problem 2. Parallel matrix multiplication with static load balancing approach

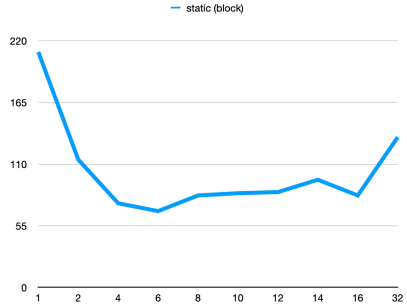
(a) experiment environment

- CPU type : Apple M1
- number of cores : 8 core CPU
- memory size : 16 GB
- OS : macOS Monterey

(b) execution time (ms) per the number of entire threads = {1,2,4,6,8,10,12,14,16,32}.

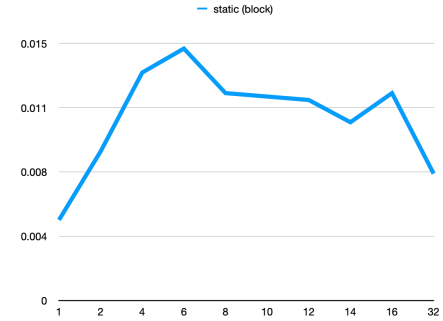
Execution time

Exec time	1	2	4	6	8	10	12	14	16	32
static (block)	210	114	75	68	82	84	85	96	82	134



performace (1/exec time)

performace. (1/exec time)	1	2	4	6	8	10	12	14	16	32
static (block)	0.0047	0.0087	0.0133	0.0147	0.0121	0.0119	0.0117	0.0104	0.0121	0.0074



Because this experiment was done at 8-core cpu environment, performance improves until 6-8 threads. Interesting point is that more than 8 threads, the execution time rather increases. In my opinion, excessive number of threads doesn't always guarantee better performance.

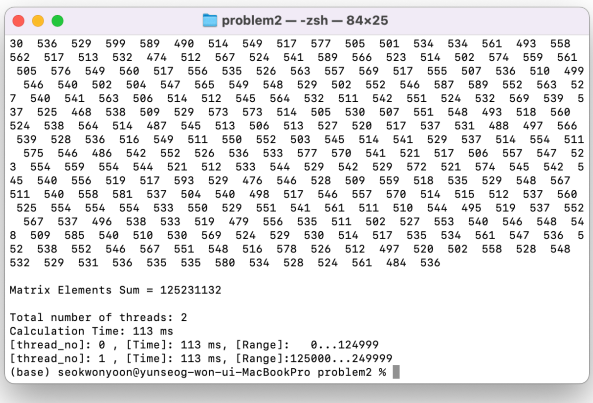
At 8-core cpu environment, It seems that for example 32 threads (more than 8) make worse queueing delay time.

(c) Analysis

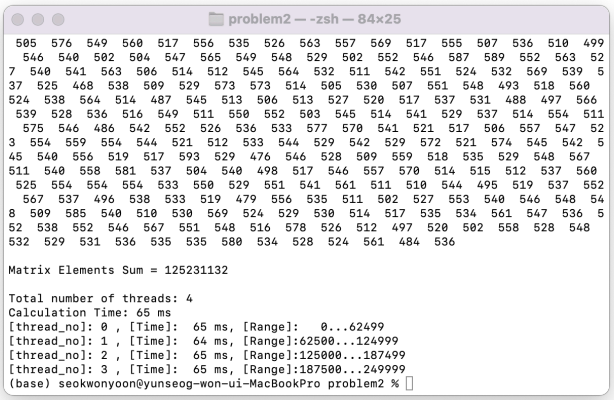
```java
import java.util.*;

class ComputeThread extends Thread {
	int[][] a;
	int[][] b;
	BlockedMat c;
	public int from;
	public int to;
	long timeDiff = 0;

	ComputeThread(int[][] a, int[][] b, BlockedMat c, int f, int t) {
	  this.a = a;
	  this.b = b;
	  this.c = c;
	  from = f;
	  to = t;
	}
	long getTimeDiff() {
	    return timeDiff;
	}
	public void run() { // overriding, must have this type
	  long s = System.currentTimeMillis();
	  int d = c.col;
	  for (int i = from; i<=to; i++) {
	      int sum = 0;
	      for (int j = 0; j<b.length; j++) {
	          sum += a[i/d][j] * b[j][i%d];
	      }
	      c.setPartialSum(i/d, i%d, sum);
	  }
	  long e = System.currentTimeMillis();
	  timeDiff = e - s;
	}

}
```

```java
class BlockedMat {
    public int row;
    public int col;
    public int[][] mat;

    BlockedMat(int r, int c) {
        row = r;
        col = c;
        mat = new int[row][col];
    }
    synchronized void setPartialSum(int r, int c, int sum) {
        mat[r][c] = sum;
    }
    public void printMatrix() {
        System.out.println("Matrix["+mat.length+"]["+mat[0].length+"]");
        int rows = mat.length;
        int columns = mat[0].length;
        int sum = 0;
        for (int i = 0; i < rows; i++) {
           for (int j = 0; j < columns; j++) {
              System.out.printf("%4d " , mat[i][j]);
              sum+=mat[i][j];
           }
           System.out.println();
         }
         System.out.println();
         System.out.println("Matrix Elements Sum = " + sum + "\n");
    }
}
```

```java
public class MatmultD {
  private static Scanner sc = new Scanner(System.in);
  private static BlockedMat multRes;

  public static void main(String [] args) {
    // set thread nums
    int thread_no=0;
    if (args.length==1) thread_no = Integer.valueOf(args[0]);
    else thread_no = 1;

    // read matrix a, b from .txt file
    int a[][]=readMatrix();
    int b[][]=readMatrix();

    multRes = new BlockedMat(a.length, b[0].length);
    ComputeThread[] ct = new ComputeThread[thread_no];
    int totalNeededComputation = a.length * b[0].length;

    // if there's too much thread set appropriate thread num;
    if (totalNeededComputation < thread_no) {
        System.out.println("Too much thread.. downgrade");
        thread_no = totalNeededComputation;
    }
    int d = totalNeededComputation / thread_no;

    long startTime = System.currentTimeMillis();
    for (int i = 0; i < thread_no; i++) {
        if (i == thread_no - 1) {
            ct[i] = new ComputeThread(a, b, multRes, i*d, totalNeededComputation-1);
        } else {
            ct[i] = new ComputeThread(a, b, multRes, i*d, (i+1)*d - 1);
        }
        ct[i].start();
    }

    // wait the other threads
    try {
        for (int i = 0 ;i < thread_no; i++) {
            ct[i].join();
        }
    } catch (InterruptedException e) { System.out.println("Error occurred!");}
    long endTime = System.currentTimeMillis();

    multRes.printMatrix();

    System.out.printf("Total number of threads: %d\n" , thread_no);
    System.out.printf("Calculation Time: %d ms\n" , endTime-startTime);

    for (int i = 0 ;i < thread_no; i++) {
        System.out.printf("[thread_no]:%2d , [Time]:%4d ms, [Range]:%4d...%4d\n", i, ct[i].getTimeDiff(), ct[i].from, ct[i].to);
    }
  }

  public static int[][] readMatrix() {
      int rows = sc.nextInt();
      int cols = sc.nextInt();
      int[][] result = new int[rows][cols];
      for (int i = 0; i < rows; i++) {
          for (int j = 0; j < cols; j++) {
              result[i][j] = sc.nextInt();
          }
      }
      return result;
  }
}
```



2 threads

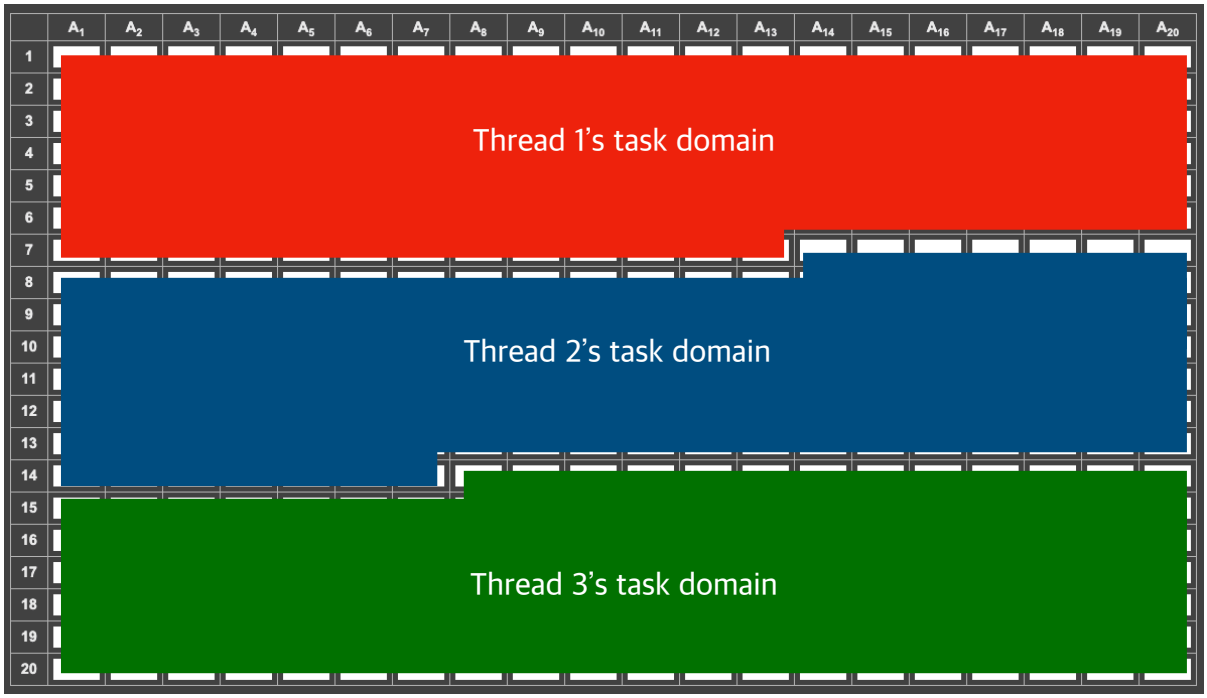


4 threads

It seems that the loads allocated to each thread are very similar.

Let's say we multiply two matrices A(m * n) and B(n * k). Then there result matrix C will be m*k. It means that we have to calculate m*k elements to get result C.

If you use "t" number of threads, then divide C's elements into t bunch of tasks for t threads.



For example, let's say the result matrix A's shape is 20 by 20. If you use 3 threads. Each of their task domain will be like the image above. Each thread calculate their separated task domain.

(d) How to compile and execute the source code

At terminal,
(*) javac MatmultD.java

After compilation execute the program.
(*) java MatmultD 6 < mat500.txt
6 means the number of threads to use, < mat500.txt means the file that contains two matrices is given as standard input.