OS Assignment1

Himanshu Jindal

December 2023

Low-Level Design Report & Output Analysis

Program Overview

The program initiates by reading values from the input file, 'input.txt', retrieving the parameters N (the upper limit for number checking) and K (the number of processes to create). In case of failure to open the file or if N or K is less than or equal to zero, an error is returned.

The design of the program ensures an equitable distribution of workload among all processes. This is achieved by allocating each process its share of both larger and smaller numbers. For instance, when dividing the range up to 55 among 6 processes, the allocation is as follows:

Process Number	Allocated Numbers
1	1, 7, 13, 19, 25, 31, 37, 43, 49, 55
2	2, 8, 14, 20, 26, 32, 38, 44, 50
3	3, 9, 15, 21, 27, 33, 39, 45, 51
4	4, 10, 16, 22, 28, 34, 40, 46, 52
5	5, 11, 17, 23, 29, 35, 41, 47, 53
6	6, 12, 18, 24, 30, 36, 42, 48, 54

Table 1: Number Allocation for $N = 55$ and $K = 6$

Design Choices

1. **Shared Memory:**
   - Shared memory is used to store the results of Tetrahedral checks for each process efficiently.
   - Each child process has its shared memory space, accessible by the parent process for result aggregation.

Child Process	Shared Memory	Purpose	Size (in int)	Total Size (in int)
1	`ptr[0]`	Stores results for numbers $6n + 1$	10	10
2	`ptr[1]`	Stores results for numbers $6n + 2$	9	19
3	`ptr[2]`	Stores results for numbers $6n + 3$	9	28
4	`ptr[3]`	Stores results for numbers $6n + 4$	9	37
5	`ptr[4]`	Stores results for numbers $6n + 5$	9	46
6	`ptr[5]`	Stores results for numbers $6n + 6$	9	55

Table 2: Shared Memory Usage for $N = 55$ and $K = 6$

2. **File Input:**
    - Input data is read from `input.txt`, and output results are stored in individual and main output files.
    - File names are dynamically generated based on the process number to avoid conflicts in parallel execution.

3. **Parallel Execution:**
    - The program creates K child processes, each responsible for checking a subset of numbers.
    - Parallelization is achieved by dividing the range (1 to N) into K subsets, and each process checks its assigned subset.

Complications and Solutions

1. **Shared Memory Usage:**
    - **Complication:** A more comfortable solution was a shared memory that could be shared to all the child processes which could populate it based on the number being Tetrahedral or not. This was implemented and making Shared memory for each process gave rise to errors like the shared memories getting destroyed when we end all the processes.
    - **Solution:** Used process number to create unique shared memory names for each child, and a lot of debugging and movinhg shared memory declarations in and out of the loops and changing declaration styles and names.

2. **Memory Management:**
    - **Complication:** Ensuring proper allocation and deallocation of shared memory and avoiding memory leaks.
    - **Solution:** Explicitly closed shared memory and freed allocated memory at the end of the program.

3. **Calculation of Runtime and Graph creation:**
    - **Complication:** Calculation of Runtimes for the program for different values of K and N and making the Graphs.
    - **Solution:** Used python code to run the C program, calculate the runtimes and draw the graphs using matplotlib.

Analysis of Output

1. **Correctness:**
    - Checked numbers are accurate, and the program correctly identifies Tetrahedral numbers using the `isTetrahedral` function that bruteforces to check if the number

    $$T_p = \frac{p \cdot (p+1) \cdot (p+2)}{6} \quad \text{where } p \geq 1$$

    - Output files (`OutFileI.txt`) contain the expected results for each process. Each file contains the numbers it was allotted to check and the result of the Tetrahedral check on it in the following format.

      1: a tetrahedral number

      2: Not a tetrahedral number

      ⋮

      4: a tetrahedral number

⋮

90: a tetrahedral number

⋮

- Main Output file (`OutMain.txt`) that lists down all the processes and the Tetrahedral Numbers each process has identified in the following format.

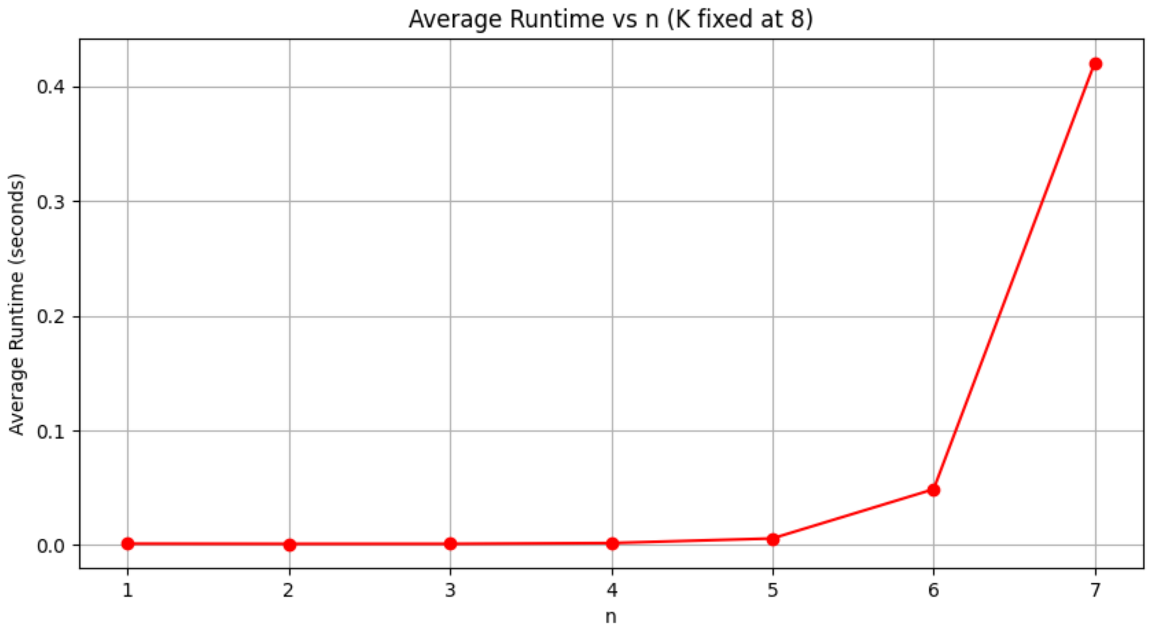
  P1: num1 num2 . . .

  P2: num5 num6 . . .

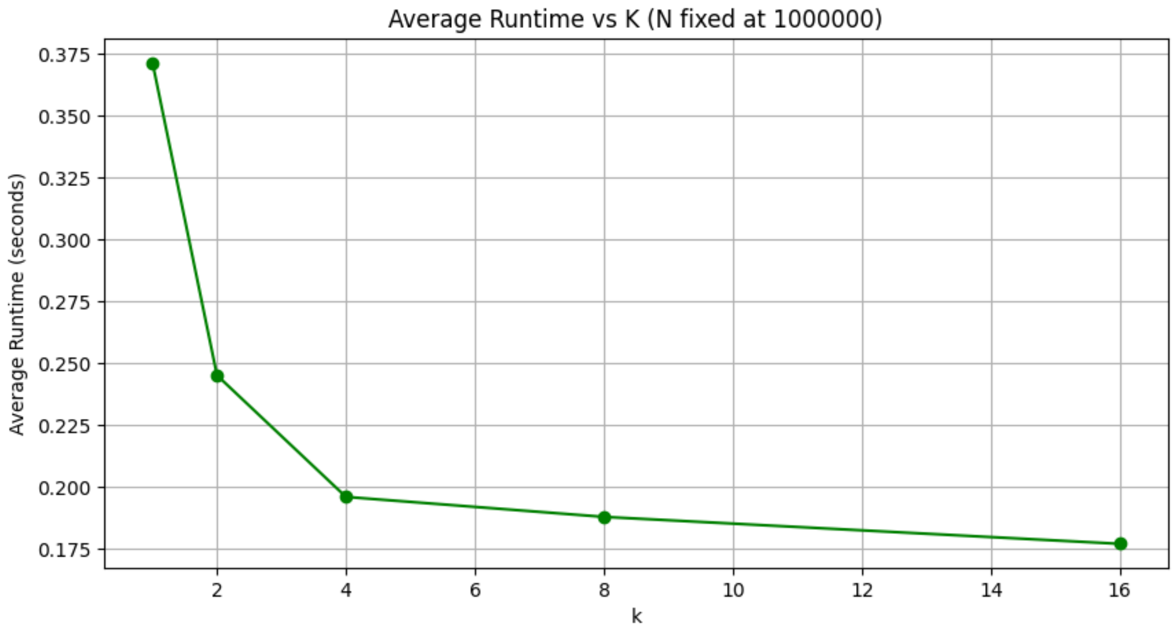
  ⋮

2. **Parallelization:**
   - The program successfully parallelizes the task, with each process handling a distinct subset of numbers.
   - Parallel execution speeds up the overall process, especially for large N and K values. Can be clearly visualized by Figure 1.
3. **Resource Management:**
   - Shared memory is properly managed, and memory is released after use.
   - File handles are closed to prevent resource leaks.

Graphs



(a) Time vs Size, N



(b) Time vs Number of Processes, K

Figure 1: Performance Metrics

Conclusion

Using multiple processes to work together made our Tetrahedral number finder much faster. The speedup we got was was significant as can be seen from the Figure 1 . Though there's a bit of extra work to manage all the processes, the benefits are clear.

As we checked more and more numbers ('N' getting bigger), a single process took too much time. The way we split the work among different processes showed that our solution can handle bigger tasks efficiently.

Creating separate processes with `fork()` introduced a challenge – each process had its own memory. To solve this, we used shared memory (`shm_open()`) to let the processes talk to each other. This made our program work better by sharing information about the Tetrahedral numbers they found.

In a nutshell, our program not only finds Tetrahedral numbers accurately but also does it much faster by working together. This way of designing things sets us up for handling big tasks well.