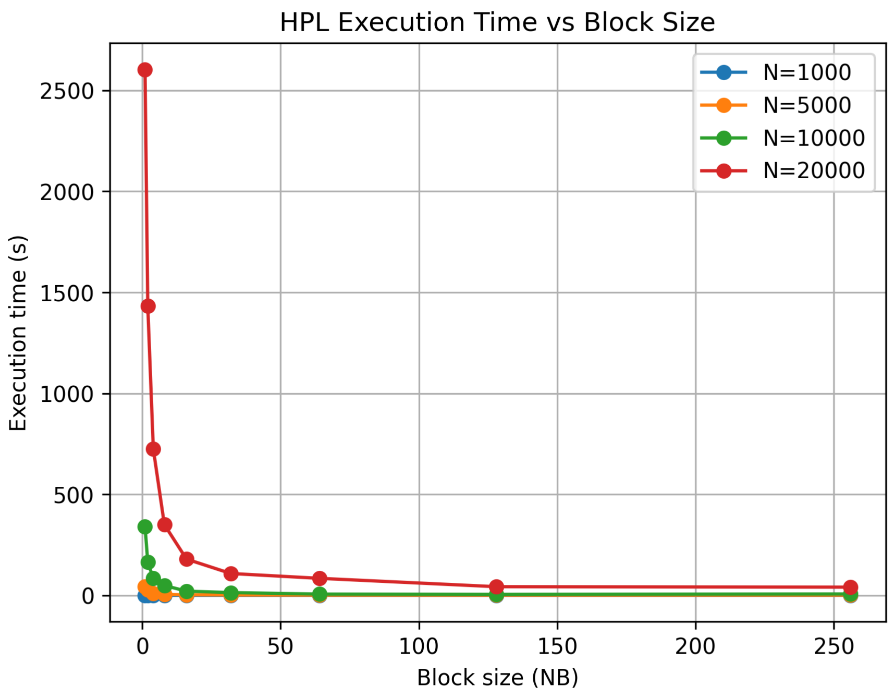


Figure 7: HPL execution time versus block size for different matrix sizes

6.5 Efficiency Analysis

Efficiency was computed as the ratio between the measured performance and the theoretical peak performance of the processor. Figure 8 shows that efficiency improves for larger matrix sizes and appropriate block sizes.

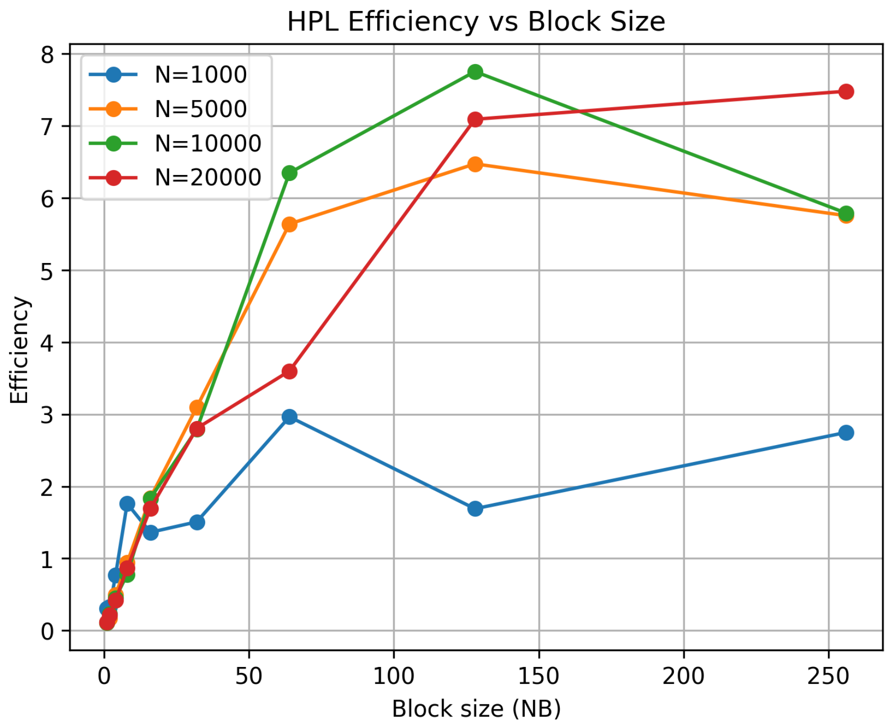


Figure 8: HPL efficiency versus block size for different matrix sizes

6.6 Discussion

From these results, I observed that the block size has a strong impact on HPL performance. Very small block sizes lead to poor cache usage, while very large block sizes can reduce flexibility.

An intermediate block size provides the best balance between computation and memory access. Larger matrix sizes achieve higher performance and efficiency because they better exploit the processor and amortize memory overhead. Overall, the results follow the expected behavior of HPL and confirm the importance of tuning the block size.

7 Conclusion

This laboratory assignment highlighted the importance of memory access patterns and data locality in program performance. Through the different exercises, we observed how loop reordering, blocking, and compiler optimizations can significantly reduce execution time by improving cache utilization.

Although some behaviors were not always straightforward to understand at first, but going back to the course slides and searching for additional explanations helped clarify the observed results. This process made it easier to connect the experimental outcomes with the theoretical concepts discussed in class.

Overall, this TP provided practical insight into performance optimization techniques and reinforced key ideas related to memory hierarchy, debugging, and efficient program design in high-performance computing.