2023 - Spring, Multi Core Computing Assignment1, Problem2 Performance Report
20150736 음호준

Environment

CPU Type
Apple M1 Pro(No Hyper Threading)

Number of cores
8(6 for Performance, 2 for Efficiency)

Clock Speed
Not Revealed

Memory Size
16GB

OS Type
mac os Ventura 13.3.1

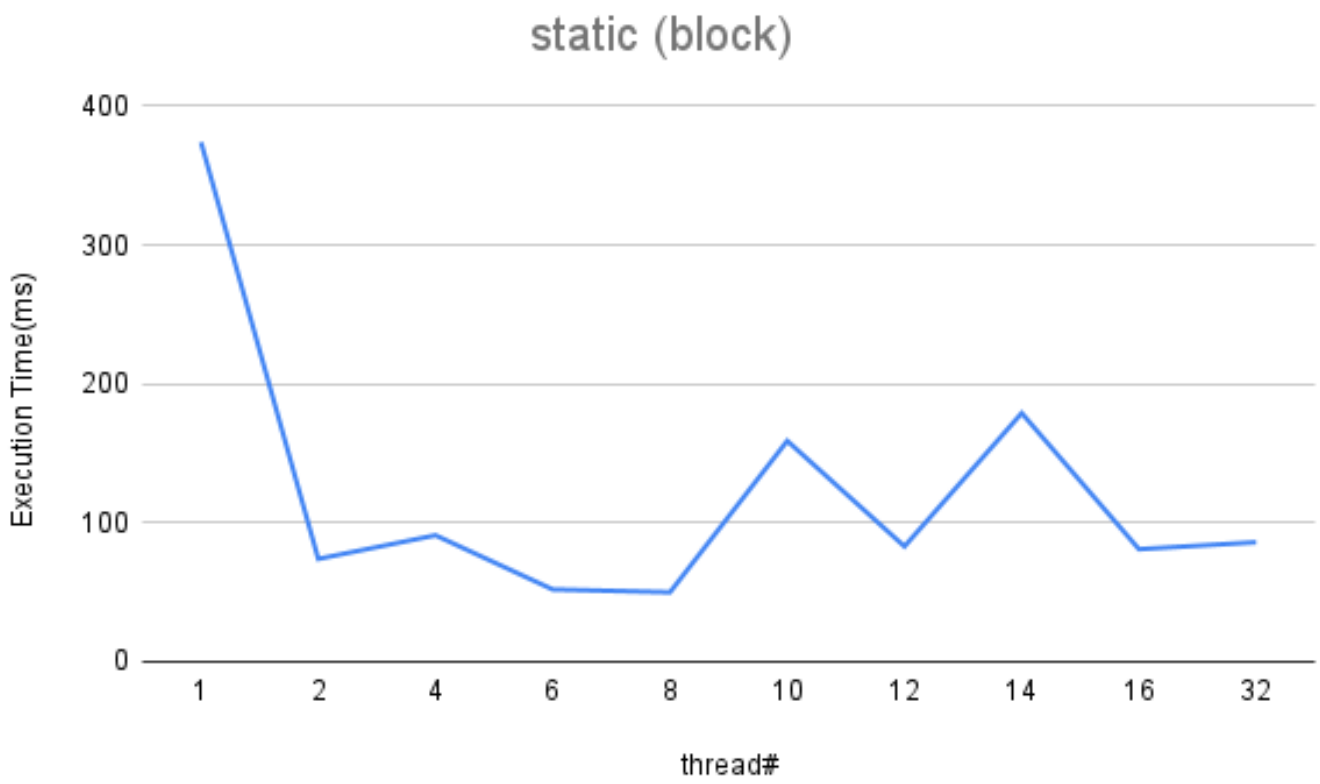
Tables and Graphs

Execution Time for each threads

Table

exec time(ms)\threads	1	2	4	6	8	10	12	14	16	32
static (block)	374	74	91	52	50	159	83	179	81	86

Graph

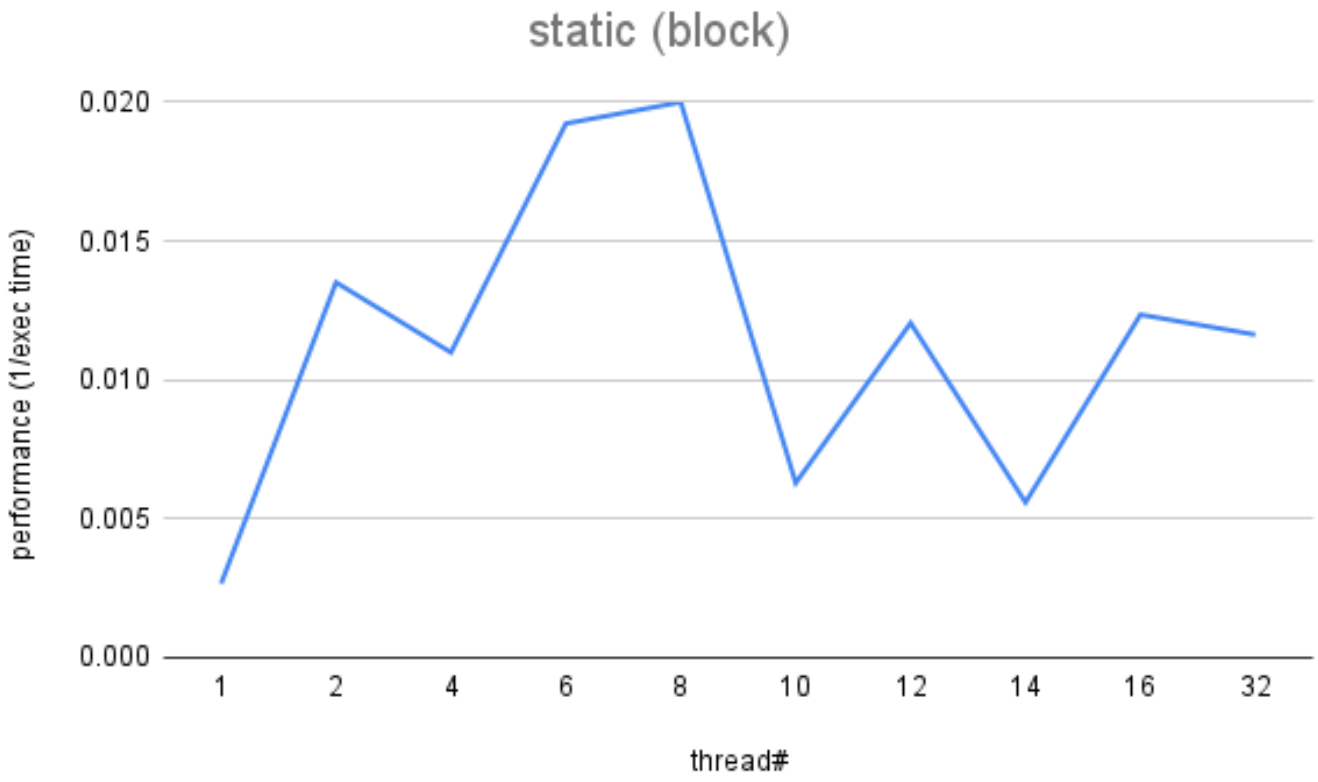


Performance for each threads

Table

performance (1/exec time)\threads	1	2	4	6	8	10	12	14	16	32
static (block)	0.00267	0.01351	0.01099	0.01923	0.02	0.00629	0.01205	0.00559	0.01235	0.01163

Graph



Explanation and Analysis of the results
I used block decomposition approach for load balancing matrix multiplication.

Block Decomposition
  Divide the numbers from 1 to 199999 into number of block according to the number of threads.
  For instance, if I have to use 4 threads, then divide the whole matrix into 4 blocks sequentially from top to bottom.
  It means that from 0th row to nth row(the last), the blocks composed of multiple rows which are partition of whole matrix
  are assigned to each worker(thread). Then each worker performs matrix multiplication.

This is an example with 5x5 matrix.
> 0 0 0 0 0 <- block 1 start
> 0 0 0 0 0
> 0 0 0 0 0 <- block 1 end
> 0 0 0 0 0 <- block 2 start
> 0 0 0 0 0 <- block 2 end

The Performance(1/execution time(ms)) is increase rapidly when the number of thread is small.
The performance has peak when 8 threads are used(same as the number of cores).
Then it gets down for 10 threads, then gets high again for 12 threads.
Then it gets high again for 16 threads, then gets little down for 32 threads(maximum).

The reasonable analysis for this phenomenon seems that subtasks are assigned to workers well until 8 threads(same as
cores) but in the case of using more threads than cores, assigning such a heavy task(matrix multiplication)

to threads
results in having more computing overhead when using cores more than the computer system's maximum cores. As the cores are heterogeneous(6 for performance, 2 for efficiency), there would be challenges to utilize cores for performance.
Trying to assign tasks to threads more than cores can lead to low performance because of managing processors such as scheduling,
context switching, job prioritizing.

In conclusion,
We can see the tendency of increasing performance by the number of thread increasing until the number of threads under
the number of cores, but there were also differences and challenges by approaches.
I learned that for the case of assigning heavy tasks(like matrix multiplication), should be careful of using proper
load balancing strategies along with efficient algorithms including understand of thread management strategies.

Source Code

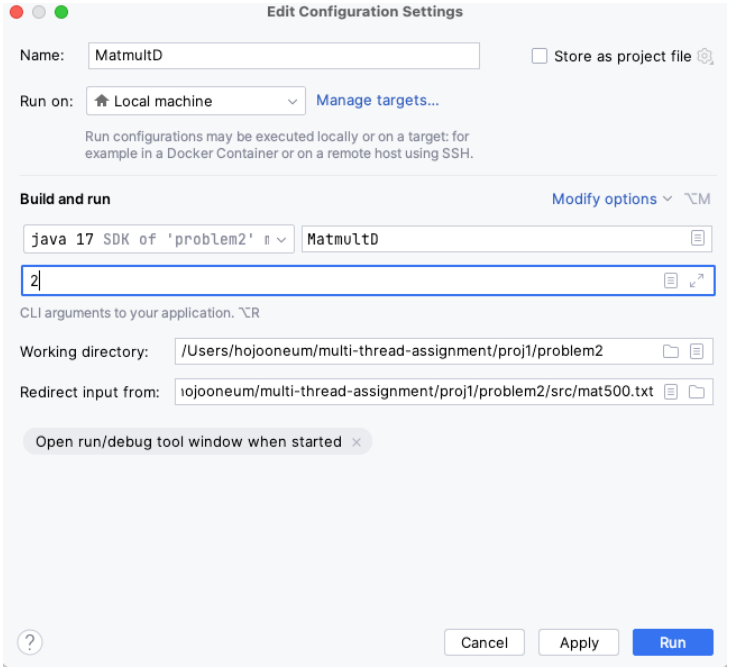
Block Decomposition for Matrix Multiplication
```java
import java.util.*;
import java.lang.*;

// command-line execution example) 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
//
// In eclipse, set the argument value and file input by using the menu [Run]->[Run Configurations]->{[Arguments], [Common->Input File]}.

// Original JAVA source code: http://stackoverflow.com/questions/21547462/how-to-multiply-2-dimensional-arrays-matrix-multiplication
public class MatmultD
{
    private static Scanner sc = new Scanner(System.in);
    public static void main(String [] args)
    {
        int thread_no=0;
        if (args.length==1) thread_no = Integer.valueOf(args[0]);
        else thread_no = 1;

        int a[][]=readMatrix();
        int b[][]=readMatrix();
        // We use static load balancing approach with multi-threads 2,4,6...32
        int[][] c;
        long startTime = System.currentTimeMillis();
        if (thread_no == 1) {
            c = multMatrix(a,b);
        }
        else {
            c = multiplyMatrixInParallel(a,b, thread_no);
        }
        long endTime = System.currentTimeMillis();

        //printMatrix(a);
        //printMatrix(b);
        printMatrix(c);
```

```
        //System.out.printf("thread_no: %d₩n" , thread_no);
        //System.out.printf("Calculation Time: %d ms₩n" , endTime-startTime);

        System.out.printf("[thread_no]:%2d , [Time]:%4d ms₩n", thread_no, endTime-startTime);
    }

    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;
    }

    public static void printMatrix(int[][] mat) {
        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 Sum = " + sum + "₩n");
    }

    private static int[][] multiplyMatrixInParallel(int a[][], int b[][], int threads) {
        // We use divide and conquer approach for this multiplication.
        // Decompose the matrix into number of threads.
        // For example, 4x4 -> 1x4 with 4 threads.
        final int aRows = a.length;
        final int aCols = a[0].length;
        final int[][] resultMatrix = new int[aRows][aCols];
        Thread[] workers = new Thread[threads];
        int rowStart = 0; int rowEnd;
        final int rowStride = Math.floorDiv(aRows,threads);

        int wid = 0;
        for (int i=0;i<threads;i++) {
            rowStart = i * rowStride;
            rowEnd = (i != threads-1) ? rowStart + rowStride : aRows;
            workers[wid] = new MatMultWorker(wid, a, b, resultMatrix, aCols, rowStart, rowEnd, 0, aCols);
            workers[wid].start();
            wid++;
        }
        try {
            for (Thread worker : workers) {
                worker.join();
            }
        } catch (InterruptedException e) {}
        return resultMatrix;
```

```
    }

    public static int[][] multMatrix(int a[][], int b[][]){//a[m][n], b[n][p]
        if(a.length == 0) return new int[0][0];
        if(a[0].length != b.length) return null; //invalid dims

        int n = a[0].length;
        int m = a.length;
        int p = b[0].length;
        int ans[][] = new int[m][p];

        for(int i = 0;i < m;i++){
            for(int j = 0;j < p;j++){
                for(int k = 0;k < n;k++){
                    ans[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return ans;
    }
}

public class MatMultWorker extends Thread {
    final int wid;
    final int[][] a; final int[][] b; final int[][] resultMatrix;
    final int aCols; final int rowStart; final int rowEnd;
    final int colStart; final int colEnd;
    public MatMultWorker(int wid, int[][] a, int[][] b, int[][] resultMatrix, int aCols, int rowStart, int
rowEnd, int colStart, int colEnd) {
        super("wid "+wid);
        this.wid = wid;
        this.a = a;
        this.b = b;
        this.aCols = aCols;
        this.resultMatrix = resultMatrix;
        this.rowStart = rowStart;
        this.rowEnd = rowEnd;
        this.colStart = colStart;
        this.colEnd = colEnd;
    }

    public void run() {
        System.out.println(getName()+" is working.");
        long startTime = System.currentTimeMillis();
        for (int i=rowStart;i<rowEnd;i++) {
            for (int j=colStart;j<colEnd;j++) {
                for (int k=0;k<aCols;k++) {
                    resultMatrix[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println(getName()+" is done.");
        String execTimeMsg = "Execution time of " + getName() + " is " + (endTime - startTime);
        System.out.println(execTimeMsg);
    }
}

```

Execution Image and Output Image

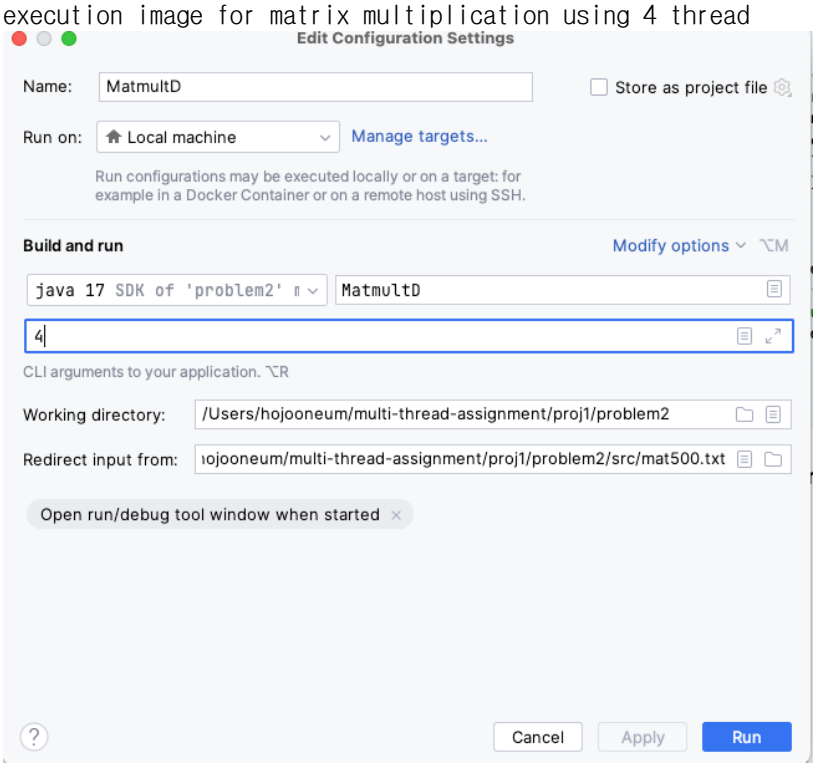
execution image for matrix multiplication using 2 thread



output image for matrix multiplication using 2 thread

```
477 471 486 495 438 487 519 516 506 503 478 483 490 492 502 467 490 475 474 506 483 518 537 473 455 488 453 496 521 539
446 471 501 462 435 493 535 518 538 485 494 508 496 515 532 489 480 486 481 495 525 519 509 476 461 491 465 511 499 536
473 468 473 487 457 455 503 497 498 497 496 468 480 497 502 453 477 475 498 507 493 508 513 472 469 495 472 455 493 535
442 455 472 461 402 448 480 514 456 480 450 481 466 475 460 435 461 468 459 496 478 498 491 458 443 427 433 464 488 503
474 417 462 467 415 471 512 539 491 493 457 483 479 519 499 464 454 439 490 500 478 502 503 423 454 468 458 441 466 517
466 483 493 487 448 476 501 531 513 499 487 470 524 520 507 478 492 489 500 503 510 504 520 469 456 482 463 507 499 560
480 496 530 505 495 530 548 528 562 546 520 518 521 546 534 513 535 510 546 527 522 558 551 546 505 532 507 497 536 576
464 471 492 477 429 483 468 473 526 514 482 481 512 502 493 487 474 469 481 500 487 527 520 467 427 471 456 500 504 524
506 510 533 515 479 519 540 545 551 536 533 524 540 528 536 523 505 505 514 535 550 565 555 494 480 494 495 550 515 606
487 499 524 514 479 486 511 536 564 581 513 504 526 542 530 497 501 491 484 510 505 549 557 484 493 492 508 532 515 577
462 453 477 475 435 460 502 505 510 516 464 476 466 487 494 455 476 457 465 466 490 509 492 437 429 485 434 461 469 511
517 510 554 517 450 527 523 549 545 527 517 516 499 533 526 504 509 541 492 541 511 548 541 505 475 514 475 519 546 568
505 525 565 535 457 517 543 530 558 547 515 547 511 586 543 545 486 532 520 554 517 564 564 540 517 544 513 521 562 583

Matrix Sum = 125231132

[thread_no]: 2 , [Time]:  69 ms
```

execution image for matrix multiplication using 4 thread



output image for matrix multiplication using 4 thread

```
 492  472  512  512  428  450  506  500  519  518  493  463  497  530  496  470  469  444  463  489  500  542  517  464  469  503  493  484  520  548
 493  496  491  460  434  471  538  486  531  501  521  499  488  500  500  515  514  489  477  515  511  541  527  471  480  526  452  473  528  535
 477  471  486  495  438  487  519  516  506  503  478  483  490  492  502  467  490  475  474  506  483  518  537  473  455  488  453  496  521  539
 446  471  501  462  435  493  535  518  538  485  494  508  496  515  532  489  480  486  481  495  525  519  509  476  461  491  465  511  499  536
 473  468  473  487  457  455  503  497  498  497  496  468  480  497  502  453  477  475  498  507  493  508  513  472  469  495  472  455  493  535
 442  455  472  461  402  448  480  514  456  480  450  481  466  475  460  435  461  468  459  496  478  498  491  458  443  427  433  464  488  503
 474  417  462  467  415  471  512  539  491  493  457  483  479  519  499  464  454  439  490  500  478  502  503  423  454  468  458  441  466  517
 466  483  493  487  448  476  501  531  513  499  487  470  524  520  507  478  492  489  500  503  510  504  520  469  456  482  463  507  499  560
 480  496  530  505  495  530  548  528  562  546  520  518  521  546  534  513  535  510  546  527  522  558  551  546  505  532  507  497  536  576
 464  471  492  477  429  483  468  473  526  514  482  481  512  502  493  487  474  469  481  500  487  527  520  467  427  471  456  500  504  524
 506  510  533  515  479  519  540  545  551  536  533  524  540  528  536  523  505  505  514  535  550  565  555  494  480  494  495  550  515  606
 487  499  524  514  479  486  511  536  564  581  513  504  526  542  530  497  501  491  484  510  505  549  557  484  493  492  508  532  515  577
 462  453  477  475  435  460  502  505  510  516  464  476  466  487  494  455  476  457  465  466  490  509  492  437  429  485  434  461  469  511
 517  510  554  517  450  527  523  549  545  527  517  516  499  533  526  504  509  541  492  541  511  548  541  505  475  514  475  519  546  568
 505  525  565  535  457  517  543  530  558  547  515  547  511  586  543  545  486  532  520  554  517  564  564  540  517  544  513  521  562  583

Matrix Sum = 125231132

[thread_no]: 4 , [Time]:  71 ms
```

Guide for Compilation and Execution

Requirements for compilation and execution
- OS: macos(latest)
  you can use other OS of course, but please setup specific compilation, execution options by yourself.
- Processor: M1 Pro(processors can differ by each case of testing environments)
- Cores: 8(6 for performance, 2 for efficiency)
- Memory: 16GB(please use more than 8GB).
- IntelliJ IDEA 2023.1 (Ultimate Edition)
  Runtime version: 17.0.6+10-b829.5 aarch64
  VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.
  you can use eclipse ide of course, but please setup eclipse by yourself.
- Compiler: Javac
- JDK: Java 17

Steps to compilation and execution
1. Open problem2 directory with IntelliJ idea.
2. Open execution class file(MatmultD).
3. Find "Run" option on ide and Click "Edit Configurations" under "Run" option.
4. Add applications for main class.
   a. Set application name for execution.
   b. Set main class for execution.
   c. Set run configuration to "local machine".
   d. Set program arguments and environment variables.
5. Run execution class.
6. After execution, you can see the compiled java byte code files in subdirectory "out/production/problem2".