Lab Session Week 11

Student ID	Student Name	Student Email
32844700	Teh Jia Xuan	Jteh0015@student.monash.edu

Task 1 Code

```
////////////////////////////////////////////////////////////////////////////////////
///
// MatrixMul_1D_bin.c
// ---------------------------------------------------------------------------------
--
//
// Multiplies two matrices and writes the resultant multiplication into a binary
file.
//
////////////////////////////////////////////////////////////////////////////////////
///
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <pthread.h>
#define NUM_THREADS 8

int *pMatrixA, *pMatrixB;
unsigned long long *pMatrixC;
int rowA, rowB, rowC, colA, colB, colC, commonPoint;

void *ThreadFunc2(void *pArg){
    // Assume the matrix arrays, row, column and commonPoint variables are global
variables and initialised in main function.
    int i, j, k;
    int my_rank = *((int*)pArg);

    //calculate the tile partition
    //using result will divide into half even number process left odd number
process right
    int row_start_point, row_end_point;
    int col_start_point, col_end_point;
    int threadColDiv = 1;
    int threadColDivRemain = 0;
    int threadRowDiv = 1;
    int threadRowDivRemain = 0;

    threadColDiv = colC / 2;
    threadColDivRemain = colC % 2;
    threadRowDiv = rowC / (NUM_THREADS / 2);
    threadRowDivRemain = rowC % (NUM_THREADS / 2);
```

```
    if(my_rank % 2 == 0){
        // Even thread
        if(my_rank == (NUM_THREADS - 2)){
            // Last even thread
            row_start_point = (my_rank / 2) * threadRowDiv;
            row_end_point = row_start_point + threadRowDiv + threadRowDivRemain;
        }else{
            // Not last even thread
            row_start_point = (my_rank / 2) * threadRowDiv;
            row_end_point = row_start_point + threadRowDiv;
        }
        col_start_point = 0;
        col_end_point = threadColDiv;
    }else{
        // Odd thread
        if(my_rank == (NUM_THREADS - 1)){
            // Last odd thread
            row_start_point = (my_rank / 2) * threadRowDiv;
            row_end_point = row_start_point + threadRowDiv + threadRowDivRemain;
        }else{
            // Not last odd thread
            row_start_point = (my_rank / 2) * threadRowDiv;
            row_end_point = row_start_point + threadRowDiv;
        }
        col_start_point = threadColDiv;
        col_end_point = col_start_point + threadColDiv + threadColDivRemain;
    }

    // Matrix multiplication
    for(i = row_start_point; i < row_end_point; i++){
        for(j = col_start_point; j < col_end_point; j++){
            for(k = 0; k < commonPoint; k++){
                pMatrixC[(i*colC)+j] += (pMatrixA[(i*colA)+k] *
pMatrixB[(k*colB)+j]);
            }
        }
    }
    return NULL;
}

int main()
{
    // Variables
    int i = 0, j = 0, k = 0;

    /* Clock information */
    struct timespec start, end;
    double time_taken;
    pthread_mutex_t mutex;
```

```c
    pthread_mutex_init(&mutex, NULL);
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    clock_gettime(CLOCK_MONOTONIC, &start);

    // 1. Read Matrix A
    rowA = 0;
    colA = 0;

    printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

    printf("Reading Matrix A - Start\n");

    FILE *pFileA = fopen("MA_500x500.bin", "rb");
    fread(&rowA, sizeof(int), 1, pFileA);
    fread(&colA, sizeof(int), 1, pFileA);

    pMatrixA = (int*)malloc((rowA*colA) * sizeof(int));
    for(i = 0; i < rowA; i++){
        fread(&pMatrixA[i*colA], sizeof(int), colA, pFileA);
    }
    fclose(pFileA);

    printf("Reading Matrix A - Done\n");

    // 2. Read Matrix B
    rowB = 0;
    colB = 0;

    printf("Reading Matrix B - Start\n");

    FILE *pFileB = fopen("MB_500x500.bin", "rb");
    fread(&rowB, sizeof(int), 1, pFileB);
    fread(&colB, sizeof(int), 1, pFileB);

    pMatrixB = (int*)malloc((rowB*colB) * sizeof(int));
    for(i = 0; i < rowB; i++){
        fread(&pMatrixB[i*colB], sizeof(int), colB, pFileB);
    }
    fclose(pFileB);

    printf("Reading Matrix B - Done\n");

    // 3. Perform matrix multiplication
    printf("Matrix Multiplication - Start\n");

    rowC = rowA;
    colC = colB;
    pMatrixC = (unsigned long long*)calloc((rowC*colC), sizeof(unsigned long
long));
```

```
    commonPoint = colC;

    int rows_per_thread = rowC/NUM_THREADS;
    int cols_per_thread = colC/NUM_THREADS;

    for (i = 0; i < NUM_THREADS; i++){
        //creating threads to compute
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, ThreadFunc2, &thread_ids[i]);
    }

    for (i = 0; i< NUM_THREADS; i++){
        //join them tgt after computing
        pthread_join(threads[i], NULL);
    }



    printf("Matrix Multiplication - Done\n");

    // 4. Write resuls to a new file
    printf("Write Resultant Matrix C to File - Start\n");

    FILE *pFileC = fopen("MC_500x500.bin", "wb");
    fwrite(&rowC, sizeof(int), 1, pFileC);
    fwrite(&colC, sizeof(int), 1, pFileC);
    for(i = 0; i < rowC; i++){
        fwrite(&pMatrixC[i*colC], sizeof(unsigned long long), colC, pFileC); // use
fwrite to write one row of columns to the file
    }



    fclose(pFileC);
    printf("Write Resultant Matrix C to File - Done\n");
    //time taken for them to complete overall program
    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time (Including read, multiplication and write)(s): %lf\n",
time_taken);    // ts

    // Clean up
    free(pMatrixA);
    free(pMatrixB);
    free(pMatrixC);

    printf("Matrix Multiplication using 1-Dimension Arrays - Done\n");
    return 0;
}
```

Task 1

Overall Time	0.873422
Computational Time (parallelizable)	0.779986
Read write time (serial)	0.093436
Parallelizable code	0.779986 / 0.873422 = 0.893
Serial	0.107
Theoretical Speed Up	1/ 0.107 + (0.893/8) = 4.573

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 0.873422
Overall computation time(s): 0.779986
Matrix Multiplication using 1-Dimension Arrays - Done
```

<u>Table</u>

Number of CPU cores or logical processes: 8 **Number of threads used for POSIX/OMP: 8**					
Matrix size	**500x500**	**1000x1000**	**2000x2000**	**3000x3000**	**4000x4000**
Serial time (s)	1.036289	7.205710	52.871768	180.674973	424.491226
Parallel time POSIX/OMP	0.329884	2.313118	12.446038	36.649904	83.631660
Speed up (Ts/Tp)	3.141374	3.11515	4.24808	4.929753	5.075724

<u>Analysis</u>

The theoretical speed up is 4.574 which is calculated by the amdahl's law

Speed up = 1/rs + rp/n

Rs = 0.093436/0.873422 = 0.107

Rp = 1 - 0.107 = 0.893
Speed up = 1/ 0.107 + 0.893/8 = 4.574

500x500
Serial code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 1.036289
Matrix Multiplication using 1-Dimension Arrays - Done
```

Parallel code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 0.329884
Matrix Multiplication using 1-Dimension Arrays - Done
```

1000x1000
Serial code

```
Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 7.205710
Matrix Multiplication using 1-Dimension Arrays - Done
```

Parallel code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 2.313118
Matrix Multiplication using 1-Dimension Arrays - Done
```

2000x2000

Serial code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 52.871768
Matrix Multiplication using 1-Dimension Arrays - Done
```

Parallel code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 12.446038
Matrix Multiplication using 1-Dimension Arrays - Done
```

3000x3000

Serial code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 180.674973
Matrix Multiplication using 1-Dimension Arrays - Done
```

Parallel code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 36.649904
Matrix Multiplication using 1-Dimension Arrays - Done
```

4000x4000

Serial code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 424.491226
Matrix Multiplication using 1-Dimension Arrays - Done
```

Parallel code

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Matrix Multiplication - Start
Matrix Multiplication - Done
Write Resultant Matrix C to File - Start
Write Resultant Matrix C to File - Done
Overall time (Including read, multiplication and write)(s): 83.631660
```

Check Correctness with 500 x 500 Matrix Mul

Serial code provided

```
183362220 183733955 187995993 178581697 182309507 175452717 183428578 182631857 177669089 181048356 180988338 181087198 182360549 187923874 179732629 187023433 178696342 180543987 174945301 177873984 180228400 172529718 180075324
183390400 177442674 186759017 174622780 174340435 183249889 180255385 175121338 189850921 175895714 173164233 176156029 178345863 184944950 184493454 182962628 177478503 184739506 176581176 182206020 182479893 178935861 181930498
176135860 183826729 181360196 185568374 185445531 180858724 181316124 179781645 182183781 180341949 189808242 188137147 189585695 173889725 185244102 179082347 366842402 372615428 381151470 365517478 361582920 355428860 365327994
380642732 371835762 381278918 370941916 370880194 358006310 363181790 363910406 370250282 354242862 368871122 370620758 361135010 383193216 366704920 367796134 381533092 372996062 357808706 365046190 351376698 361179586 366670986
381437754 353617440 363326732 363753600 370107784 359835030 365969462 361879650 186972908 175792321 174907359 191179502 180647380 180447650 173612306 186766403 179298183 180071281 181719346 187093125 189154523 175446778 181987720
180184122 187116923 178757932 181194994 182441977 178137775 175653123 179736233 184009468 369138088 371059298 350087654 370260702 363533802 370164608 356191892 366581176 360277030 363903804 369946962 361871696 374011152 368022304
363462646 349805460 353925938 361219296 372280876 353064484 355287214 356455706 370112982 360841792 351317810 367645080 352466430 361146456 367312438 357141190 362173022 362371410 362108634 354027832 369661048 362506008 380765912
352656670 179066997 187272047 180767788 175171530 179977659 184708259 182961251 169356909 177297095 184205508 186702851 177871066 181934492 184841396 184177011 186334594 181084796 180067489 173395669 183371115 171482204 173133348
186913811 186964869 368110014 380338638 364528296 371942690 370710660 360826740 366414012 365493806 364829728 359652128 371418496 361033274 349926368 363194366 357546950 357587962 369409868 353463112 370475616 368478556 352362580
353592268 349337300 371153006 362246836 364178682 365150330 353386514 368479266 368831084 357491956 369374468 358713602 369518680 359677596 350392952 365249002 353698048 181564267 184302249 186125602 173308500 181584695 187583167
180575591 182498384 180827026 186232330 183234763 185079270 185665497 187828476 183123834 179938111 181408346 175131088 177329260 182191236 179080976 185565426 178151651 177364171 352829276 368956500 353354470 356698072 360067362
349606948 353755596 369638950 362209668 340710106 346806494 361175864 368358880 373223290 354030436 367980588 364177000 361511838 370522160 366984898 373173740 352903978 350059282 365630536 353441556 360213076 377621168 355605638
351832118 370419250 388990452 366724838 357819154 359929228 359733642 364979318 367863640 374411736 181955933 184730677 189749823 172465793 177750304 175819365 176013274 186166314 177045964 180706022 183765324 183266263 183107966
182683099 183821890 182870294 182503626 183819411 178707906 176135570 183200784 190243407 183088053 182395249 362173558 365291306 366676542 358755802 360256574 369695016 361053914 366782004 351206616 351250708 367758398 366261062
358269562 371076418 367912512 372968932 362297778 362410050 359090572 368198152 355638346 357219394 345488424 360540796 371148986 354352656 367750620 351514156 360299606 365388938 367200440 343407968 352071102 359257446 360659150
364908342 362993190 366906014 189540865 178769078 181970333 179678353 183538962 180390711 186240156 176651174 178830213 183160685 181783378 174646646 179405267 182146616 190374578 190238320 177937563 184256644 179512085 183290508
183136092 187326457 182332373 179822328 365448498 371341216 366154636 360833550 363040630 361366846 360956454 364926356 379865412 351277974 359854376 360149496 359371680 367426654 358591644 365728900 362725464 365056680 365708262
362716436 356954330 367536460 353853502 360388708 369951430 367647182 372072676 362394164 374195960 360808868 365709406 365295306 353236944 361574556 363782848 362044408 360774824 362051414 181898639 183089842 177218830 180378940
181990445 179075013 174658406 177436951 180810206 180571825 180054860 184460444 181365921 179183670 179141338 177289121 178612339 189130595 179434944 183035711 180800197 181508217 183582167 185539956 377746014 365314952 370657864
366906570 367846142 356954048 363739092 368266342 364773944 355015858 365743576 363157308 359888132 366365306 365888100 376457454 371591142 356884012 371925640 370970692 358273902 363531824 362442352 367449512 343959188 369743082
356334954 350788674 369122624 372846210 354977778 359593026 357267576 352930298 361120042 358896420 355055276 349638902 188023388 183313449 184206503 181896225 171664651 186209668 184250889 179698889 180588369 175701188 178280314
188048416 185415091 188068533 186065424 182286410 174560884 181657265 181514221 178983906 176006387 179341410 183175924 177406666 186450956 181500261 179568580 178979538
```

Parallel code with POSIX

```
183362220 183733955 187995993 178581697 182309507 175452717 183428578 182631857 177669089 181048356 180988338 181087198 182360549 187923874 179732629 187023433 178696342 180543987 174945301 177873984 180228400 172529718 180075324
183390400 177442674 186759017 174622780 174340435 183249889 180255385 175121338 189850921 175895714 173164233 176156029 178345863 184944950 184493454 182962628 177478503 184739506 176581176 182206020 182479893 178935861 181930498
176135860 183826729 181360196 185568374 185445531 180858724 181316124 179781645 182183781 180341949 189808242 188137147 189585695 173889725 185244102 179082347 366842402 372615428 190575735 182758739 180791460 177714430 182663997
190321366 185917881 190639459 185470958 185440097 179003155 181590895 181955203 185125141 177121431 184435561 185310379 180567505 191596608 183352460 183898067 190766546 186498031 178904353 182523095 175688349 180589793 183335493
190718877 176808720 181663366 181876800 185053892 179917515 182984731 180939825 186972908 175792321 174907359 191179502 180647380 180447650 173612306 186766403 179298183 180071281 181719346 187093125 189154523 175446778 181987720
180184122 187116923 178757932 181194994 182441977 178137775 175653123 179736233 184009468 369138088 371059298 175043827 185130351 181766901 185082304 178095946 183290588 180138515 181951902 184973481 180935848 187005576 184011152
181731323 174902730 176962969 180609648 186140438 176532242 177643607 178227853 185056491 180420896 175658905 183822540 176233215 180573228 183656219 178570595 181086511 181185705 181054317 177013916 184830524 181253004 190382956
176328335 179066997 187272047 180767788 175171530 179977659 184708259 182961251 169356909 177297095 184205508 186702851 177871066 181934492 184841396 184177011 186334594 181084796 180067489 173395669 183371115 171482204 173133348
186913811 186964869 368110014 380338638 182264148 185971345 185355330 180413370 183207006 182746903 182414864 179826064 185709248 180516637 174963184 181597183 178773475 178793981 184704934 176731556 185237808 184239278 176181290
176796134 174668650 185576503 181123418 182089341 182575165 176693257 184239633 184415542 178745978 184687234 179356801 184759340 179838798 175196476 182624501 176849024 181564267 184302249 186125602 173308500 181584695 187583167
180575591 182498384 180827026 186232330 183234763 185079270 185665497 187828476 183123834 179938111 181408346 175131088 177329260 182191236 179080976 185565426 178151651 177364171 352829276 368956500 176677235 178349036 180033681
174803474 176877798 184819475 181104834 170355053 173403247 180587932 184179440 186611645 177015218 183990294 182088500 180755919 185261080 183492449 186586870 176451989 175029641 182815268 176720778 180106538 188810584 177802819
175916059 185209625 194495226 183362419 178909577 179964614 179866821 182489659 183931820 187205868 181955933 184730677 189749823 172465793 177750304 175819365 176013274 186166314 177045964 180706022 183765324 183266263 183107966
182683099 183821890 182870294 182503626 183819411 178707906 176135570 183200784 190243407 183088053 182395249 362173558 365291306 183338271 179377901 180128287 184847508 180526957 183391002 175603308 175625354 183879199 183130531
179134781 185538209 183956256 186484466 181148889 181205025 179545286 184099076 177819173 178609697 172744212 180270398 185574493 177176328 183875310 175757078 180149803 182694469 183600220 171703984 176035551 179628723 180329575
182454171 181496595 183453007 189540865 178769078 181970333 179678353 183538962 180390711 186240156 176651174 178830213 183160685 181783378 174646646 179405267 182146616 190374578 190238320 177937563 184256644 179512085 183290508
183136092 187326457 182332373 179822328 365448498 371341216 183077318 180416775 181520315 180683423 180478227 182463178 189932706 175638987 179927188 180074748 179685840 183713327 179295822 182864450 181362732 182528340 182854131
181358218 178477165 183768230 176926751 180194354 184975715 183823591 186036338 181197082 187097980 180404434 182854703 182647653 176618472 180787278 181891424 181022204 180387412 181025707 181898639 183089842 177218830 180378940
181990445 179075013 174658406 177436951 180810206 180571825 180054860 184460444 181365921 179183670 179141338 177289121 178612339 189130595 179434944 183035711 180800197 181508217 183582167 185539956 377746014 365314952 185328932
183453285 183923071 178477024 181869546 184133171 182386972 177507929 182871788 181578654 179944066 183182653 182944050 188228727 185795571 178442006 185962820 185485346 179136951 181765912 181221176 183724756 171979594 184871541
178167477 175394337 184561312 186423105 177488889 179796513 178633788 176465149 180560021 179448210 177527638 174819451 188023388 183313449 184206503 181896225 171664651 186209668 184250889 179698889 180588369 175701188 178280314
188048416 185415091 188068533 186065424 182286410 174560884 181657265 181514221 178983906 176006387 179341410 183175924 177406666 186450956 181500261 179568580 178979538
```

Task 2 – Code

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <mpi.h>

//combine local matrices to the global matrix
void combineLocalMatrices(unsigned long long *localMatrix, unsigned long long
*globalMatrix, int row_start, int row_end, int col_start, int col_end, int colC) {
    //only loop through the necessary row and column for the local matrix
    //+= is because each of the elem in global matrix initially is 0, so x+0 = x
    for(int row = row_start; row < row_end; row++){
        for (int col = col_start; col < col_end; col++){
            // Add localMatrix to globalMatrix element by element
            globalMatrix[row * colC + col] += localMatrix[row * colC + col];
        }
    }
}


int main(int argc, char *argv[])
{
    // Variables
    int i = 0, j = 0, k = 0;
    int my_rank, size;
    int position;
    int pack_size;
    int row_start_point, row_end_point;
    int col_start_point, col_end_point;
    unsigned long long *pMatrixC;
    int rowA, rowB, rowC, colA, colB, colC, commonPoint;
    int *pMatrixA = NULL, *pMatrixB = NULL;


    /* Clock information */
    struct timespec start, end, startComp, endComp;
    double time_taken;

    clock_gettime(CLOCK_MONOTONIC, &start);
    //initialise mpi
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);



    // 1. Read Matrix A
    rowA = colA = rowB = colB = 0;
    if (my_rank == 0) {
```

```c
        printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

        // Reading Matrix A
        printf("Reading Matrix A - Start\n");
        FILE *pFileA = fopen("MA_500x500.bin", "rb");
        if (!pFileA) {
            perror("File A opening failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        fread(&rowA, sizeof(int), 1, pFileA);
        fread(&colA, sizeof(int), 1, pFileA);
        pMatrixA = (int *)malloc((rowA * colA) * sizeof(int));
        if (pMatrixA == NULL) {
            perror("Memory allocation for Matrix A failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        for (i = 0; i < rowA; i++) {
            //reading row by row using fread
            fread(&pMatrixA[i * colA], sizeof(int), colA, pFileA);
        }
        fclose(pFileA);
        printf("Reading Matrix A - Done\n");

        // Reading Matrix B
        printf("Reading Matrix B - Start\n");
        FILE *pFileB = fopen("MB_500x500.bin", "rb");
        if (!pFileB) {
            perror("File B opening failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        fread(&rowB, sizeof(int), 1, pFileB);
        fread(&colB, sizeof(int), 1, pFileB);
        pMatrixB = (int *)malloc((rowB * colB) * sizeof(int));
        if (pMatrixB == NULL) {
            perror("Memory allocation for Matrix B failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        for (i = 0; i < rowB; i++) {
            //reading row by row
            fread(&pMatrixB[i * colB], sizeof(int), colB, pFileB);
        }
        fclose(pFileB);
        printf("Reading Matrix B - Done\n");

        // Initialize Matrix C
        rowC = rowA;
        colC = colB;
        commonPoint = colA; // Initialize commonPoint
    }
    //broadcast rows to all the processses
```

```
    MPI_Bcast(&rowA, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colA, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&rowB, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colB, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&rowC, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colC, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&commonPoint, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast
commonPoint
    //assign local matrix result
    pMatrixC = (unsigned long long *)calloc((rowC * colC), sizeof(unsigned long
long));

    //calculate the tile partition
    //using result will divide into half even number process left odd number
process right
    int threadColDiv = colC / 2;
    int threadColDivRemain = colC % 2;
    int threadRowDiv = rowC / (size / 2);
    int threadRowDivRemain = rowC % (size / 2);

    if (my_rank == 0) {
        for (int temprank = 1; temprank < size; temprank++) {

            position = 0;

            if (temprank % 2 == 0) {
                // Even thread
                if (temprank == (size - 2)) {
                    // Last even thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv +
threadRowDivRemain;
                } else {
                    // Not last even thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv;
                }
                col_start_point = 0;
                col_end_point = threadColDiv;
            } else {
                // Odd thread
                if (temprank == (size - 1)) {
                    // Last odd thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv +
threadRowDivRemain;
                } else {
                    // Not last odd thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv;
```

```
                }
                col_start_point = threadColDiv;
                col_end_point = col_start_point + threadColDiv +
threadColDivRemain;
            }
            //send to all processes
            //after getting their own row_start_point, row end  point, col start
and col end
            MPI_Send(&row_start_point, 1, MPI_INT, temprank, 0, MPI_COMM_WORLD);
            MPI_Send(&row_end_point, 1, MPI_INT, temprank, 1, MPI_COMM_WORLD);
            MPI_Send(&col_start_point, 1, MPI_INT, temprank, 2, MPI_COMM_WORLD);
            MPI_Send(&col_end_point, 1, MPI_INT, temprank, 3, MPI_COMM_WORLD);

            //create subarray to store all the necessary rows and column
            //needed for multiplication for that process
            int subarray_size = rowC*colC;
            int *subarrayA = (int *)malloc(subarray_size * sizeof(int));
            int *subarrayB = (int *)malloc(subarray_size * sizeof(int));
            if (!subarrayA || !subarrayB) {
                perror("Subarray allocation failed");
                MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
            }

            //loop through matrix A to copy all the necessary cells
            for (i = row_start_point; i < row_end_point; i++) {
                for (k = 0; k < commonPoint; k++){
                    subarrayA[((i - row_start_point) * colA) + k] = pMatrixA[(i *
colA) + k];
                }
            }

            //loop through matrix B to copy all the necessary cells
            for(j = col_start_point; j < col_end_point; j++){
                for (k = 0; k < commonPoint; k++) {
                    subarrayB[((j - col_start_point) * rowA) + k] = pMatrixB[(k *
colB) + j];
                }
            }

            //pack size according to the subarray size
            MPI_Pack_size(subarray_size + (sizeof(unsigned long long) * 2),
MPI_INT, MPI_COMM_WORLD, &pack_size);
            int buffer_size = pack_size;
            //initialise buffer for subarray to send to process
            char *bufferA = (char *)malloc(buffer_size);
            char *bufferB = (char *)malloc(buffer_size);
            clock_gettime(CLOCK_MONOTONIC, &startComp);

            //pack the time and subarrays
```

```
            MPI_Pack(&startComp.tv_sec, 1, MPI_UINT64_T, bufferA, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Pack(&startComp.tv_nsec, 1, MPI_UINT64_T, bufferA, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Pack(subarrayA, subarray_size, MPI_INT, bufferA, buffer_size,
&position, MPI_COMM_WORLD);

            //send the subarrays and time together to process
            MPI_Send(bufferA, position, MPI_PACKED, temprank, 4, MPI_COMM_WORLD);
            position = 0;
            MPI_Pack(subarrayB, subarray_size, MPI_INT, bufferB, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Send(bufferB, position, MPI_PACKED, temprank, 5, MPI_COMM_WORLD);

        }
    } else {
        //for other processes receive from root rank
        //receive their own row/col start and end
        MPI_Recv(&row_start_point, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&row_end_point, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&col_start_point, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&col_end_point, 1, MPI_INT, 0, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        //preparation to receive the packed subarrays
        int totalElem = rowC*colC + (sizeof(unsigned long long) * 2);
        MPI_Pack_size(totalElem, MPI_INT, MPI_COMM_WORLD, &pack_size);
        int buffer_size = pack_size;
        char *recv_bufferA = (char *)malloc(buffer_size);
        char *recv_bufferB = (char *)malloc(buffer_size);


        if (!recv_bufferA || !recv_bufferB) {
            perror("Buffer allocation failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }


        //receive the subarrays using buffer
        MPI_Recv(recv_bufferA, buffer_size, MPI_PACKED, 0, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(recv_bufferB, buffer_size, MPI_PACKED, 0, 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        clock_gettime(CLOCK_MONOTONIC, &endComp);

        pMatrixA = (int *)malloc(totalElem * sizeof(int)); // Allocate memory for
pMatrixA
```

```
        pMatrixB = (int *)malloc(totalElem * sizeof(int)); // Allocate memory for
pMatrixB

        if (!pMatrixA || !pMatrixB) {
            perror("Memory allocation for pMatrixA failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }

        position = 0;
        // Unpack timestamp and subarrays
        MPI_Unpack(recv_bufferA, buffer_size, &position, &startComp.tv_sec, 1,
MPI_UINT64_T, MPI_COMM_WORLD);
        MPI_Unpack(recv_bufferA, buffer_size, &position, &startComp.tv_nsec, 1,
MPI_UINT64_T, MPI_COMM_WORLD);
        MPI_Unpack(recv_bufferA, buffer_size, &position, pMatrixA, (rowC*colC),
MPI_INT, MPI_COMM_WORLD);
        position = 0;
        MPI_Unpack(recv_bufferB, buffer_size, &position, pMatrixB, totalElem,
MPI_INT, MPI_COMM_WORLD);
        time_taken = (endComp.tv_sec - startComp.tv_sec) * 1e9;
        time_taken = (time_taken + (endComp.tv_nsec - startComp.tv_nsec)) * 1e-9;

        //print out the timetaken
        printf("Rank %d took %lf (s) to receive matrix from Root process\n",
my_rank, time_taken);

    }

    if(my_rank == 0) { //rank 0 follow the normal way as the matrix arrange same as
before
        row_start_point = 0;
        row_end_point = row_start_point + threadRowDiv;
        col_start_point = 0;
        col_end_point = threadColDiv;

        for (i = row_start_point; i < row_end_point; i++) {
            for (j = col_start_point; j < col_end_point; j++) {
                    pMatrixC[i * colC + j] = 0; // Initialize the result element
                for (k = 0; k < commonPoint; k++) {
                    pMatrixC[(i*colC)+j] += (pMatrixA[(i*colA)+k] *
pMatrixB[(k*colB)+j]);
                }
            }
        }
    } else{ //the others the other way as it is transposed

        for (i = row_start_point; i < row_end_point; i++) {
            for (j = col_start_point; j < col_end_point; j++) {
                pMatrixC[i * colC + j] = 0; // Initialize the result element
                for (k = 0; k < commonPoint; k++) {
```

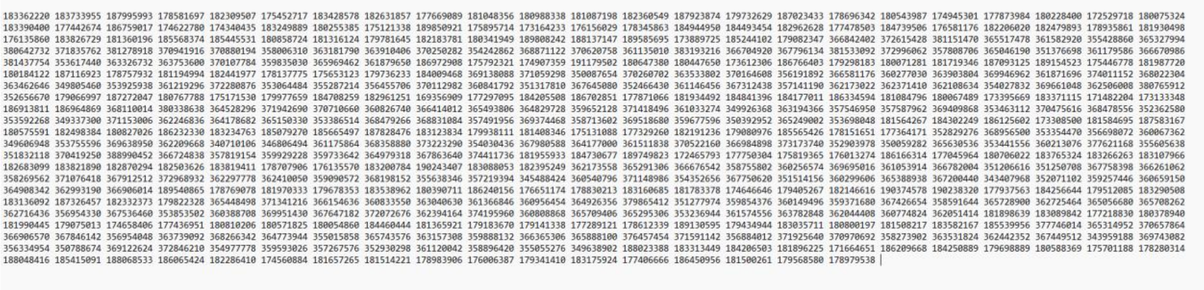
```
                    // Matrix A row remains the same
                    int elemA = pMatrixA[((i - row_start_point) * commonPoint) +
k];

                    // Matrix B column is treated as a row
                    int elemB = pMatrixB[((j - col_start_point) * commonPoint) +
k];

                    // Perform the multiplication
                    pMatrixC[(i * colC) + j] += elemA * elemB;
                }
            }
        }
    }

    unsigned long long *globalMatrixC = NULL;
    if(my_rank != 0){
        //send local pMatrixC back to root
        MPI_Pack_size(rowC*colC, MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD,
&pack_size);
        int buffer_size = pack_size;
        char *bufferC = (char *)malloc(buffer_size);
        position = 0;
        MPI_Pack(pMatrixC, rowC * colC, MPI_UNSIGNED_LONG_LONG, bufferC,
buffer_size, &position, MPI_COMM_WORLD);
        MPI_Send(bufferC, position, MPI_PACKED, 0, my_rank, MPI_COMM_WORLD);
        MPI_Send(&row_start_point, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&row_end_point, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&col_start_point, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Send(&col_end_point, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);

    }

    if (my_rank == 0){
        globalMatrixC = (unsigned long long *)calloc((rowC * colC), sizeof(unsigned
long long));
        if (!globalMatrixC) {
            perror("Memory allocation for globalMatrixC failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        //set all values to 0
        for(i = 0; i < rowC* colC; i++){
            globalMatrixC[i] = 0;
        }
        //combine the rank 0 matrixC to globalmatrixC
        combineLocalMatrices(pMatrixC, globalMatrixC, row_start_point,
row_end_point, col_start_point, col_end_point, colC);

        //loop through each rank and receive their local matrixC from each rank
        //then combine it to globalmatrixC
```

```
        for (i = 1; i < size; i++){


            MPI_Pack_size(rowC*colC, MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD,
&pack_size);
            int buffer_size = pack_size;
            char *recv_bufferC = (char *)malloc(buffer_size);

            //receive bufferC from each rank
            MPI_Recv(recv_bufferC, buffer_size, MPI_PACKED, i, i, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            position = 0;
            MPI_Unpack(recv_bufferC, buffer_size, &position, pMatrixC, rowC*colC,
MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD);
            MPI_Recv(&row_start_point, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            MPI_Recv(&row_end_point, 1, MPI_INT, i, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            MPI_Recv(&col_start_point, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            MPI_Recv(&col_end_point, 1, MPI_INT, i, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


            combineLocalMatrices(pMatrixC, globalMatrixC, row_start_point,
row_end_point, col_start_point, col_end_point, colC);

        }
    }
    //checking each rank's row/col start and end
    // printf("rank: %d, row_start: %d, row_end: %d, col_start: %d, col_end: %d",
my_rank, row_start_point, row_end_point, col_start_point, col_end_point);
    // fflush(stdout);

    //waiting for each rank to finish
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();

    if (my_rank == 0) {
        // Write result to file
        printf("Writing Matrix C - Start\n");
        FILE *pFileC = fopen("MC_500x500.bin", "wb");
        fwrite(&rowC, sizeof(int), 1, pFileC);
        fwrite(&colC, sizeof(int), 1, pFileC);
        //write row by row
        for (i = 0; i < rowC; i++) {
            fwrite(&globalMatrixC[i * colC], sizeof(unsigned long long), colC,
pFileC);
        }
```

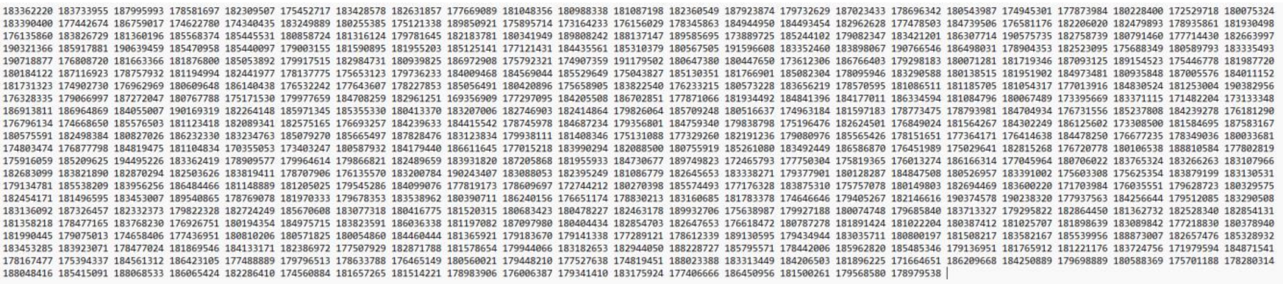
```c
        fclose(pFileC);
        printf("Writing Matrix C - Done\n");

        // Time calculation of overall program
        clock_gettime(CLOCK_MONOTONIC, &end);
        time_taken = (end.tv_sec - start.tv_sec) * 1e9;
        time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
        printf("Elapsed Time: %f seconds\n", time_taken);
    }

    return 0;
}
```

Check Correctness with 500x500 Matrix Mul

Serial code provided



Parallel code with MPI



Task 2 – Table & Analysis

Number of nodes (Specify 1 if only using your local computer): 1 **Number of CPU cores or logical processes per node: 1** **Number of MPI processes: 8** **Theoretical speed up: 8.466**					
Matrix Size	**500x500**	**1000x1000**	**2000x2000**	**3000x3000**	**4000x4000**
Serial Time	1.036289	7.205710	52.871768	180.674973	424.491226
Parallel Time	0.576	2.419	13.170	39.405	89.506
MPI Communication time	0.017	0.272	0.260	0.478	0.808
Speed Up	1.799	2.979	4.015	4.585	4.743

MONASH University

<u>**Compare and analysis**</u>

As the matrix size increases the speed up improves from 1.799 for 500x500 to 4.743 for 4000x4000. Larger matrix sizes allow more computational work to be distributed across the threads. The actual speed up is more than the theoretical speed up for the matrix that are 3000x3000 and above.

Why it achieves lower speed up compared to posix
Mpi processes it achieve lower speed up compared to posix. This is due to mpi has distributed memory, so it needs to send the matrix across the processes. Hence, the communication time is more than using posix as posix using shared memory which is faster.

When problem size increases
Beyond a certain point, increasing the matrix size doesn’t result in increase in speed up. For example, the jump from 3000x3000 to 4000x4000 shows only a slight increase in speed up due to the factors like increased communication time according to the table 4000x4000 communication time is 2 times of 3000x3000.

Why actual value is close to theoretical value
For larger matrices, the actual speed up close to the theoretical value because the parallel workload increases, reducing the impact of communication overhead.

500x500

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.002362 (s) to receive matrix from Root process
Rank 2 took 0.002507 (s) to receive matrix from Root process
Rank 3 took 0.002357 (s) to receive matrix from Root process
Rank 4 took 0.002539 (s) to receive matrix from Root process
Rank 5 took 0.002198 (s) to receive matrix from Root process
Rank 6 took 0.002804 (s) to receive matrix from Root process
Rank 7 took 0.002376 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 0.576258 seconds
```

1000x1000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.009935 (s) to receive matrix from Root process
Rank 2 took 0.021745 (s) to receive matrix from Root process
Rank 3 took 0.008968 (s) to receive matrix from Root process
Rank 4 took 0.008636 (s) to receive matrix from Root process
Rank 5 took 0.009761 (s) to receive matrix from Root process
Rank 6 took 0.009753 (s) to receive matrix from Root process
Rank 7 took 0.008842 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 2.419641 seconds
```

2000x2000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.025111 (s) to receive matrix from Root process
Rank 2 took 0.033228 (s) to receive matrix from Root process
Rank 3 took 0.032725 (s) to receive matrix from Root process
Rank 4 took 0.061699 (s) to receive matrix from Root process
Rank 5 took 0.032120 (s) to receive matrix from Root process
Rank 6 took 0.037845 (s) to receive matrix from Root process
Rank 7 took 0.037464 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 13.170289 seconds
```

3000x3000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.057411 (s) to receive matrix from Root process
Rank 2 took 0.050736 (s) to receive matrix from Root process
Rank 3 took 0.062953 (s) to receive matrix from Root process
Rank 4 took 0.066934 (s) to receive matrix from Root process
Rank 5 took 0.086156 (s) to receive matrix from Root process
Rank 6 took 0.078544 (s) to receive matrix from Root process
Rank 7 took 0.075797 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 39.404785 seconds
```

4000x4000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.084597 (s) to receive matrix from Root process
Rank 2 took 0.098275 (s) to receive matrix from Root process
Rank 3 took 0.119932 (s) to receive matrix from Root process
Rank 4 took 0.127551 (s) to receive matrix from Root process
Rank 5 took 0.117937 (s) to receive matrix from Root process
Rank 6 took 0.105058 (s) to receive matrix from Root process
Rank 7 took 0.154665 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 89.505719 seconds
```

Task 3 – Code

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <mpi.h>
#include <pthread.h>
#define NUM_THREADS 2

int row_start_point, row_end_point;
int col_start_point, col_end_point;
unsigned long long *pMatrixC;
int rowA, rowB, rowC, colA, colB, colC, commonPoint;
int *pMatrixA = NULL, *pMatrixB = NULL;
int my_rank, size;
pthread_mutex_t mutex;
unsigned long long *globalMatrixC = NULL;

//combine local matrices to the global matrix
void combineLocalMatrices(unsigned long long *localMatrix, unsigned long long
*globalMatrix, int row_start, int row_end, int col_start, int col_end, int colC) {
    //only loop through the necessary row and column for the local matrix
    //+= is because each of the elem in global matrix initially is 0, so x+0 = x
    for(int row = row_start; row < row_end; row++){
        for (int col = col_start; col < col_end; col++){
            // Add localMatrix to globalMatrix element by element
            globalMatrix[row * colC + col] += localMatrix[row * colC + col];
        }
    }
}

void *ThreadReadMatrixA(void *arg) {
    //assign row/col start end for each thread to read
    int thread_id = *((int *)arg);
    int rows_per_thread = rowA / NUM_THREADS;
    int start_row = thread_id * rows_per_thread;
    int end_row = (thread_id == NUM_THREADS - 1) ? rowA : start_row +
rows_per_thread;

    FILE *pFileA = fopen("MA_500x500.bin", "rb");
    if (!pFileA) {
        perror("File A opening failed");
        pthread_exit(NULL);
    }
    //loop through and copy to pmatrixA
    fseek(pFileA, sizeof(int) * 2 + sizeof(int) * start_row * colA, SEEK_SET); //
Seek to the appropriate row
    for (int i = start_row; i < end_row; i++) {
        fread(&pMatrixA[i * colA], sizeof(int), colA, pFileA);
```

```
    }
    fclose(pFileA);

    pthread_exit(NULL);
}

void *ThreadReadMatrixB(void *arg) {
    //assign each threads row/col to start and end to read
    int thread_id = *((int *)arg);
    int cols_per_thread = colB / NUM_THREADS;
    int start_col = thread_id * cols_per_thread;
    int end_col = (thread_id == NUM_THREADS - 1) ? colB : start_col +
cols_per_thread;

    FILE *pFileB = fopen("MB_500x500.bin", "rb");
    if (!pFileB) {
        perror("File B opening failed");
        pthread_exit(NULL);
    }

    fseek(pFileB, sizeof(int) * 2, SEEK_SET); // Skip the matrix dimensions

    for (int i = 0; i < rowB; i++) {
        //loop through to read and put in matrix B
        int *row_buffer = (int *)malloc(sizeof(int) * colB); // Temporary buffer
for the row
        fread(row_buffer, sizeof(int), colB, pFileB);        // Read the entire row
into buffer
        memcpy(&pMatrixB[i * colB + start_col], &row_buffer[start_col], sizeof(int)
* (end_col - start_col));  // Copy only the required columns
        free(row_buffer);
    }

    fclose(pFileB);
    pthread_exit(NULL);
}

void *ThreadWriteMatrixC(void *arg) {
    //use multiple thread to write to matrixC file
    int thread_id = *((int *)arg);
    int rows_per_thread = rowC / NUM_THREADS;
    int start_row = thread_id * rows_per_thread;
    int end_row = (thread_id == NUM_THREADS - 1) ? rowC : start_row +
rows_per_thread;

    // Use write mode for a fresh file
    FILE *pFileC = fopen("MC_500x500.bin", "rb+");
    if (!pFileC) {
        perror("File C opening failed");
        pthread_exit(NULL);
```

```
    }

    // Synchronize file writing to prevent race conditions
    pthread_mutex_lock(&mutex);
    //write to globalmatrixC
    fseek(pFileC, sizeof(int) * 2 + sizeof(unsigned long long) * start_row * colC,
SEEK_SET); // Seek to the appropriate row
    for (int i = start_row; i < end_row; i++) {
        fwrite(&globalMatrixC[i * colC], sizeof(unsigned long long), colC, pFileC);
    }

    pthread_mutex_unlock(&mutex);//unlock after done for other threads to write

    fclose(pFileC);
    pthread_exit(NULL);
}

void *ThreadComp(void *pArg){
    int i, j, k;
    int threads_num = *((int*)pArg);
    //distribute the rows and cols to each thread
    int row_per_thread = (row_end_point - row_start_point) / NUM_THREADS;
    int rptr = (row_end_point - row_start_point) % NUM_THREADS;
    int start_point = row_start_point + threads_num * row_per_thread;
    int end_point = start_point + row_per_thread;

    if(threads_num == NUM_THREADS - 1)
        end_point += rptr;

    if (my_rank == 0){//rank 0 follow the normal way as the matrix arrange same as
before
        for (i = start_point; i < end_point ; i++) {
                for (j = col_start_point; j < col_end_point; j++) {
                        pMatrixC[i * colC + j] = 0; // Initialize the result
element
                    for (k = 0; k < commonPoint; k++) {
                        pMatrixC[(i*colC)+j] += (pMatrixA[(i*colA)+k] *
pMatrixB[(k*colB)+j]);
                    }
                }
            }

    } else{//the other ranks diff as matrix b is transposed
        for (i = start_point; i < end_point; i++) {
            for (j = col_start_point; j < col_end_point; j++) {
                pMatrixC[i * colC + j] = 0; // Initialize the result element
                for (k = 0; k < commonPoint; k++) {
                    // Matrix A row remains the same
                    int elemA = pMatrixA[((i - row_start_point) * commonPoint) +
k];
```

```
                    // Matrix B column is treated as a row
                    int elemB = pMatrixB[((j - col_start_point) * commonPoint) +
k];

                    // Perform the multiplication
                    pMatrixC[(i * colC) + j] += elemA * elemB;

                }
            }
        }
    }
    pthread_exit(NULL);
}


int main(int argc, char *argv[])
{
    // Variables
    int i = 0, j = 0, k = 0;

    int position;
    int pack_size;

    //initialise mutex
    pthread_mutex_init(&mutex, NULL);
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];


    /* Clock information */
    struct timespec start, end, startComp, endComp;
    double time_taken;

    clock_gettime(CLOCK_MONOTONIC, &start);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);



    // 1. Read Matrix A
    rowA = colA = rowB = colB = 0;
    if (my_rank == 0) {
        printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

        // Reading Matrix A
        printf("Reading Matrix A - Start\n");
        FILE *pFileA = fopen("MA_500x500.bin", "rb");
```

```c
    if (!pFileA) {
        perror("File A opening failed");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    fread(&rowA, sizeof(int), 1, pFileA);
    fread(&colA, sizeof(int), 1, pFileA);
    pMatrixA = (int *)malloc((rowA * colA) * sizeof(int));
    if (pMatrixA == NULL) {
        perror("Memory allocation for Matrix A failed");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

   // Create threads to read Matrix A
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, ThreadReadMatrixA, &thread_ids[i]);
    }

    // Join threads after reading Matrix A
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Reading Matrix A - Done\n");

    // Reading Matrix B
    printf("Reading Matrix B - Start\n");
    FILE *pFileB = fopen("MB_500x500.bin", "rb");
    if (!pFileB) {
        perror("File B opening failed");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    fread(&rowB, sizeof(int), 1, pFileB);
    fread(&colB, sizeof(int), 1, pFileB);
    pMatrixB = (int *)malloc((rowB * colB) * sizeof(int));
    if (pMatrixB == NULL) {
        perror("Memory allocation for Matrix B failed");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    // Create threads to read Matrix B
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, ThreadReadMatrixB, &thread_ids[i]);
    }

    // Join threads after reading Matrix B
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
        printf("Reading Matrix B - Done\n");

        // Initialize Matrix C
        rowC = rowA;
        colC = colB;
        commonPoint = colA; // Initialize commonPoint
    }
    //broadcast rows to all the processses
    MPI_Bcast(&rowA, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colA, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&rowB, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colB, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&rowC, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&colC, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&commonPoint, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast
commonPoint

    //assign local matrix result
    pMatrixC = (unsigned long long *)calloc((rowC * colC), sizeof(unsigned long
long));

    //calculate the tile partition
    //using result will divide into half even number process left odd number
process right
    int threadColDiv = colC / 2;
    int threadColDivRemain = colC % 2;
    int threadRowDiv = rowC / (size / 2);
    int threadRowDivRemain = rowC % (size / 2);

    //root rank calculate the subarray for matrixA and matrixB
    //send it to the corresponding rank
    if (my_rank == 0) {
        for (int temprank = 1; temprank < size; temprank++) {
            position = 0;
            if (temprank % 2 == 0) {
                // Even thread
                if (temprank == (size - 2)) {
                    // Last even thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv +
threadRowDivRemain;
                } else {
                    // Not last even thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv;
                }
                col_start_point = 0;
                col_end_point = threadColDiv;
```

```
            } else {
                // Odd thread
                if (temprank == (size - 1)) {
                    // Last odd thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv +
threadRowDivRemain;
                } else {
                    // Not last odd thread
                    row_start_point = (temprank / 2) * threadRowDiv;
                    row_end_point = row_start_point + threadRowDiv;
                }
                col_start_point = threadColDiv;
                col_end_point = col_start_point + threadColDiv +
threadColDivRemain;
            }
            //send row/col start and end point to each rank
            MPI_Send(&row_start_point, 1, MPI_INT, temprank, 0, MPI_COMM_WORLD);
            MPI_Send(&row_end_point, 1, MPI_INT, temprank, 1, MPI_COMM_WORLD);
            MPI_Send(&col_start_point, 1, MPI_INT, temprank, 2, MPI_COMM_WORLD);
            MPI_Send(&col_end_point, 1, MPI_INT, temprank, 3, MPI_COMM_WORLD);

            int subarray_size = rowC*colC;
            int *subarrayA = (int *)malloc(subarray_size * sizeof(int));
            int *subarrayB = (int *)malloc(subarray_size * sizeof(int));
            if (!subarrayA || !subarrayB) {
                perror("Subarray allocation failed");
                MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
            }

            //copy necessary cells to subarrayA
            for (i = row_start_point; i < row_end_point; i++) {
                for (k = 0; k < commonPoint; k++){
                    subarrayA[((i - row_start_point) * colA) + k] = pMatrixA[(i *
colA) + k];
                }
            }
            //copy ncessary cells to subarrayB for matrix multiplication
            for(j = col_start_point; j < col_end_point; j++){
                for (k = 0; k < commonPoint; k++) {
                    subarrayB[((j - col_start_point) * rowA) + k] = pMatrixB[(k *
colB) + j];
                }
            }

            //calculate the packsize of subarrays
            MPI_Pack_size(subarray_size + (sizeof(unsigned long long) * 2),
MPI_INT, MPI_COMM_WORLD, &pack_size);
            int buffer_size = pack_size;
            char *bufferA = (char *)malloc(buffer_size);
```

```
            char *bufferB = (char *)malloc(buffer_size);
            clock_gettime(CLOCK_MONOTONIC, &startComp);

            //pack the time and subarrays to send to the corresponding rank
            MPI_Pack(&startComp.tv_sec, 1, MPI_UINT64_T, bufferA, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Pack(&startComp.tv_nsec, 1, MPI_UINT64_T, bufferA, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Pack(subarrayA, subarray_size, MPI_INT, bufferA, buffer_size,
&position, MPI_COMM_WORLD);

            MPI_Send(bufferA, position, MPI_PACKED, temprank, 4, MPI_COMM_WORLD);

            position = 0;
            MPI_Pack(subarrayB, subarray_size, MPI_INT, bufferB, buffer_size,
&position, MPI_COMM_WORLD);
            MPI_Send(bufferB, position, MPI_PACKED, temprank, 5, MPI_COMM_WORLD);


        }
    } else {
        //recv row/col start n end point from root rank
        MPI_Recv(&row_start_point, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&row_end_point, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&col_start_point, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&col_end_point, 1, MPI_INT, 0, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        //preparation to receive submatrix from root rank
        int totalElem = rowC*colC + (sizeof(unsigned long long) * 2);
        MPI_Pack_size(totalElem, MPI_INT, MPI_COMM_WORLD, &pack_size);
        int buffer_size = pack_size;
        char *recv_bufferA = (char *)malloc(buffer_size);
        char *recv_bufferB = (char *)malloc(buffer_size);


        if (!recv_bufferA || !recv_bufferB) {
            perror("Buffer allocation failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }


        //receive subarray
        MPI_Recv(recv_bufferA, buffer_size, MPI_PACKED, 0, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(recv_bufferB, buffer_size, MPI_PACKED, 0, 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
        clock_gettime(CLOCK_MONOTONIC, &endComp);

        pMatrixA = (int *)malloc(totalElem * sizeof(int)); // Allocate memory for
pMatrixA
        pMatrixB = (int *)malloc(totalElem * sizeof(int)); // Allocate memory for
pMatrixB

        if (!pMatrixA || !pMatrixB) {
            perror("Memory allocation for pMatrixA failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }

        position = 0;
        // Unpack timestamp and matrix and store in local pmatrix
        MPI_Unpack(recv_bufferA, buffer_size, &position, &startComp.tv_sec, 1,
MPI_UINT64_T, MPI_COMM_WORLD);
        MPI_Unpack(recv_bufferA, buffer_size, &position, &startComp.tv_nsec, 1,
MPI_UINT64_T, MPI_COMM_WORLD);
        MPI_Unpack(recv_bufferA, buffer_size, &position, pMatrixA, (rowC*colC),
MPI_INT, MPI_COMM_WORLD);
        position = 0;
        MPI_Unpack(recv_bufferB, buffer_size, &position, pMatrixB, totalElem,
MPI_INT, MPI_COMM_WORLD);
        time_taken = (endComp.tv_sec - startComp.tv_sec) * 1e9;
        time_taken = (time_taken + (endComp.tv_nsec - startComp.tv_nsec)) * 1e-9;

        //time taken for ranks to receive from the root rank
        printf("Rank %d took %lf (s) to receive matrix from Root process\n",
my_rank, time_taken);

    }

    if(my_rank == 0) { //rank 0 default setting values
        row_start_point = 0;
        row_end_point = row_start_point + threadRowDiv;
        col_start_point = 0;
        col_end_point = threadColDiv;
    }

    //create threads to calculate their tile results
    for (i = 0; i < NUM_THREADS; i++){
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, ThreadComp, &thread_ids[i]);
    }
    //join after calculating
    for (i = 0; i< NUM_THREADS; i++){
        pthread_join(threads[i], NULL);
    }
```

```
    if(my_rank != 0){
        //send local pMatrixC back to root
        MPI_Pack_size(rowC*colC, MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD,
&pack_size);
        int buffer_size = pack_size;
        char *bufferC = (char *)malloc(buffer_size);
        position = 0;
        MPI_Pack(pMatrixC, rowC * colC, MPI_UNSIGNED_LONG_LONG, bufferC,
buffer_size, &position, MPI_COMM_WORLD);
        MPI_Send(bufferC, position, MPI_PACKED, 0, my_rank, MPI_COMM_WORLD);
        MPI_Send(&row_start_point, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&row_end_point, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&col_start_point, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Send(&col_end_point, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);

    }

    if (my_rank == 0){
        globalMatrixC = (unsigned long long *)calloc((rowC * colC), sizeof(unsigned
long long));
        if (!globalMatrixC) {
            perror("Memory allocation for globalMatrixC failed");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        //set all values to 0
        for(i = 0; i < rowC* colC; i++){
            globalMatrixC[i] = 0;
        }
        //combine root rank local matrixC to globalmatrixC
        combineLocalMatrices(pMatrixC, globalMatrixC, row_start_point,
row_end_point, col_start_point, col_end_point, colC);

       //loop through each rank and receive their local matrixC from each rank
        //then combine it to globalmatrixC
        for (i = 1; i < size; i++){
            MPI_Pack_size(rowC*colC, MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD,
&pack_size);
            int buffer_size = pack_size;
            char *recv_bufferC = (char *)malloc(buffer_size);
            //receive and unpack local matrix C that receive from other ranks
            MPI_Recv(recv_bufferC, buffer_size, MPI_PACKED, i, i, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            position = 0;
            MPI_Unpack(recv_bufferC, buffer_size, &position, pMatrixC, rowC*colC,
MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD);
            MPI_Recv(&row_start_point, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
            MPI_Recv(&row_end_point, 1, MPI_INT, i, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            MPI_Recv(&col_start_point, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            MPI_Recv(&col_end_point, 1, MPI_INT, i, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            //combine it with globalmatrixC
            combineLocalMatrices(pMatrixC, globalMatrixC, row_start_point,
row_end_point, col_start_point, col_end_point, colC);
        }
    }


    //wait for all ranks
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();

    if (my_rank == 0) {
        // Write result to file
        printf("Writing Matrix C - Start\n");
        FILE *pFileC = fopen("MC_500x500.bin", "wb");
        fwrite(&rowC, sizeof(int), 1, pFileC);
        fwrite(&colC, sizeof(int), 1, pFileC);


        // Create threads to write Matrix C
        for (int i = 0; i < NUM_THREADS; i++) {
            thread_ids[i] = i;
            pthread_create(&threads[i], NULL, ThreadWriteMatrixC, &thread_ids[i]);
        }

        // Join threads after writing Matrix C
        for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }



        fclose(pFileC);
        printf("Writing Matrix C - Done\n");

        // Time calculation of overall program
        clock_gettime(CLOCK_MONOTONIC, &end);
        time_taken = (end.tv_sec - start.tv_sec) * 1e9;
        time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
        printf("Elapsed Time: %f seconds\n", time_taken);
    }

    return 0;
```

```
}
```

Check correctness with Matrix Mul

500x500

Provided Matrix Mul

```
183362220 183733955 187995993 178581697 182309507 175452717 183428578 182631857 177669089 181048356 180988338 181087198 182360549 187923874 179732629 187023433 178696342 180543987 174945301 177873984 180228400 172529718 180075324
183390400 177442674 186759017 174622780 174340435 183249889 180255385 175121338 189850921 175895714 173164233 176156029 178345863 184944950 184493454 182962628 177478503 184739506 176581176 182206020 182479893 178935861 181930498
176135860 183826729 181360196 185568374 185445531 180858724 181316124 179781645 182183781 180341949 189808242 188137147 189585695 173889725 185244102 179082347 183421201 186307714 190575735 182758739 180791460 177714430 182663997
190321366 185917881 190639459 185470958 185440097 179003155 181590895 181955203 185125141 177121431 184435561 185310379 180567505 191596608 183352460 183898067 190766546 186498031 178904353 182523095 175688349 180589793 183335493
190718877 176808720 181663366 181876800 185053892 179917515 182984731 180939825 186972908 175792321 174907359 191179502 180647380 180447650 173612306 186766403 179298183 180071281 181719346 187093125 189154523 175446778 181987720
180184122 187116923 178757932 181194994 182441977 178137775 175653123 179736233 184009468 184569044 185529649 175043827 185130351 181766901 185082304 178095946 183290588 180138515 181951902 184973481 180935848 187005576 184011152
181731323 174902730 176962969 180609648 186140438 176532242 177643607 178227853 185056491 180420896 175658905 183822540 176233215 180573228 183656219 178570595 181086511 181185705 181054317 177013916 184830524 181253004 190382956
176328335 179066997 187272047 180767788 175171530 179977659 184708259 182961251 169356909 177297095 184205508 186702851 177871066 181934492 184841396 184177011 186334594 181084796 180067489 173395669 183371115 171482204 173133348
186913811 186964869 184055007 190169319 182264148 185971345 185355330 180413370 183207006 182746903 182414864 179826064 185709248 180516637 174963184 181597183 178773475 178793981 184704934 176731556 185237808 184239278 176181290
176796134 174668650 185576503 181123418 182089341 182575165 176693257 184239633 184415542 178745978 184687234 179356801 184759340 179838798 175196476 182624501 176849024 181564267 184302249 186125602 173308500 181584695 187583167
180575591 182498384 180827026 186232330 183234763 185079270 185665497 187828476 183123834 179938111 181408346 175131088 177329260 182191236 179080976 185565426 178151651 177364171 176414638 184478250 176677235 178349036 180033681
174803474 176877798 184819475 181104834 170355053 173403247 180587932 184179440 186611645 177015218 183990294 182088500 180755919 185261080 183492449 186586870 176451989 175029641 182815268 176720778 180106538 188810584 177802819
175916059 185209625 194495226 183362419 178909577 179964614 179866821 182489659 183931820 187205868 181955933 184730677 189749823 172465793 177750304 175819365 176013274 186166314 177045964 180706022 183765324 183266263 183107966
182683099 183821890 182870294 182503626 183819411 178707906 176135570 183200784 190243407 183088053 182395249 181086779 182645653 183338271 179377901 180128287 184847508 180526957 183391002 175603308 175625354 183879199 183130531
179134781 185538209 183956256 186484466 181148889 181205025 179545286 184099076 177819173 178609697 172744212 180270398 185574493 177176328 183875310 175757078 180149803 182694469 183600220 171703984 176035551 179628723 180329575
182454171 181496595 183453007 189540865 178769078 181970333 179678353 183538962 180390711 186240156 176651174 178830213 183160685 181783378 174646646 179405267 182146616 190374578 190238320 177937563 184256644 179512085 183290508
183136092 187326457 182332373 179822328 182724249 185670608 183077318 180416775 181520315 180683423 180478227 182463178 189932706 175638987 179927188 180074748 179685840 183713327 179295822 182864450 181362732 182528340 182854131
181358218 178477165 183768230 176926751 180194354 184975715 183823591 186036338 181197082 187097980 180404434 182854703 182647653 176618472 180787278 181891424 181022204 180387412 181025707 181898639 183089842 177218830 180378940
181990445 179075013 174658406 177436951 180810206 180571825 180054860 184460444 181365921 179183670 179141338 177289121 178612339 189130595 179434944 183035711 180800197 181508217 183582167 185539956 188873007 182657476 185328932
183453285 183923071 178477024 181869546 184133171 182386972 177507929 182871788 181578654 179944066 183182653 182944050 188228727 185795571 178442006 185962820 185485346 179136951 181765912 181221176 183724756 171979594 184871541
178167477 175394337 184561312 186423105 177488889 179796513 178633788 176465149 180560021 179448210 177527638 174819451 188023388 183313449 184206503 181896225 171664651 186209668 184250889 179698889 180588369 175701188 178280314
188048416 185415091 188068533 186065424 182286410 174560884 181657265 181514221 178983906 176006387 179341410 183175924 177406666 186450956 181500261 179568580 178979538
```

Task 3 MPI with threads

```
183362220 183733955 187995993 178581697 182309507 175452717 183428578 182631857 177669089 181048356 180988338 181087198 182360549 187923874 179732629 187023433 178696342 180543987 174945301 177873984 180228400 172529718 180075324
183390400 177442674 186759017 174622780 174340435 183249889 180255385 175121338 189850921 175895714 173164233 176156029 178345863 184944950 184493454 182962628 177478503 184739506 176581176 182206020 182479893 178935861 181930498
176135860 183826729 181360196 185568374 185445531 180858724 181316124 179781645 182183781 180341949 189808242 188137147 189585695 173889725 185244102 179082347 366842402 372615428 190575735 182758739 180791460 177714430 182663997
190321366 185917881 190639459 185470958 185440097 179003155 181590895 181955203 185125141 177121431 184435561 185310379 180567505 191596608 183352460 183898067 190766546 186498031 178904353 182523095 175688349 180589793 183335493
190718877 176808720 181663366 181876800 185053892 179917515 182984731 180939825 186972908 175792321 174907359 191179502 180647380 180447650 173612306 186766403 179298183 180071281 181719346 187093125 189154523 175446778 181987720
180184122 187116923 178757932 181194994 182441977 178137775 175653123 179736233 184009468 369138088 371059298 175043827 185130351 181766901 185082304 178095946 183290588 180138515 181951902 184973481 180935848 187005576 184011152
181731323 174902730 176962969 180609648 186140438 176532242 177643607 178227853 185056491 180420896 175658905 183822540 176233215 180573228 183656219 178570595 181086511 181185705 181054317 177013916 184830524 181253004 190382956
176328335 179066997 187272047 180767788 175171530 179977659 184708259 182961251 169356909 177297095 184205508 186702851 177871066 181934492 184841396 184177011 186334594 181084796 180067489 173395669 183371115 171482204 173133348
186913811 186964869 368110014 380338638 182264148 185971345 185355330 180413370 183207006 182746903 182414864 179826064 185709248 180516637 174963184 181597183 178773475 178793981 184704934 176731556 185237808 184239278 176181290
176796134 174668650 185576503 181123418 182089341 182575165 176693257 184239633 184415542 178745978 184687234 179356801 184759340 179838798 175196476 182624501 176849024 181564267 184302249 186125602 173308500 181584695 187583167
180575591 182498384 180827026 186232330 183234763 185079270 185665497 187828476 183123834 179938111 181408346 175131088 177329260 182191236 179080976 185565426 178151651 177364171 352829276 368956500 176677235 178349036 180033681
174803474 176877798 184819475 181104834 170355053 173403247 180587932 184179440 186611645 177015218 183990294 182088500 180755919 185261080 183492449 186586870 176451989 175029641 182815268 176720778 180106538 188810584 177802819
175916059 185209625 194495226 183362419 178909577 179964614 179866821 182489659 183931820 187205868 181955933 184730677 189749823 172465793 177750304 175819365 176013274 186166314 177045964 180706022 183765324 183266263 183107966
182683099 183821890 182870294 182503626 183819411 178707906 176135570 183200784 190243407 183088053 182395249 362173558 365291306 183338271 179377901 180128287 184847508 180526957 183391002 175603308 175625354 183879199 183130531
179134781 185538209 183956256 186484466 181148889 181205025 179545286 184099076 177819173 178609697 172744212 180270398 185574493 177176328 183875310 175757078 180149803 182694469 183600220 171703984 176035551 179628723 180329575
182454171 181496595 183453007 189540865 178769078 181970333 179678353 183538962 180390711 186240156 176651174 178830213 183160685 181783378 174646646 179405267 182146616 190374578 190238320 177937563 184256644 179512085 183290508
183136092 187326457 182332373 179822328 365448498 371341216 183077318 180416775 181520315 180683423 180478227 182463178 189932706 175638987 179927188 180074748 179685840 183713327 179295822 182864450 181362732 182528340 182854131
181358218 178477165 183768230 176926751 180194354 184975715 183823591 186036338 181197082 187097980 180404434 182854703 182647653 176618472 180787278 181891424 181022204 180387412 181025707 181898639 183089842 177218830 180378940
181990445 179075013 174658406 177436951 180810206 180571825 180054860 184460444 181365921 179183670 179141338 177289121 178612339 189130595 179434944 183035711 180800197 181508217 183582167 185539956 377746014 365314952 185328932
183453285 183923071 178477024 181869546 184133171 182386972 177507929 182871788 181578654 179944066 183182653 182944050 188228727 185795571 178442006 185962820 185485346 179136951 181765912 181221176 183724756 171979594 184871541
178167477 175394337 184561312 186423105 177488889 179796513 178633788 176465149 180560021 179448210 177527638 174819451 188023388 183313449 184206503 181896225 171664651 186209668 184250889 179698889 180588369 175701188 178280314
188048416 185415091 188068533 186065424 182286410 174560884 181657265 181514221 178983906 176006387 179341410 183175924 177406666 186450956 181500261 179568580 178979538
```

Task 3 – Table & Analysis

Number of nodes (Specify 1 if only using your local computer): 1
Number of CPU cores or logical processes per node: 8
Number of MPI processes: 4
Number of threads per MPI process: 2
Theoretical speed up: 8.466

Matrix Size	500x500	1000x1000	2000x2000	3000x3000	4000x4000
Serial Time	1.036289	7.205710	52.871768	180.674973	424.491226
Task 3 Parallel Time	0.557	2.146	12.208	39.505	89.257
Speed Up	1.860	3.358	4.331	4.573	4.755
MPI communication time	0.006868	0.02038	0.074234	0.1633	0.230557
Task 2 Parallel Time	0.576	2.419	13.170	39.405	89.506

MONASH University

Compare and analysis

The communication time in table 3 is significantly lower than table 2. This results in better speed up in table 3 compared to table 2. The reasons of why task 3 achieve a better speed up:

1) **Hybrid approach**
   Task 3 used hybrid approach which is 4 processes and 2 threads per processes, this result in smaller communication time in task 3. Thus, reduces the overhead of message passing, allowing for better parallel performance. By distributing task to MPI and each process will have their own threads to distribute the workload again, this lower down the communication time and increase the computational power as threads using shared memory so they can compute faster compared to MPI.

2) **Decrease communication time**
   The speed up in task 3 is slightly higher than task 2. This indicates better performance when using 4 MPI processes with 2 threads per process. This is due to lower communication overhead compared to 8 mpi processes according to the table above by using hybrid approach, it significantly lower down the MPI communication time by 2 times. In Task 3, each MPI process is handling more data internally, which reduces the need for frequent interprocess communication whereas in task 2 root rank needs to distribute tasks to 8 processes which increase in communication time. Additionally, since there are 2 threads per MPI process so the work can be distributed within each process, so they don’t need to exchange much data between processes.

Actual speed up is close to theoretical speed up
Furthermore, when the problem size increases the speed up increase as well. The speed up of task 3 began to more than the theoretical speed up when the matrix size is 3000x3000 and above. This is due to for larger matrices the parallel workload increases. Hence, increase in the speed up.

500x500

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.002231 (s) to receive matrix from Root process
Rank 2 took 0.002298 (s) to receive matrix from Root process
Rank 3 took 0.002339 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 0.557018 seconds
```

1000x1000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.006462 (s) to receive matrix from Root process
Rank 2 took 0.006381 (s) to receive matrix from Root process
Rank 3 took 0.007537 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 2.146042 seconds
```

2000x2000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.023296 (s) to receive matrix from Root process
Rank 2 took 0.024509 (s) to receive matrix from Root process
Rank 3 took 0.026429 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 12.207574 seconds
```

3000x3000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.052840 (s) to receive matrix from Root process
Rank 2 took 0.056190 (s) to receive matrix from Root process
Rank 3 took 0.054269 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 39.504677 seconds
```

4000x4000

```
Matrix Multiplication using 1-Dimension Arrays - Start

Reading Matrix A - Start
Reading Matrix A - Done
Reading Matrix B - Start
Reading Matrix B - Done
Rank 1 took 0.073501 (s) to receive matrix from Root process
Rank 2 took 0.076708 (s) to receive matrix from Root process
Rank 3 took 0.080348 (s) to receive matrix from Root process
Writing Matrix C - Start
Writing Matrix C - Done
Elapsed Time: 89.257441 seconds
```

AI Acknowledgment Statement

I would like to acknowledge the contributions of artificial intelligence technologies (ChatGPT) that have supported my work.