

## 2025.1 Multicore Computing, Project #1

20223961 김수아

### Problem #1

#### 1. Environment

##### A. CPU

- i. 13th Gen Intel Core i7-1355U
- ii. 10 cores(12 logical processors)
- iii. 1.70GHz base clock speed

B. Memory : 15.7GB

C. OS : Windows 11

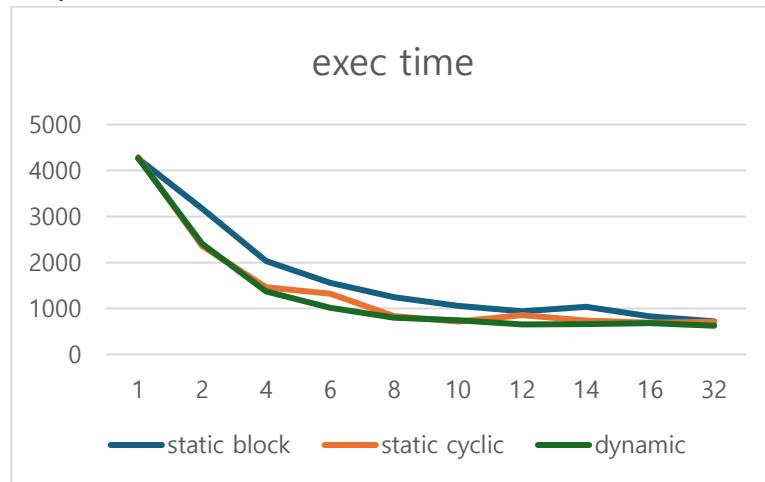
#### 2. Performance of programs

##### A. Execution Time (milliseconds)

##### i. Table

Exec time	1	2	4	6	8	10	12	14	16	32
Static(block)	4260	3172	2030	1557	1248	1060	940	1038	831	720
Static (cyclic)	4293	2360	1459	1320	838	717	857	738	692	711
Dynamic [task size : 10 numbers]	4268	2406	1373	1013	802	746	654	661	681	626

ii. Graph

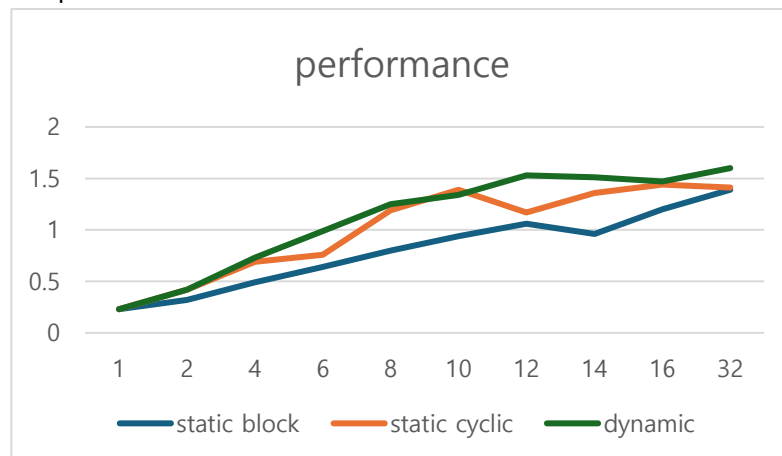


B. Performance (1/execution time)

i. Table

performance	1	2	4	6	8	10	12	14	16	32
Static(block)	0.23	0.32	0.49	0.64	0.80	0.94	1.06	0.96	1.20	1.39
Static (cyclic)	0.23	0.42	0.69	0.76	1.19	1.39	1.17	1.36	1.44	1.41
Dynamic [task size : 10 numbers]	0.23	0.42	0.73	0.99	1.25	1.34	1.53	1.51	1.47	1.60

ii. Graph



### C. Analysis

#### i. Performance changes as the number of threads increase

In general, the performance graph draws an upward-sloping curve. By executing multiple instructions simultaneously, overall performance is enhanced. However, the slope of the graph is not 1 but rather more gradual, and after a certain point, performance either decreases or plateaus. This can be explained by Amdahl's Law. As only part of the work can be parallelized, there is a limit to improve performance linearly even as the number of threads increases. Additionally, as the number of threads increases, the overhead for thread creation, management and synchronization also increases, which affects performance badly.

#### ii. Differences between three decomposition methods

The dynamic approach performed best. In the case of the block decomposition method, since the entire data is cut and allocated in order, there must have been a load imbalance between the threads that calculate small numbers that are relatively easy and the threads that calculate large numbers.

In addition, it is expected that the performance of the cyclic method is slightly improved than the static block method by allocating 10 alternating numbers. In fact, when the execution time of each thread is checked, the two methods took a long time for one or two threads, so there were cases where the execution time was delayed because the other threads waited.

On the other hand, the dynamic partitioning method was most effective as it dynamically allocates directly as soon as the task is finished in a thread.

## D. Screen capture image

### i. Execute result (end num = 200,000, thread num = 6)

#### 1. Block

```
package problem1;

public class pc_static_block { 실행
    private static int NUM_END = 200000; // default input 4개 사용 위치
    private static int NUM_THREADS = 6; // default number of threads 7개 사용 위치

    public static void main (String[] args) { 실행
        if (args.length==2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        totalCounter[] counters = new totalCounter[NUM_THREADS];
        Thread[] threads = new Thread(NUM_THREADS);

        "C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.4.1\lib\idea
        Thread 0 execution time: 214ms
        Thread 1 execution time: 572ms
        Thread 2 execution time: 819ms
        Thread 3 execution time: 1039ms
        Thread 4 execution time: 1338ms
        Thread 5 execution time: 1499ms
        Program Execution Time: 1499ms
        Performance: 0.66711140768587 operations per second
        1...199999 prime# counter=17984

        종료 코드 0(으)로 완료된 프로세스
```

#### 2. Cyclic

```
package problem1;

import java.util.concurrent.atomic.AtomicInteger;

public class pc_dynamic { 실행
    private static int NUM_END = 200000; // default input 4개 사용 위치
    private static int NUM_THREADS = 4; // default number of threads 5개 사용 위치
    private static AtomicInteger nextStartIndex = new AtomicInteger( initialValue: 0); // 2개 사용 위치

    public static void main (String[] args) { 실행
        if (args.length==2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        "C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.4.1\lib\idea
        Thread 0 execution time: 971ms
        Thread 1 execution time: 971ms
        Thread 2 execution time: 967ms
        Thread 3 execution time: 969ms
        Thread 4 execution time: 969ms
        Thread 5 execution time: 969ms
        Program Execution Time: 969ms
        Performance: 1.017293997965412 operations per second
        1...199999 prime# counter=17984

        종료 코드 0(으)로 완료된 프로세스
```

#### 3. Dynamic

```
package problem1;

public class pc_static_cyclic { 실행
    private static int NUM_END = 200000; // default input 4개 사용 위치
    private static int NUM_THREADS = 4; // default number of threads 6개 사용 위치

    public static void main (String[] args) { 실행
        if (args.length==2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        totalCounter[] counters = new totalCounter[NUM_THREADS];
        Thread[] threads = new Thread(NUM_THREADS);

        "C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.4.1\lib\idea
        Thread 0 execution time: 695ms
        Thread 1 execution time: 1395ms
        Thread 2 execution time: 743ms
        Thread 3 execution time: 731ms
        Thread 4 execution time: 1354ms
        Thread 5 execution time: 731ms
        Program Execution Time: 1395ms
        Performance: 0.7168458781362007 operations per second
        1...199999 prime# counter=17984

        종료 코드 0(으)로 완료된 프로세스
```

E. Explanation for compile and execute

- i. To run the programs from command line
  1. `cd out/production/[project name]`
  2. `java problem1.pc_static_block [number_of_threads] [end_number]`  
`java problem1.pc_static_cyclic [number_of_threads] [end_number]`  
`java problem1.pc_dynamic [number_of_threads] [end_number]`
- ii. Modify the constants NUM\_END and NUM\_THREADS at the top of the code to the desired values and then run the program

## **Problem #2**

1. Environment

- A. CPU
  - i. 13th Gen Intel Core i7-1355U
  - ii. 10 cores(12 logical processors)
  - iii. 1.70GHz base clock speed
- B. Memory : 15.7GB
- C. OS : Windows 11

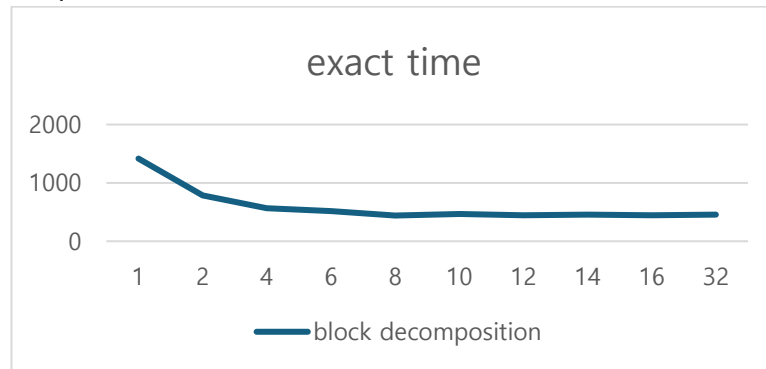
2. Performance of programs

- A. Block decomposition method
  - i. Execution Time (milliseconds)

1. Table

	1	2	4	6	8	10	12	14	16	32
Exec time	1417	786	567	515	441	467	444	454	446	455

## 2. Graph

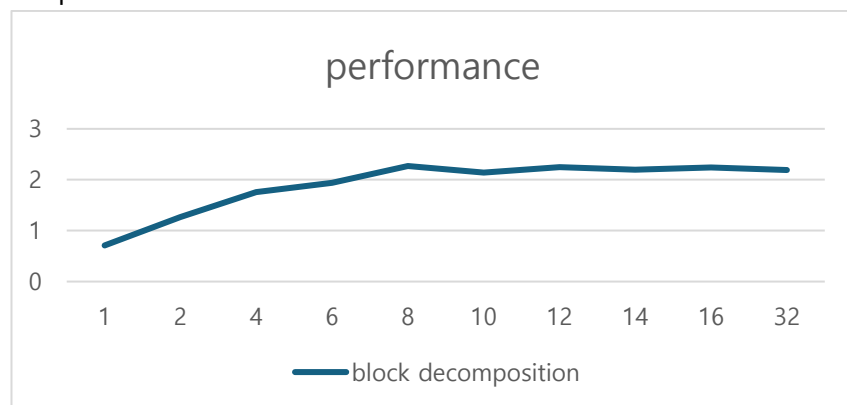


## ii. Performance (1/execution time)

### 1. Table

	1	2	4	6	8	10	12	14	16	32
Performance (1/exec time)	0.71	1.27	1.76	1.94	2.27	2.14	2.25	2.20	2.24	2.19

## 2. Graph



## iii. Analysis

### 1. Performance changes as the number of threads increases

The reason why the number of threads and the performance were not linear can be explained in the same way by Amdahl's law described in the analysis of problem1

### 2. Granularity

If the number of threads is small, it is the case of coarse-grain, and if the number of threads is large, it is the case of fine-grain parallelism.

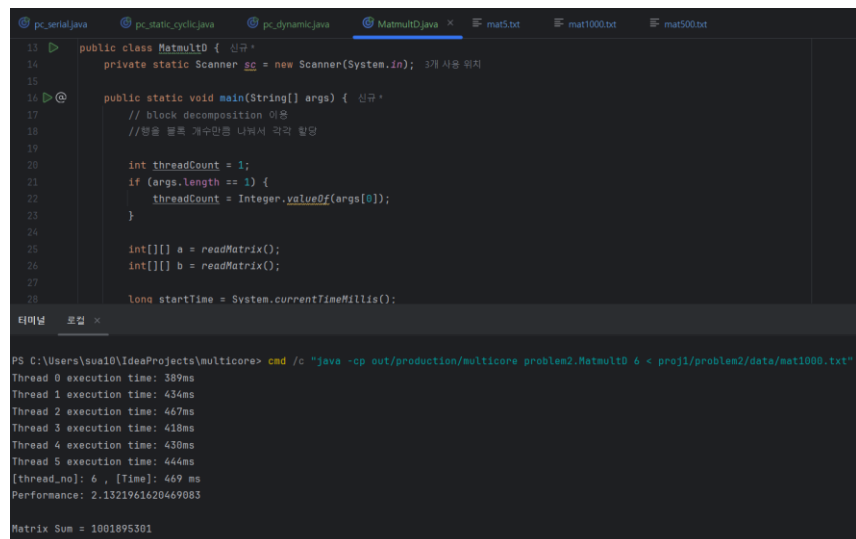
Since coarse-grain parallelism performs a large amount of computational work per thread, overhead is reduced and the efficiency of parallel processing increases. However, problems

can occur with load balancing. In the case of fine-grain parallelism, it can provide better load balancing but may suffer from performance degradation due to relatively high communication overhead compared to the actual computation.

The provided mat1000.txt contains evenly distributed values of 0, 1, 2. (= well load-balanced data) The experimental results show continuous performance improvement as thread number 1~8, but there is no significant change even if the thread is increased after that. This is expected to be due to overhead, which is a disadvantage of fine-grain parallelism, as the number of threads increases while the input value is already load balancing.

iv. Screen capture image

1. Thread num = 6



```
13 public class MatmultD {
14     private static Scanner sc = new Scanner(System.in);
15
16     public static void main(String[] args) {
17         // block decomposition 이용
18         // 행을 블록 개수만큼 나누어서 각각 할당
19
20         int threadCount = 1;
21         if (args.length == 1) {
22             threadCount = Integer.valueOf(args[0]);
23         }
24
25         int[][] a = readMatrix();
26         int[][] b = readMatrix();
27
28         long startTime = System.currentTimeMillis();
29
30         Thread 0 execution time: 389ms
31         Thread 1 execution time: 434ms
32         Thread 2 execution time: 467ms
33         Thread 3 execution time: 418ms
34         Thread 4 execution time: 430ms
35         Thread 5 execution time: 444ms
36         [thread_no]: 6 , [Time]: 449 ms
37         Performance: 2.1321961620469083
38         Matrix Sum = 1801895301
```

v. Explanation for compile and execute

1. cmd /c "java -cp out/production/multicore problem2.MatmultD  
[# of threads] proj1/problem2/data/mat1000.txt"