# Exercise Set 5

Darina Öö

19.02.2024

## 1 Problem 1

### 1.1 Methodology

Two separate C++ programs were written and executed on an Ubuntu system (refer to file_write_comparison.cpp). The first program writes the array to a file in an unformatted manner using `fwrite`, while the second program uses `fprintf` with the format string `%g` for formatted output.

### 1.2 Implementation

The core implementation involved iterating over the array and writing each element to a file. CPU time was measured using the `clock()` function before and after the write operation to calculate the time used.

```
#include <iostream>
#include <cstdio>
#include <ctime>
#include <vector>

int main() {
    // Array initialization and file writing code here
    return 0;
}
```

### 1.3 Instructions for Implementation

1. **Save the Code:** Copy the provided code into a text editor and save it as `file_write_comparison.cpp`.

2. **Compile the Program:** Open a terminal and compile the code using a C++ compiler:

```
g++ -o file_write_comparison file_write_comparison.cpp
```

3. **Run the Program:** After compiling, run the program by typing:

```
./file_write_comparison
```

4. **Compare the Results:** The program outputs the CPU time taken for both the formatted (`%g` ASCII format) and unformatted binary write operations, facilitating a comparison of efficiency between these two methods of writing data to a file in C++.

### 1.4 Results

The execution results were as follows:

- Unformatted write CPU time: 0.19868 seconds
- Formatted write CPU time: 1.32748 seconds

### 1.5 Conclusion

The unformatted writing method proved to be significantly more efficient than the formatted method in terms of CPU time used. This outcome demonstrates the overhead associated with converting binary data to a human-readable format, which impacts performance when dealing with large datasets.

## 2 Problem 2

### 2.1 Methodology

The provided Fortran program 'omit_frame_pointer.f90' was compiled with 'gfortran' using six combinations of optimization and frame pointer settings. The executable performance was then measured to understand the impact of these compiler options.

### 2.2 Compilation Instructions

The program was compiled with the following commands in a bash script, creating six different executables to compare performance across compiler options:

```
gfortran -O0 -fno-omit-frame-pointer omit_frame_pointer.f90 -o prog_a
gfortran -O0 -fomit-frame-pointer omit_frame_pointer.f90 -o prog_b
...
gfortran -O2 -fomit-frame-pointer omit_frame_pointer.f90 -o prog_f
```

### 2.3 Execution and Results

The executables were run, and their output was recorded, showing the sum of an array and the CPU time used for the computation. The results were as follows:

| Executable | Sum of Array | CPU Time (seconds) |
|---|---|---|
| prog_a | 0.0025600000 | 0.6464280000 |
| prog_b | 0.0025600000 | 0.6559440000 |
| prog_c | 0.0025600000 | 0.1021570000 |
| prog_d | 0.0025600000 | 0.1055250000 |
| prog_e | 0.0025600000 | 0.0340610000 |
| prog_f | 0.0025600000 | 0.0271180000 |

### 2.4 Analysis

The data indicates a significant performance improvement as the optimization level increases from ‘-O0‘ to ‘-O2‘. The effect of omitting the frame pointer is minimal, suggesting that other optimizations at level ‘-O2‘ have a more substantial impact on execution time. Higher optimization levels (e.g., ‘-O2‘) significantly enhance performance for compute-intensive tasks. The ‘-fomit-frame-pointer‘ option shows a negligible effect on execution time, underscoring the importance of comprehensive optimization strategies over specific compiler flags for this type of application.

## 3 Problem 3

### 3.1 Methodology

The program was compiled with MPI and run across different processor counts. The output was observed to understand how MPI processes interact and the order in which messages are printed.

### 3.2 Instructions for Compilation and Execution

```
mpicc -o mpiexample mpiexample.c
mpiexec -n 1 ./mpiexample > output_1.txt
mpiexec -n 2 ./mpiexample > output_2.txt
mpiexec -n 4 ./mpiexample > output_4.txt
mpiexec -n 8 ./mpiexample > output_8.txt
```

### 3.3 Results

The execution outputs varied with the number of processors:

- With 1 processor, only the host name was printed.

- With 2, 4 and 8 processors, messages from other processes were received and printed in no guaranteed order.

### 3.4 Analysis

The order of messages is non-deterministic due to the asynchronous nature of MPI. The root process receives messages from non-root processes in an unpredictable order, demonstrating the parallel execution model's dynamic behavior. This exercise illustrates MPI's capability to manage parallel processes across a distributed system, showcasing how message passing facilitates communication between processes executing concurrently.

## 4 Problem 4

### 4.1 Methodology

The program mpi_ping_pong.c initiates with an integer value of 0. It then performs a sequence of MPI send and receive operations between two processes. Each time a process receives the message, it increments the integer by one before sending it back. After two exchanges, the receiving process prints the final integer value.

### 4.2 Execution Instructions

1. Compile the program using `mpicc`:

```
mpicc -o mpi_ping_pong mpi_ping_pong.c
```

2. Run the program with 2 MPI processes:

```
mpiexec -n 2 ./mpi_ping_pong
```

### 4.3 Results

Running the program with 2 processes outputs:

```
Final value received by process 1: 3
```

### 4.4 Analysis

The final value of 3 confirms the successful message exchanges and increments. This demonstrates MPI's capability for process synchronization and data exchange in parallel computing. This exercise showcases basic MPI communication mechanisms and the effectiveness of message passing in parallel computation for synchronizing processes and modifying shared data.