并行分布式计算

课程报告

姓名：林睿

学号：21307130150

1. 任务描述

1.1 选题

使用 OpenMP 编程实现并行快速排序(PQS)算法

使用 MPI 实现 Parallel sorting by regular sampling(PSRS)算法

选做部分：MPI 与 CUDA 异构实现矩阵乘法与 Megatron

1.2 具体任务细节

在本实验的必做部分中主要实现了两个 C++程序，分别使用 OpenMP 框架编程实现并行快速排序算法和使用 MPI 实现并行正则采样排序算法。程序分别通过命令行接收两个参数<desired_num_threads> 和 <desire_size_of_array>来对并发进程数和待排序数组大小进行控制，并在程序中包含了对错误命令行参数的处理、给定数组大小的随机数组生成、排序算法具体实现、打印结果、结果比较与验证（和串行快速排序）、性能度量（以串行的快速排序作为基准）等具体任务细节。

1.3 实验环境

Ubuntu 2 Linux(wsl)环境

2. 算法逻辑

2.1 并行快速排序算法

2.1.1 快速排序算法

快速排序算法的基本流程如下：

选择基准元素：从数组中选择一个元素作为基准（通常是数组中间的元素）。

划分阶段：将数组中的元素分为两部分，小于基准的元素放在左侧，大于基准的元素放在右侧。

递归阶段：对左右两个部分递归地应用快速排序。

合并阶段：在递归的过程中，数组逐渐变得有序。

2.1.2 并行化处理

在并行快速排序中，可以通过以下方式实现并行化：

任务分配：将划分和递归的过程分解为任务，使用多个进程同时处理这些任务。

并行递归：对数组划分后，左右两个部分的排序可以并行进行，从而加速整体排序过程。

阈值处理：为了避免过多的进程创建和管理开销，可以设置一个阈值，当子数组的大小小于阈值时，使用串行排序。（阈值为所设置的进程数）

通过这些并行化措施，可以在多核处理器上更有效地利用资源，加速排序过程。

2.2 并行正则采样排序算法

2.2.1 正则采样排序算法

正则采样排序算法的基本流程如下：

无序序列的划分及局部排序：根据数据块的划分方法，将无序序列划分成 p(p 为处理器或进程数)部分，每个处理器对其中的一部分进行串行快速排序，这样每个处理器就会拥有一个局部有序序列。

正则采样：每个处理器从局部有序序列中选取第 w,2w,...,(p-1)w 共 p-1 个代表元素。其中 w = n/p^2。

确定主元：每个处理器都将自己选取好的代表元素发送给处理器 p0。p0 对这 p 段有序序列做多路归并排序，再从这排序后的序列中选取第 p-1,2(p-1), ...,(p-1)(p-1)共 p-1 个元素作为主元。

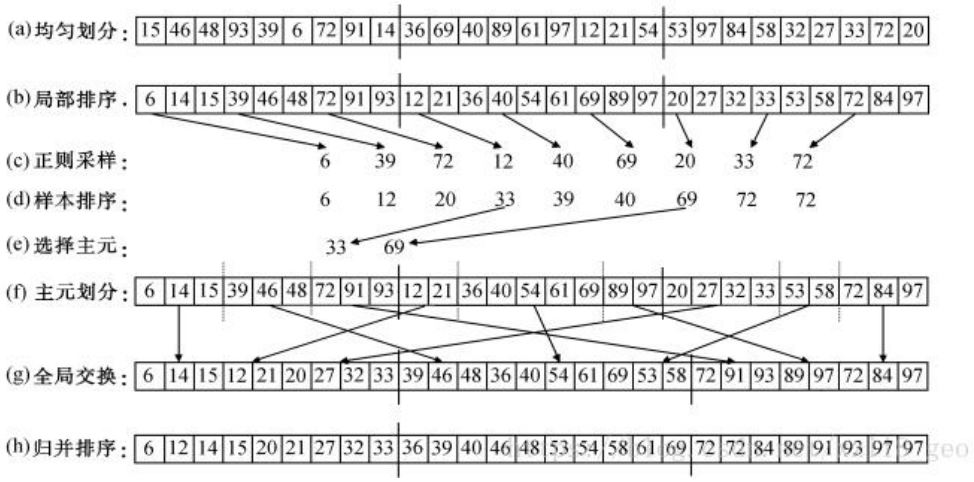
分发主元：将这 p-1 个主元分发给各个处理器。

局部有序序列划分：每个处理器在接收到主元后，根据主元将自己的局部有序序列划分成 p 段。

p 段有序序列的分发：每个处理器将自己的第 i 段发送给第 i 个处理器，是处理器 i 都拥有所有处理器的第 i 段。

多路排序：每个处理器将上一步得到的 p 段有序序列做多路归并。

经过这 7 步后，一次将每个处理器的数据取出，这些数据是有序的。示例图如下：



2.2.2 并行化处理

在正则采样排序中，可以通过以下方式实现并行化：

无序序列的划分及局部排序中，将无序序列划分成为 p 部分，并由每个处理器并行地对其负责的部分进行串行快速排序并一定间隔地选取样本。局部有序序列划分时也可以并行进行。p 段有序序列的分发可以在每个处理器之间进行通讯，这个步骤也可以并行。最终的多路归并排序也可以并行。

3. 编程代码实现

3.1 并行快速排序算法(OpenMP)

首先定义元素交换、划分的方法

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

在快速排序实现的代码部分，将对两个子段的并行快速排序递归调用分配成两个任务（OpenMP 调用），并且定义了一个 cutoff 值，如果此段数组的长度小于 cutoff 值（后面将其设为 100），则直接使用串行化代码以提高速度并且防止无止境的递归调用。在 quickSort_parallel 中创建一个进程执行内部的程序并且等待由当前进程创建的所有并行任务执行完。

```
void quickSort_parallel_internal(int arr[], int low, int high, int cutoff) {
    if (low < high) {
        int pi = partition(arr, low, high);

        if (high - low < cutoff) {
            quickSort_parallel_internal(arr, low, pi - 1, cutoff);
            quickSort_parallel_internal(arr, pi + 1, high, cutoff);
        } else {
#pragma omp task
            {
                quickSort_parallel_internal(arr, low, pi - 1, cutoff);
            }
#pragma omp task
            {
                quickSort_parallel_internal(arr, pi + 1, high, cutoff);
            }
        }
    }
}
```

```
void quickSort_parallel(int arr[], int low, int high, int cutoff) {
#pragma omp parallel
#pragma omp single
    {
        quickSort_parallel_internal(arr, low, high, cutoff);
#pragma omp taskwait
    }
}
```

主程序中，先根据指定的数组大小，随机产生数组（元素不妨设置为 1000 以内，只要是 int 类型的，都不会改变排序算法的开销）

```
if (argc != 3){
    fprintf(stderr, "Usage: %s <desired_num_threads> <desire_size_of_array>\n", argv[0]);
    exit(1);
}
srand((unsigned int)time(NULL));
int SIZE = atoi(argv[2]);  // 将第二个命令行参数转换为整数
int parallelArr[SIZE];
int serialArr[SIZE];
printf("                          OpenMP PQS \n\n");
printf("-------------------------Generate array-------------------------");
printf("\nArray Size: %d\n\n", SIZE);
for (int i = 0; i < SIZE; i++) {
    parallelArr[i] = (rand() % 1000) + 1;
    serialArr[i] = parallelArr[i];
}
printf("Array Before Sorted:\n");
printArray(parallelArr, SIZE);
int n = sizeof(parallelArr) / sizeof(parallelArr[0]);
```

然后设置并行区域所使用的进程数，并且执行排序算法，以及测试时间。

```
int desired_num_threads = atoi(argv[1]);
omp_set_num_threads(desired_num_threads);
double start_time = omp_get_wtime();

int cutoff = 100;
#pragma omp parallel
{
    #pragma omp single
    {
        int num_threads = omp_get_num_threads();
        printf("Number of threads in parallel region: %d\n", num_threads);
    }
}
quickSort_parallel(parallelArr, 0, n - 1, cutoff);
double end_time = omp_get_wtime();
```

```
printf("\nWork took %f seconds\n\n", end_time - start_time);
```

最后执行串行快速排序算法，比对结果与测试串行算法所需要时间以度量加速比。

```
printf("\n\n------------------------Validate Result------------------------\n\n");
double start_time_serial = omp_get_wtime();
serialQuickSort(serialArr, 0, n - 1);
double end_time_serial = omp_get_wtime();

printf("\nSerial Work took %f seconds\n\n", end_time_serial - start_time_serial);
int valid = 1;
for (int i = 0; i < SIZE; i++) {
    if (serialArr[i] != parallelArr[i]) {
        printf("Unmatch element in element %d  value: %d\n", i, serialArr[i]);
        valid = 0;
    }
}
```

3.2 并行正则采样排序算法(MPI)

定义归并排序和快速排序等算法方便后续使用。

```
int cmp(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

// 归并排序的合并操作
void merge(int a[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int *b = (int *)malloc((n1 + 1) * sizeof(int));
    int *c = (int *)malloc((n2 + 1) * sizeof(int));
    memcpy(b, a + p, n1 * sizeof(int));
    memcpy(c, a + q + 1, n2 * sizeof(int));
    b[n1] = INT32_MAX;
    c[n2] = INT32_MAX;
    int i = 0, j = 0;
    for (int k = p; k <= r; k++) {
        if (b[i] <= c[j]) {
            a[k] = b[i];
            i++;
        } else {
            a[k] = c[j];
            j++;
        }
    }
    free(b);
    free(c);
}

// 归并排序
void mergesort(int a[], int p, int r, int pos[], int size[]) {
    if (p < r) {
        int q = (p + r) / 2;
        mergesort(a, p, q, pos, size);
        mergesort(a, q + 1, r, pos, size);
        merge(a, pos[p], pos[q] + size[q] - 1, pos[r] + size[r] - 1);
    }
}
```

```
void swap(int* a, int* b) {

    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void serialQuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        serialQuickSort(arr, low, pi - 1);
        serialQuickSort(arr, pi + 1, high);
    }
}
```

在主函数中，首先根据命令行参数指定的数组大小生成随机数组。（与 3.1 相同，代码不再展示）

阶段 1：无序序列的划分以及局部排序。使用 localSize 标记每个进程收到的数组数据，MPI_Scatter 方法将数据在进程中分发，再用 MPI_Gather 方法将局部排好序的数组会聚到根进程中。

```
// 阶段1：无序序列的划分及局部排序
// Scatter data to all processes
int localSize = SIZE / numprocs;
int *localArr = (int *)malloc(sizeof(int) * localSize);
MPI_Scatter(parallelArr, localSize, MPI_INT, localArr, localSize, MPI_INT, 0, MPI_COMM_WORLD);
qsort(localArr, localSize, sizeof(int), cmp);
MPI_Gather(localArr, localSize, MPI_INT, parallelArr, localSize, MPI_INT, 0, MPI_COMM_WORLD);
```

阶段 2：正则采样，将局部排好序的数组进行正则采样，采样的数据存放在 samples 中，再使用 MPI_Bcast 广播到所有的进程中。

```
// 阶段2：正则采样
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(parallelArr, SIZE, MPI_INT, 0, MPI_COMM_WORLD);
int *samples = (int*)malloc(numprocs*numprocs*sizeof(int));
for(int i = 0; i < numprocs; i++){
    // printf("i:%d, index: %d, value:%d\n",i,n/numprocs*myid+n*i/(numprocs*numprocs),parallelArr[n/numprocs*myid+n*i/(numprocs*numprocs)]);
    samples[numprocs*myid + i] = parallelArr[n/numprocs*myid+n*i/(numprocs*numprocs)];
}
for(int i = 0; i < numprocs; i++){
    MPI_Bcast(samples+numprocs*i, numprocs*sizeof(int), MPI_BYTE, i, MPI_COMM_WORLD);
}
```

阶段 3：对采样数组进行排序并且间隔地选取主元。

```
//阶段3 采样排序、确定主元
//先对采样数组进行排序；
if(myid == 0)
    qsort(samples, numprocs*numprocs, sizeof(int), cmp);
// 选取主元
int *main_val = (int*)malloc((numprocs - 1)*sizeof(int));
if(myid == 0){
    for(int i = 0; i < numprocs - 1; i++){
        main_val[i] = samples[numprocs*(i + 1)];
    }
}
```

阶段 4：将这 p-1 个主元分发到所有的进程中。

```
//阶段4 分发这p-1个主元
MPI_Bcast(main_val, (numprocs-1)*sizeof(int), MPI_BYTE, 0, MPI_COMM_WORLD);
```

阶段 5：根据这 p-1 个主元将有序序列划分成 p 段，这里定义了一个 blocksize 数组，每个处理器中都有这个数组，用于记录在该进程的局部数组中进行主元划分后 p 个小块，每个小块的元素个数，然后统一到一个 blocksizes 数组中，blocksizes[i]就表示第 i//p 号的进程需要传递到 i%numproc 号进程的元素个数。

```
//阶段5 根据p-1个主元将有序序列划分成p段
//每个线程都有一个这个数组，只需要记录每个线程的块大小数组即可，这也就是通讯的量
//blocksizes[i]表示 第i//numproc号线程 需要给到 第i%numproc号线程 的元素个数
int *blocksize = (int*)malloc(numprocs*sizeof(int));
memset(blocksize, 0, numprocs*sizeof(int));
int index = 0;
for(int i = 0; i < n/numprocs; i++){
    while(parallelArr[n/numprocs*myid+i] > main_val[index]){ // 很巧妙的一个设计
        index += 1;
        if(index == numprocs - 1){
            blocksize[numprocs - 1] = n/numprocs - i;
            break;
        }
    }
    if(index == numprocs - 1){
        blocksize[numprocs - 1] = n/numprocs - i;
        break;
    }
    blocksize[index]++;
}
int *blocksizes = (int*)malloc(numprocs*numprocs*sizeof(int));
for(int i = 0; i < numprocs; i++){
    blocksizes[numprocs*myid + i] = blocksize[i];
}
for(int i = 0; i < numprocs; i++){
    MPI_Bcast(blocksizes+numprocs*i, numprocs*sizeof(int), MPI_BYTE, i, MPI_COMM_WORLD);
}
```

阶段 6：全局交换，这也是代码中最核心的一部分，首先每个进程需要统计接收块的大小，这个通过调用 MPI_Alltoall 方法传递 blocksize 数组中的元素来完成，然后统计每个进程最终处理的总元素数量，并把最后的数组存在 array_exchanged 中，其中 sendPos 为每隔进程需要发送给其他进程的位置标记。然后调用 MPI_Barrier 来设同步点，MPI_Alltoallv 来执行可变通信。

```
//阶段6 全局交换
//blocksizes[i]表示 第i//numproc号进程 需要给到 第i%numproc号进程 的元素个数
// 各进程统计要接收的块的大小
int* recv_size = (int*)malloc(numprocs*sizeof(int));
MPI_Alltoall(blocksize, 1, MPI_INT, recv_size, 1, MPI_INT, MPI_COMM_WORLD);
int totalSize = 0;
for(int i = 0; i < numprocs; i++){
    totalSize += blocksizes[myid + i*numprocs];
}
// 全局交换
int *array_exchanged = (int*)malloc(totalSize*sizeof(int));
int *sendPos = (int*)malloc(numprocs*sizeof(int));
int *recvPos = (int*)malloc(numprocs*sizeof(int));
sendPos[0] = 0;
recvPos[0] = 0;
for(int i = 1; i < numprocs; i++){
    sendPos[i] = sendPos[i - 1] + blocksize[i - 1];
    recvPos[i] = recvPos[i - 1] + recv_size[i - 1];
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Alltoallv(parallelArr+n/numprocs*myid, blocksize, sendPos, MPI_INT, array_exchanged, recv_size, recvPos, MPI_INT, MPI_COMM_WORLD);
```

阶段 7：归并排序并统一发送到进程 0。其中 listSize 存放每个进程的总元素数，方便确定后面接收的位置。

```
//阶段7 归并排序 发送到进程0
mergesort(array_exchanged, 0, numprocs - 1, recvPos, recv_size);
//下面的数组存放每个进程的totalsize大小，方便确定后面接受的位置
int *listSize = (int*)malloc(numprocs*sizeof(int));
MPI_Gather(&totalSize, 1, MPI_INT, listSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
if(myid == 0){
    recvPos[0] = 0;
    for(int i = 1; i < numprocs; i++){
        recvPos[i] = recvPos[i - 1] + listSize[i - 1];
    }
}
MPI_Gatherv(array_exchanged, totalSize, MPI_INT, parallelArr, listSize, recvPos, MPI_INT, 0, MPI_COMM_WORLD);
```

最后执行串行快速排序算法，比对结果与测试串行算法所需要时间以度量加速比。

```
printf("\n\n------------------------Validate Result------------------------\n\n");
double start_time_serial = omp_get_wtime();
serialQuickSort(serialArr, 0, n - 1);
double end_time_serial = omp_get_wtime();

printf("\nSerial Work took %f seconds\n\n", end_time_serial - start_time_serial);
int valid = 1;
for (int i = 0; i < SIZE; i++) {
    if (serialArr[i] != parallelArr[i]) {
        printf("Unmatch element in element %d  value: %d\n", i, serialArr[i]);
        valid = 0;
    }
}
```

4. 实验测试

4.1 实验结果测试

运行 test_correct.sh 脚本，得到输出符合预期，也可以在 output 目录下的 txt 文件中查看具体的输出情况。

```bash
#!/bin/bash

# Run PQS_OpenMP
./build/PQS_OpenMP 10 1000 > output/output_PQS_OpenMP.txt

# Run PSRS_MPI
mpirun -n 10 ./build/PSRS_MPI 1000 > output/output_PSRS_MPI.txt

# Run tests
output_PQS_OpenMP=$(./build/PQS_OpenMP 10 1000)
output_PSRS_MPI=$(mpirun -n 10 ./build/PSRS_MPI 1000)

# Check if both outputs contain the expected message
if [[ $output_PQS_OpenMP == *"Good! Result is valid."* && $output_PSRS_MPI == *"Good! Result is valid."* ]]; then
    echo "Test passed: Results match expected output."
else
    echo "Test failed: Results do not match expected output."
fi
```

```
jackie@linr:/mnt/c/Users/24398/Desktop/pj$ ./test_correct.sh
Test passed: Results match expected output.
```

output_PSRS_MPI.txt 文件如下图所示，输出结果符合预期。

```
                    MPI PSRS

------------------------Generate array------------------------
Array Size: 1000

Array Before Sorted:
583 559 236 981 131 217 614 606 399 568 349 389 296 598 440 912 848 644 142 281 464 496 192 496 748 4
Number of threads: 10
Exucution Completed!

------------------------Result------------------------

Sorted array:
2 2 3 3 5 5 5 6 11 11 12 12 12 13 18 18 19 21 24 25 25 26 26 26 27 27 28 30 31 32 33 35 35 36 37 3

Work took 0.000700 seconds


Serial Work took 0.000040 seconds



------------------------Validate Result------------------------

Good! Result is valid.
```

也可以在终端中运行可执行文件。例如./build/PQS_OpenMP 10 1000 和 mpirun -np 4 ./build/PSRS_MPI 1000，得到的输出如下。当然也可以在 examples 中查看运行的样例。如./PQS_OpenMP.sh，结果也将如下图所示。

```
-------------------------Result-------------------------

Sorted array:
3 7 7 7 8 8 9 12 12 13 14 14 15 15 18 20 21 21 22 23 24 25 26 26 28 29 29 30 30 30 31 36 3
1 102 102 105 105 105 106 106 107 107 107 108 109 110 110 110 111 111 114 114 117 119 119
57 163 164 165 167 170 173 173 173 174 174 176 177 177 178 179 183 183 183 184 185 186 187
230 230 230 231 231 233 233 234 235 235 236 237 238 238 241 241 242 242 245 246 249 249 25
 296 297 298 298 299 300 301 301 301 302 304 305 305 307 307 308 311 311 311 312 314 314 3
2 354 354 354 357 357 358 358 359 362 363 364 364 370 370 373 373 375 376 377 380 381 381
34 434 435 435 437 438 438 440 441 441 442 442 443 444 447 448 448 449 450 451 451 452 454
507 508 508 509 512 513 513 513 515 515 516 516 517 518 519 525 525 527 527 529 529 529 53
 577 577 580 582 582 584 586 588 588 592 593 594 595 596 597 598 599 599 602 603 604 604 6
6 657 658 659 659 659 660 660 661 662 662 665 665 666 666 669 671 673 674 675 675 676 676
19 719 720 720 720 720 721 723 725 725 725 725 725 725 727 727 728 729 732 732 734 734 735
776 777 779 779 780 781 783 784 784 788 788 790 791 791 791 793 794 795 796 796 797 797 79
 839 839 839 840 841 842 842 847 850 852 853 854 854 854 855 856 857 858 858 864 865 866 8
5 906 908 909 909 910 910 911 913 913 914 915 915 917 918 921 923 923 923 925 925 926 927
78 979 979 980 980 980 981 981 982 982 982 983 985 985 986 987 987 988 988 988 991 992 993

Work took 0.000357 seconds



-------------------------Validate Result-------------------------


Serial Work took 0.000078 seconds

Good! Result is valid.
```

4.2 实验性能测试

在实验的源代码 PQS_OpenMP.cpp 和 PSRS_MPI.cpp 中有串行快速排序的算法，我们可以用它来验证并行算法的正确性以及度量性能。

4.2.1 串行快速排序算法性能测试

在 PQS_OpenMP.cpp 中指定不同的数据量大小，串行快速排序算法可以打印出对应的执行时间。如下表，对 900000 个元素的数组和 1000000 数组分别做测试的原因是在并行快速排序(OpenMP)算法中，指定数据量为 1000000 会出现内存不够的段错误，故只能指定数据量为 900000，但并行正则采样排序(MPI)算法中可以指定数据量为 1000000。

n	1000	5000	10000	50000	100000	500000	900000	1000000
time	0.000062	0.00037	0.000846	0.007533	0.022737	0.417134	1.199372	1.477752

```
printf("\n\n-------------------------Validate Result-------------------------\n\n");
double start_time_serial = omp_get_wtime();
serialQuickSort(serialArr, 0, n - 1);
double end_time_serial = omp_get_wtime();

printf("\nSerial Work took %f seconds\n\n", end_time_serial - start_time_serial);
int valid = 1;
for (int i = 0; i < SIZE; i++) {
    if (serialArr[i] != parallelArr[i]) {
        printf("Unmatch element in element %d  value: %d\n", i, serialArr[i]);
        valid = 0;
    }
}
```

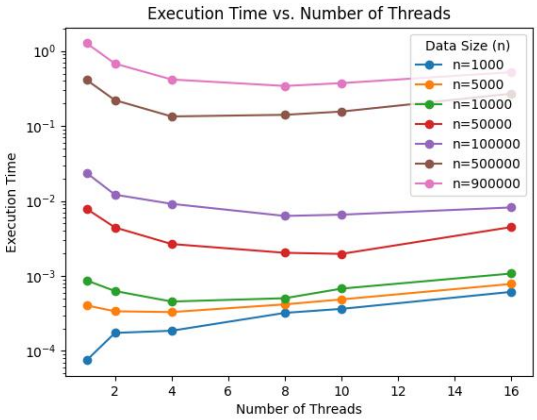
4.2.2 并行快速排序(OpenMP)的性能测试

在 PQS_OpenMP.cpp 中指定不同的数据量大小和并行线程数测试时间如下表。

n/thread	1000	5000	10000	50000	100000	500000	900000
1	0.000075	0.000405	0.000872	0.007855	0.023664	0.41136	1.271363
2	0.000174	0.000338	0.00063	0.004438	0.012143	0.221102	0.679611
4	0.000186	0.00033	0.000456	0.002668	0.009162	0.134183	0.417525

8	0.000322	0.000418	0.000505	0.002037	0.00632	0.141105	0.342172
10	0.000364	0.000487	0.000678	0.001971	0.006571	0.155878	0.373318
16	0.000614	0.000783	0.001078	0.004493	0.008205	0.269768	0.522037

绘制图像如下（图像存在 plot_data 目录中）：



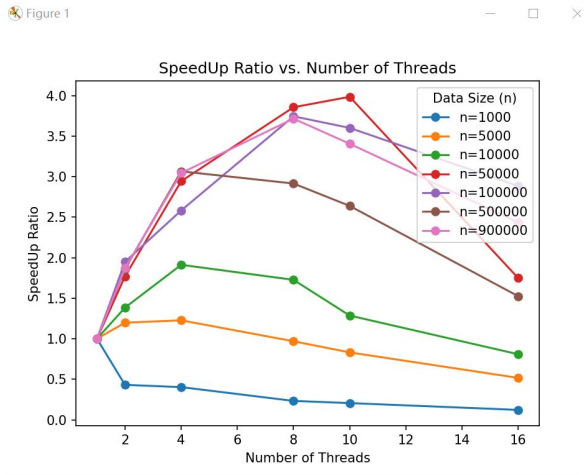
发现当线程数为 1 时执行时间与串行代码执行时间接近，可将其作为串行执行的基准时间，使用如下公式计算加速比：

$$\text{SpeedUp} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

得到加速比如下：

n/thread	1000	5000	10000	50000	100000	500000	900000
1	1	1	1	1	1	1	1
2	0.431	1.1982	1.3841	1.7699	1.9488	1.8605	1.8707
4	0.4032	1.2273	1.9123	2.9442	2.5828	3.0657	3.0450
8	0.2329	0.9689	1.7267	3.8562	3.7443	2.9153	3.7156
10	0.206	0.8316	1.2861	3.9853	3.6013	2.6390	3.4056
16	0.1221	0.5172	0.8089	1.7483	2.8841	1.5249	2.4354

绘制加速比与线程数、数据量的关系如下：



可以发现，加速比在线程数为 4、8、10 时都有可能达到最大值，当线程数超过 10 时会引起性能下降。

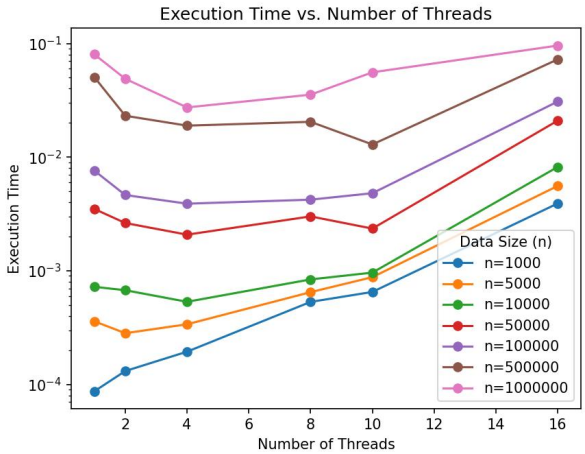
4.2.3 并行正则采样排序(MPI)的性能测试

在 PSRS_MPI.cpp 中指定不同的数据量大小和并行线程数测试时间如下表。

n/thread	1000	5000	10000	50000	100000	500000	1000000
1	0.000087	0.000359	0.000722	0.003496	0.007597	0.05052	0.080214
2	0.000131	0.000282	0.000674	0.002631	0.00464	0.023125	0.048881

4	0.000194	0.000338	0.000533	0.002078	0.003893	0.018926	0.027372
8	0.000532	0.000648	0.000839	0.003008	0.004222	0.020449	0.035462
10	0.000651	0.000878	0.000963	0.00235	0.004813	0.012921	0.055755
16	0.003912	0.00558	0.008192	0.020949	0.030838	0.072493	0.095872

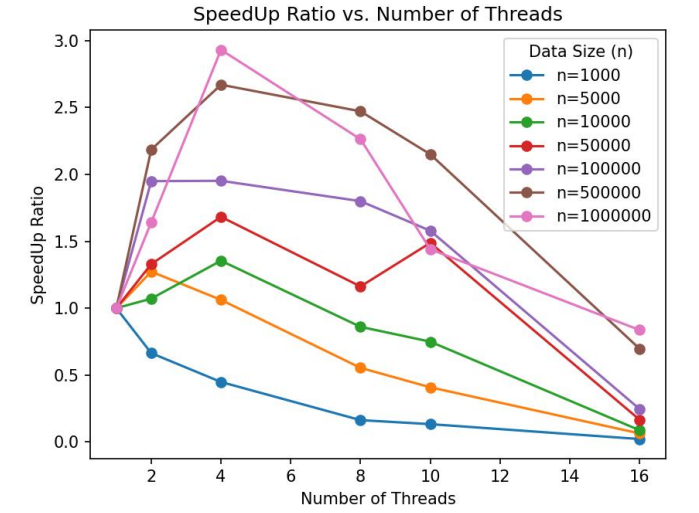
绘制图像如下（图像存在 plot_data 目录中）：



当线程数为 1 时执行时间与串行快速排序算法执行时间不同，仍然将其作为串行执行的基准时间（确保使用的是同一个算法），使用同样的公式计算加速比。绘制加速比表格如下：

n/thread	1000	5000	10000	50000	100000	500000	1000000
1	1	1	1	1	1	1	1
2	0.6641	1.273	1.0712	1.3288	1.9488	2.1846	1.641
4	0.4485	1.0621	1.3546	1.6824	1.9515	2.6693	2.9305
8	0.1635	0.554	0.8605	1.1622	1.7994	2.4705	2.262
10	0.1336	0.4089	0.7497	1.4877	1.5784	2.1495	1.4397
16	0.0222	0.0643	0.0881	0.1669	0.2464	0.6969	0.8367

绘制加速比与线程数、数据量的关系如下：



可以发现，加速比在线程数为 4 时达到最大值，当线程数超过 10 时会引起性能的大幅度下降，当线程数为 16 时，数据量达到 1000000 执行时间都比串行算法还慢。

4.3 实验结果分析

4.3.1 在串行快速排序算法性能测试中使用了不同规模的数据量，并记录了相应的执行时间。随着数据量的增加，执行时间也相应增加，这是符合预期的。串行算法的执行时间呈现出明显的线性关系，即随着问题规模的增加，执

行时间线性增长。

4.3.2 在并行快速排序算法(OpenMP)中，使用了不同的线程数和数据量，并记录了相应的执行时间。根据加速比的计算结果，可以得出以下结论：

当线程数较小时，加速比呈现出增加的趋势，说明并行算法在一定范围内能够有效地提高性能。

然而，当线程数超过一定阈值时，加速比开始下降。这可能是因为线程数增加导致了额外的开销，例如线程间的通信和同步，从而抵消了并行化带来的好处。

并且当线程数为 1 时，算法的执行时间与串行快速排序算法执行时间基本吻合，证明了算法的正确性。

4.3.3 在并行正则采样算法(MPI)中，同样使用了不同的线程数和数据量，并记录了相应的执行时间，执行时间基本符合：

在一定范围内增加并行线程数，运行时间减少；增加数据量，运行时间增大。

根据加速比的计算结果，可以得出以下结论：和 OpenMP 的快速排序相似，当线程数较小时，加速比呈现出增加的趋势，说明并行算法在一定范围内能够有效地提高性能。当线程数超过一定阈值时，加速比开始下降。这也可能是因为线程数增加导致了额外的通信和同步开销。

但是当并行正则采样算法(MPI)的线程数设置为 1 时，算法的运行时间会比串行化快速排序或 OpenMP 快速排序线程设为 1 时快一些，经过对照试验，发现是-O2 编译优化选项带来的优化 MPI 其他编译优化共同作用的结果（就是会比串行算法快一点）。编译器对代码进行优化的方式，以及对循环展开、内联等优化技术的应用会影响程序的性能，但有时也可能引入一些复杂性。一般来说，开启优化选项有助于提高程序的执行效率，但在特定情况下，可能需要根据具体问题进行调整。

改进后（CXXFLAGS 是 PQS_OpenMP 的编译指令）：

```
CC = mpicxx
CXX = mpicxx
CFLAGS = -Wall -O2 -fopenmp
CXXFLAGS = -Wall -O2 -fopenmp
LDFLAGS = -lm -fopenmp
```

改进前：

```
CC = mpicxx
CXX = g++
CFLAGS = -Wall -O2
CXXFLAGS = -Wall -fopenmp
LDFLAGS = -lm -fopenmp
```

于是我使用-O2 编译优化选项，重新做了 OpenMP 的测速实验，如 4.3.4 中所示。

4.3.4 编译优化选项下的并行快速排序(OpenMP)实验

执行时间表格如下：

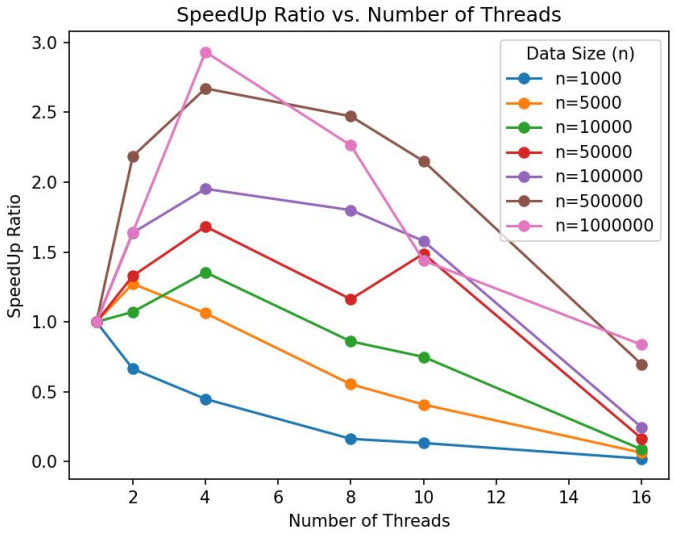
n/thread	1000	5000	10000	50000	100000	500000	900000
1	0.000057	0.000253	0.000469	0.003451	0.008454	0.126038	0.275193
2	0.000124	0.000376	0.000364	0.002401	0.005476	0.0833	0.192562
4	0.000243	0.000433	0.000457	0.001882	0.004327	0.078546	0.201898
8	0.000296	0.000545	0.000637	0.002485	0.007944	0.123499	0.256894
10	0.00042	0.000819	0.000822	0.0024	0.009752	0.157787	0.323109
16	0.000622	0.000844	0.001079	0.004288	0.011036	0.24176	0.522001

加速比表格如下：

n/thread	1000	5000	10000	50000	100000	500000	1000000
1	1	1	1	1	1	1	1
2	0.664122137	1.273049645	1.071216617	1.32877233	1.637284483	2.184648649	1.641005708
4	0.448453608	1.062130178	1.354596623	1.68238691	1.951451323	2.66934376	2.930512933
8	0.163533835	0.554012346	0.860548272	1.162234043	1.799384178	2.470536457	2.26197056
10	0.133640553	0.408883827	0.749740395	1.487659574	1.57843341	2.149512828	1.438687113

16	0.022239264	0.064336918	0.088134766	0.166881474	0.246351903	0.696894873	0.836678071

绘制加速比图像如下：



5. 选做部分

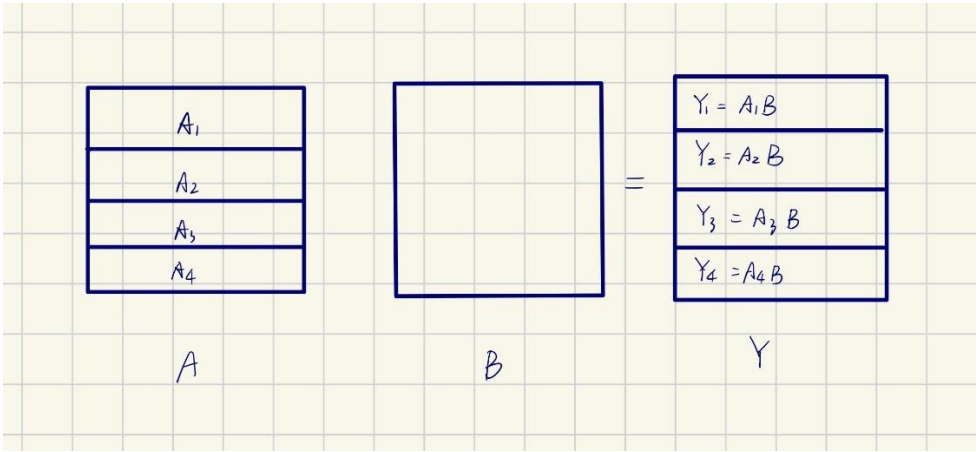
参考 PPT 上的选题与此前读论文所得到的启发[1]，本次实验选做部分选择了 MPI 与 CUDA 实现矩阵乘法以及对于 Megatron 中矩阵乘法的简析。

5.1 算法逻辑

矩阵乘法可以使用两种切分方式以实现并行化：

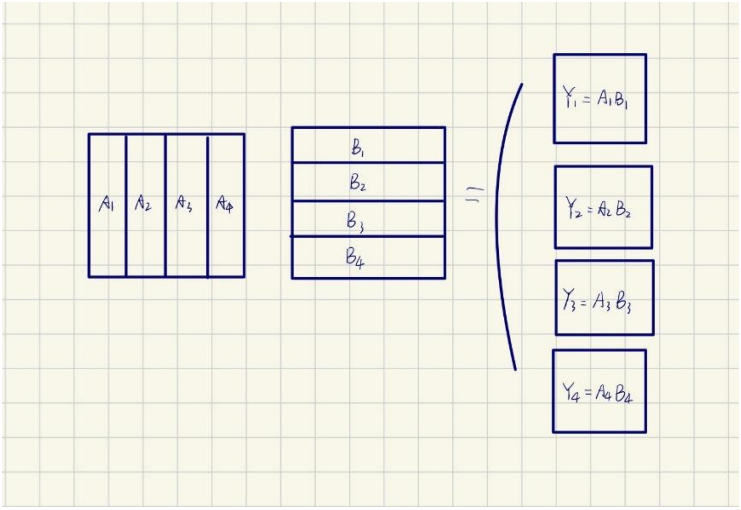
5.1.1

我们考虑矩阵 A • B，可以使用下图所示切分方式，将 A 按行切分，然后分发给每个处理器；B 不切分（直接分发给所有处理器），所得到的结果在每个处理器上只有完整矩阵的一部分，最后将其拼接起来（这里使用 MPI_Gather 操作将其汇总）得到结果矩阵。A 不切分，B 按列切分的逻辑与这种切分方式的逻辑本质上是相同的，在此不再赘述。



5.1.2

也可以将 A 按列切分，B 按行切分，如下图所示，这样每个处理器作乘法之后会得到一个与结果矩阵大小相同的矩阵，再将这些矩阵作一个 MPI_Reduce 操作将其相加汇总到根进程即可得到结果矩阵（如果需要对结果矩阵在每个处理器上进行下一步计算的话，可以使用 MPI_AllReduce 操作，再传回所有的处理器。



5.2 代码实现

5.2.1 A 按行切分，B 不切分

首先进行 MPI 的初始化。

```
//MPI Intialization
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid); //当前进程的rank
MPI_Comm_size(MPI_COMM_WORLD, &numprocs); //获取进程的总数
```

传入命令行参数并解析等。

```
//Checking if valid number of arguements have been passed
if(argc < 6)
{
    if(myid == Root)
        printf("Usage:< mpirun >< -n >< Number of processors >< ./Program Name >< Number of Rows of Ma
    MPI_Finalize();
    exit(-1);
}
if ((argc >= 6 && strcmp(argv[5],"-v") == 0)){
    verify=1;
    printf("v=1");
}

if ((argc ==7 && strcmp(argv[6],"-p") == 0) || (argc == 6 && strcmp(argv[5],"-p")==0)){
    print=1;
}
//Assigning values to RowsNo, ColsNo, VectorSize from the arguements passed
RowsNo = atoi( argv[1] ); //矩阵1的行数
ColsNo = atoi( argv[2] ); //矩阵1的列数
RowsNo2= atoi( argv[3] ); //矩阵2的行数
ColsNo2= atoi( argv[4] ); //矩阵2的列数
```

矩阵的初始化：在根进程中，对 A、B 矩阵初始化，并将结果矩阵初始化为 0。

```
int IntializingMatrixVectors(float **MatrixA, float **MatrixB, float **ResultM, int RowsNo, int ColsNo, int RowsNo2, int ColsNo2)
{
    float *TempMatrixA, *TempVectorB, *TempResultM, *TempMatrixB;
    int Status = 1;  // 初始设为 1，表示初始化成功
    int Index;

    // 分配内存
    TempMatrixA = (float *)malloc(RowsNo * ColsNo * sizeof(float));
    if(TempMatrixA == NULL)
        Status = 0;  // 内存分配失败

    TempMatrixB = (float *)malloc(RowsNo2 * ColsNo2 * sizeof(float));
    if(TempMatrixB == NULL)
        Status = 0;  // 内存分配失败

    TempResultM = (float *)malloc(RowsNo * ColsNo2 * sizeof(float));
    if(TempResultM == NULL)
        Status = 0;  // 内存分配失败

    // 初始化矩阵和向量
    int a = 10;
    for(Index = 0; Index < RowsNo * ColsNo; Index++)
        TempMatrixA[Index] = (float)rand() / (float)(RAND_MAX / a);

    for(Index = 0; Index < RowsNo2 * ColsNo2; Index++)
        TempMatrixB[Index] = (float)rand() / (float)(RAND_MAX / a);

    for(Index = 0; Index < ColsNo2 * RowsNo; Index++)
        TempResultM[Index] = 0.0f;
```

计算分割矩阵的大小和部分结果矩阵的大小，然后将矩阵 A 分割并传输给各个进程。

```
//计算分割大小以及结果矩阵的大小
ScatterSize = RowsNo / numprocs;
elements = (RowsNo*ColsNo2)/numprocs;

MyMatrixA = (float *)malloc(ScatterSize * ColsNo * sizeof(float));
if(MyMatrixA == NULL)
    Status = 0;

MyResultMatrix = (float *)malloc(elements* sizeof(float));
if(MyResultMatrix == NULL)
    Status = 0;

MPI_Scatter(MatrixA, ScatterSize * ColsNo, MPI_FLOAT, MyMatrixA, ScatterSize * ColsNo, MPI_FLOAT, Root, MPI_COMM_WORLD);
```

对 GPU 进行内存分配与内存拷贝，然后进行计算，其中 blockSize 为每一个线程块（二维）的大小，gridSize 整个网格（grid）中包含的线程块数，保证 blockSize*gridSize 为一个进程结果矩阵（也就是最终矩阵结果）行和列的大小，这样可以保证网格中的一个单元由一个线程计算，最大化利用并行性。

```
  else
  {
      cudaSetDevice(myid);
      SAFE( cudaMalloc( (void **)&DeviceMA, ScatterSize * ColsNo * sizeof(float) ) );
      SAFE( cudaMalloc( (void **)&DeviceMB, bsize*sizeof(float) ) );
      SAFE( cudaMalloc( (void **)&DeviceMyResultM, elements * sizeof(float) ) );
      cudaMemcpy( (void *)DeviceMA, (void *)MyMA, ScatterSize * ColsNo * sizeof(float), cudaMemcpyHostToDevice );
      cudaMemcpy( (void *)DeviceMB, (void *)MB,  bsize*sizeof(float), cudaMemcpyHostToDevice );
      start_time = MPI_Wtime();…
      dim3 blockSize(16, 16);
      dim3 gridSize((ColsNo2 + blockSize.x - 1) / blockSize.x, (ScatterSize + blockSize.y - 1) / blockSize.y);
      MatrixMultiplication<<<gridSize, blockSize>>>(DeviceMA, DeviceMB, DeviceMyResultM, RowsNo, ColsNo, ColsNo2, Sca
      cudaMemcpy( (void *)MyResultMatrix, (void *)DeviceMyResultM, elements * sizeof(float), cudaMemcpyDeviceToHost )
      end_time = MPI_Wtime();
  }
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Gather(MyResultMatrix,elements, MPI_FLOAT, ResultM, elements, MPI_FLOAT, Root, MPI_COMM_WORLD);//收集到跟进程的R
```

其中 CUDA 矩阵乘法核函数的定义如下：

```
__global__ void MatrixMultiplication(float *MA, float *MB, float *Res, int r1, int c1, int c2, int ScatterSize, int myid) {
    //单线程矩阵乘法（一个元素）的核函数
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    // 取出矩阵的行和列（在网格中）
    if (row < ScatterSize && col < c2) { //单线程进行矩阵乘法
        float sum = 0.0;
        for (int k = 0; k < c1; ++k) {
            sum += MA[row * c1 + k] * MB[k * c2 + col];
        }
        Res[row * c2 + col] = sum;
    }
}
```

进程同步与矩阵收集（拼接）。

```
MPI_Barrier(MPI_COMM_WORLD);
MPI_Gather(MyResultMatrix,elements, MPI_FLOAT, ResultM, elements, MPI_FLOAT, Root, MPI_COMM_WORLD);//收集到跟进程的ResultM中
```

测试结果与运行时间的实验：

```
  //验证
  int valid = 1;
  if (myid == Root){

      CPUResultM = (float *)malloc(RowsNo * ColsNo2 * sizeof(float));
      double start_time_cpu = MPI_Wtime();
      for (int i = 0; i < RowsNo; i++) {
              for (int j = 0; j < ColsNo2; j++)
              {
                  float sum = 0.0;
                  for (int k = 0; k < RowsNo2; k++)
                      sum = sum + MA[i * ColsNo + k] * MB[k * ColsNo2 + j];

                  CPUResultM[i * ColsNo2 + j] = sum;
              }
      }
      double end_time_cpu = MPI_Wtime();
```

```
if(verify==1)
{
    printf("\n\n------------------------Validate Result------------------------\n\n");
    for(Index = 0; Index < ColsNo2 * RowsNo; Index++)
    {
        float a = ResultM[Index];
        float b = CPUResultM[Index];
        if (abs(a,b) >=  0.01 * a)
        {
            valid = 0;
            printf("Unmatch element in element [%d, %d], gpu_value: %f, cpu_value: %f\n", \
            Index/ColsNo2,Index % ColsNo2, ResultM[Index], CPUResultM[Index]);
        }
    }
    if (valid == 1) {
        printf("Good! Result is valid.\n");
    } else {
        printf("Result is invalid!!!\n");
    }
}
printf("\nGPU Work took %f seconds\n\n", end_time - start_time);
printf("\nCPU Work took %f seconds\n\n", end_time_cpu - start_time_cpu);
}
```

5.2.2 A 按列划分，B 按行划分

其余部分都与 5.2.1 相同，不同的只有矩阵的分割、分发与核函数的定义和调用：
对 B 矩阵进行分割与分发。

```
//计算分割大小以及结果矩阵的大小
ScatterSize = ColsNo / numprocs; //10
elements = (RowsNo*ColsNo2);
MyMatrixB = (float *)malloc(ScatterSize * ColsNo2 * sizeof(float));
MPI_Scatter(MatrixB, ScatterSize * ColsNo2, MPI_FLOAT, MyMatrixB, ScatterSize * ColsNo2, MPI_FLOAT, Root, MPI_COMM_WORLD);
```

使用 MPI_Scatterv()方法对 A 矩阵进行分割与分发，

```
int *sendcounts = (int *)malloc(numprocs * sizeof(int));
int *displs = (int *)malloc(numprocs * sizeof(int));

if (myid == 0) {
    // Calculate sendcounts and displs for each process
    for (int i = 0; i < numprocs; i++) {
        sendcounts[i] = ScatterSize;
        displs[i] = i * ScatterSize;
    }
}
// 切分，分发
for(int j = 0; j < RowsNo; j++)
{
    MPI_Scatterv(MatrixA+j*ColsNo, sendcounts, displs, MPI_FLOAT, MyMatrixA+j*ScatterSize, ScatterSize*numprocs, MPI_FLOAT, 0, MPI_COMM_WORLD);
}
```

测试发现每个矩阵能够正确收到所需要的元素。

```
jackie@linr:/mnt/c/Users/24398/Desktop/multigpumatmul-master$ make
/usr/bin/nvcc -Xcompiler -fopenmp -g -w -O4 -I..   -I /usr/include
/usr/bin/nvcc -Xcompiler -fopenmp -g -w -O4 -I..   -I /usr/include
jackie@linr:/mnt/c/Users/24398/Desktop/multigpumatmul-master$ mpir
v=1Resultant Matrix Number of Elements is 4096


v=1v=1v=1v=1v=1v=1Process 0 received:
8.401877 3.943829 7.830992 7.984400 9.116473 1.975514 3.352227 7.6
2.666657 5.397604 3.752070 7.602487 5.125354 6.677238 5.316064 0.3
7.497709 3.686635 2.941604 2.322615 5.844885 2.444127 1.523898 7.3
7.478093 6.289099 0.354209 7.478028 8.332385 9.253765 Process 1 re
2.777747 5.539700 4.773971 6.288709 3.647845 5.134009 9.522297 9.1
4.376376 9.318351 9.308098 7.209523 2.842934 7.385343 6.399788 3.5
1.254749 7.934704 1.641019 7.450714 0.745298 9.501040 0.525293 5.2
9.794341 7.438112 9.033663 9.835958 6.668803 4.972585 1.639680 8.3
5.324409 0.876436 2.604970 8.773839 6.861249 0.937402 1.112756 3.6
3.248070 9.318953 9.084846 6.220954 8.368278 8.181276 4.960744 3.3
8.090953 1.317021 0.515083 0.534223 4.577156 7.808683 6.920764 4.4
2.604449 6.496590 5.523164 9.195909 6.859863 8.097853 6.978482 3.1
6.572009 9.952999 9.358518 3.245414 8.743093 5.891567 6.377709 7.5
8.990861 7.670136 3.335692 5.367425 2.191360 4.775512 9.498203 4.6
9.466405 1.223258 Process 2 received:
6.357117 7.172969 1.416026 6.069689 0.163006 2.428868 1.372316 8.0
6.878614 1.659742 4.401045 8.800752 8.292011 3.303371 2.289682 8.9
1.762107 2.400624 7.977980 7.326544 6.565636 9.674051 6.394584 7.5
8.889487 0.769947 6.497069 2.480441 6.294797 2.291370 7.006199 3.1
```

然后对 GPU 进行内存分配与内存拷贝，然后进行计算，其中 blockSize 为每一个线程块（二维）的大小，gridSize 整个网格（grid）中包含的线程块数，保证 blockSize*gridSize 为一个进程结果矩阵（也就是最终矩阵结果）行和列的大小，这样可以保证网格中的一个单元由一个线程计算，最大化并行程度。

```
if(DeviceStatus == 0)
{
    printf("cuda is fucked away\n");exit(1);
}
else // GPU可用，在GPU上运行矩阵乘法
{
    cudaSetDevice(myid);
    printf("Unpinned mode\n");
    CUDA_SAFE_CALL( cudaMalloc( (void **)&DeviceMyMatrixA, ScatterSize * RowsNo * sizeof(float) ) );
    CUDA_SAFE_CALL( cudaMalloc( (void **)&DeviceMyMatrixB, ScatterSize * ColsNo2 * sizeof(float) ) );
    CUDA_SAFE_CALL( cudaMalloc( (void **)&DeviceMyResultVector, elements * sizeof(float) ) );
    // 将数据从主机内存同步地传输到设备内存
    cudaMemcpy( (void *)DeviceMyMatrixA, (void *)MyMatrixA, ScatterSize * RowsNo * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( (void *)DeviceMyMatrixB, (void *)MyMatrixB, ScatterSize * ColsNo2 * sizeof(float), cudaMemcpyHostToDevice );
    cudaSetDevice(myid);

    start_time = MPI_Wtime();
    // 定义线程块和网格的大小
    dim3 blockSize(16, 16);  // 16x16 线程块
    dim3 gridSize((ColsNo2 + blockSize.x - 1) / blockSize.x, (RowsNo + blockSize.y - 1) / blockSize.y);     // 调用核函数
    MatrixMultiplication<<<gridSize, blockSize>>>(DeviceMyMatrixA, DeviceMyMatrixB, DeviceMyResultVector, RowsNo, ScatterSize, ColsNo2);
    cudaMemcpy( (void *)MyResultMatrix, (void *)DeviceMyResultVector, elements * sizeof(float), cudaMemcpyDeviceToHost );

}

MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(MyResultMatrix, ResultVector, elements, MPI_FLOAT, MPI_SUM, Root, MPI_COMM_WORLD);
end_time = MPI_Wtime();
```

其中 CUDA 矩阵乘法单线程核函数的定义如下：

```cpp
__global__ void MatrixMultiplication(float *MA, float *MB, float *Res, int r1, int ScatterSize, int c2) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < r1 && col < c2) {
        float sum = 0.0;
        for (int k = 0; k < ScatterSize; ++k) {
            sum += MA[row * ScatterSize + k] * MB[k * c2 + col];
        }
        Res[row * c2 + col] = sum;
    }
}
```

5.3 性能测试

5.3.1 A 按行切分，B 不切分

计算时间：

matrix_sz/cpuorgpu	cpu_serial	proc:1	proc:2	proc:4	proc:8
64x64	0.000158	0.000539	0.000161	0.000062	0.000060
128x128	0.001588	0.000543	0.000172	0.000151	0.000073
256x256	0.013893	0.000836	0.000646	0.000296	0.00021
512x512	0.122866	0.001543	0.00075	0.000591	0.00034
1024x1024	2.057214	0.005913	0.003038	0.001733	0.00117
2048x2048	23.973938	0.033998	0.030786	0.016537	0.005348

当数据量达到 128×128 时，**MPI-CUDA 混合并行**矩阵乘法相比 cpu 的串行矩阵乘法就已经有极大的性能提升。使用如下公式计算加速比：

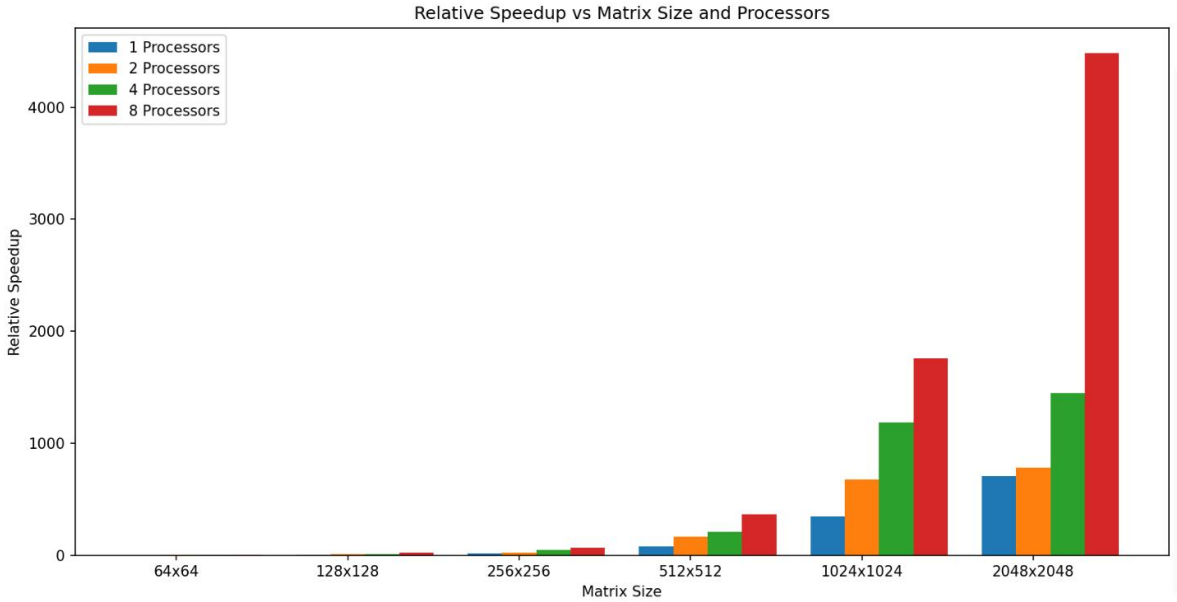
$$\text{SpeedUp} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

加速比的理论峰值为：matrix_sz^2，即每个线程并行计算结果矩阵的一个元素。

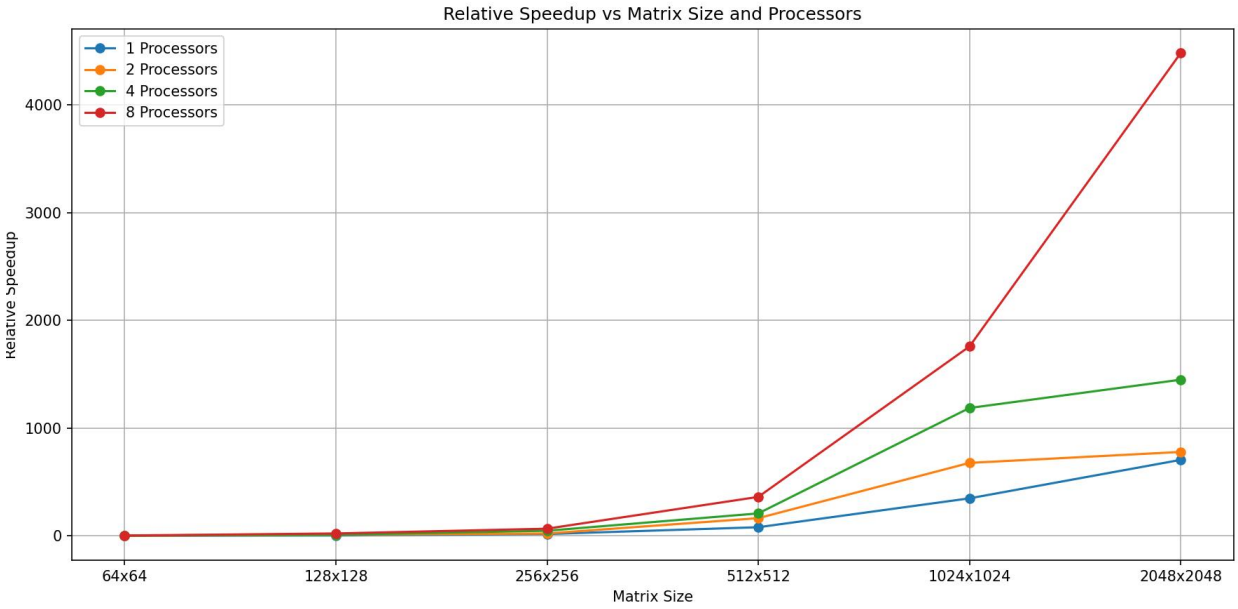
matrix_sz/cpuorgpu	cpu	proc:1	proc:2	proc:4	proc:8
64x64	1	0.293135436	0.98136646	2.548387097	2.633333333
128x128	1	2.924493554	9.23255814	10.51655629	21.75342466
256x256	1	16.61842105	21.50619195	46.93581081	66.15714286
512x512	1	79.62799741	163.8213333	207.8950931	361.3705882
1024x1024	1	347.9137494	677.160632	1187.082516	1758.302564
2048x2048	1	705.1573034	778.7285779	1449.715063	4482.785714

加速比随着 MPI 进程数变大和数据量增大而变大。

绘制加速比柱状如下：



绘制曲线图如下：



5.3.2 A 按列切分，B 按行切分

计算时间：

matrix_sz/cpuorgpu	cpu	proc:1	proc:2	proc:4
64x64	0.000173	0.000562	0.000656	0.00087
128x128	0.001511	0.000568	0.000656	0.000544
256x256	0.014307	0.000691	0.000926	0.000774
512x512	0.129245	0.001575	0.001872	0.003138
1024x1024	2.048039	0.005821	0.008031	0.01062

使用如下公式计算加速比：

$$\text{SpeedUp} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

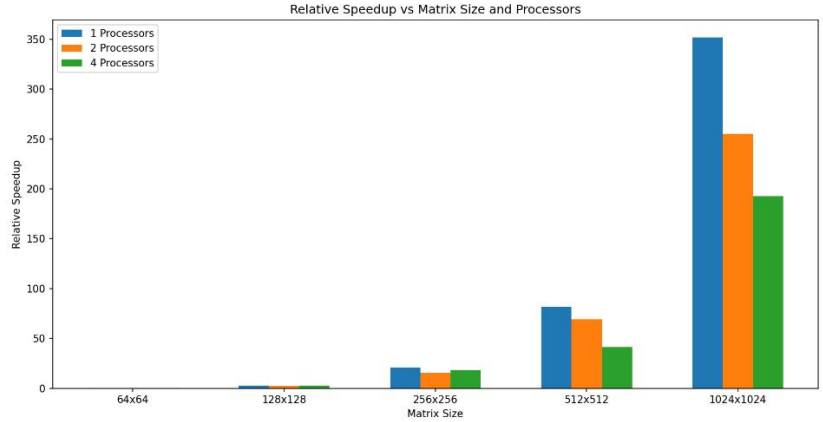
加速比的理论峰值为：matrix_sz^2，即每个线程并行计算结果矩阵的一个元素。

加速比表格如下：

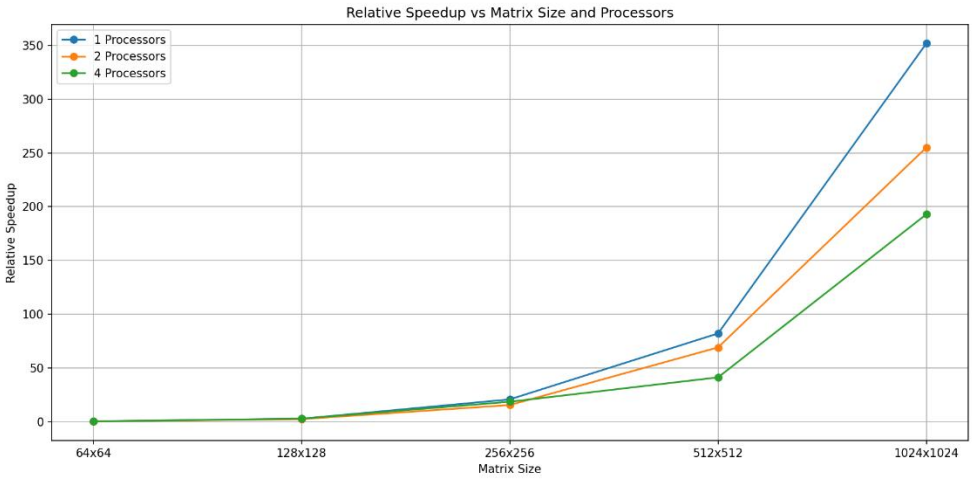
matrix_sz/cpuorgpu	cpu	proc:1	proc:2	proc:4
64x64	1	0.307829181	0.263719512	0.198850575
128x128	1	2.660211268	2.303353659	2.777573529
256x256	1	20.70477569	15.45032397	18.48449612
512x512	1	82.06031746	69.04113248	41.18706182
1024x1024	1	351.8362824	255.0166853	192.8473635

加速比随着 MPI 进程数增大而减小，随着数据量增大而变大。

绘制加速比柱状如下：



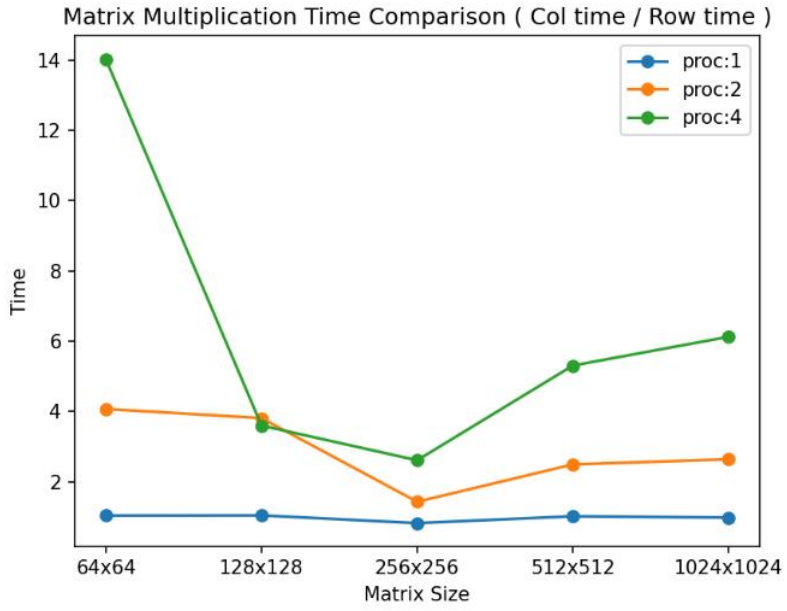
绘制曲线图如下：



可以发现，当 MPI 进程数变大时，加速比反而减小，原因可能是 MPI_Reduce 操作是一个集体通信操作，它需要在所有进程之间进行通信，将局部结果合并为全局结果。通信成本的增加可能会抵消并行计算带来的加速，尤其是在进程数较多的情况下。

5.3.3 横向对比

将 A 按列划分所花费的时间除以 A 按行划分计算所花费的时间得到图像如下，当进程数为 1 时，时间比为 1，这两种算法的本质是相同的，也说明了**代码的正确性。**



但是 A 按列划分在进程数增加时相比 A 按行划分性能更差，原因可能是：

1、 单个进程的结果矩阵的每一个元素所涉及的输入 A 和 B 矩阵的元素更少（其他的元素给到其他进程了），并没有很好地利用 A 矩阵元素存储的空间局部性和计算的连续性。

2、 多个进程的结果需要作一个 Reduce(AllReduce)操作，这个操作需要每个进程(GPU)都和根进程通信，切分地越多，通讯量就越大。

6. 实验结论

本次实验共用时约 50h。

本次实验成功实现了使用 OpenMP 框架编程的并行快速排序算法（PQS）以及使用 MPI 实现的 Parallel Sorting by Regular Sampling（PSRS）算法。以下是实验结论的总结：

1. 正确性验证： 通过对比并行算法和串行快速排序算法的结果，确认它们产生相同的有序数组，从而验证了并行算法的正确性。这为进一步的性能分析提供了基础。

2. 性能度量： 在并行算法中，通过记录程序的运行时间，得到了并行快速排序算法和 PSRS 算法在不同处理器数量和数组大小下的性能表现。通过计算加速比，我们可以清晰地了解并行算法在多处理器环境中的性能提升。

3. 实验结果： 根据具体的实验数据，观察到在给定的条件下，如不同数组大小和处理器数量，程序表现出了一定的性能差异。这些数据对于评估算法的实际效果和选择合适的算法提供了重要依据。

4. 可伸缩性： 通过在 MPI 和 OpenMP 算法中测试不同处理器数量下的性能，评估了算法的可伸缩性。实验结果表明，随着处理器数量的增加，加速比逐渐提高，但在一定点后可能出现饱和，这提示在选择处理器数量时需要权衡。

通过对 MPI 和 CUDA 混合并行矩阵乘法的实现以及性能测试，我们得到了以下结论：

1. 切分方式影响性能： 两种切分方式（A 按行切分，B 不切分 vs. A 按列切分，B 按行切分）在性能上表现出差异。从实验数据中可以观察到，A 按列切分在增加 MPI 进程数时性能反而下降，可能是因为通信成本的增加抵消了并行计算的优势。

2. 通信开销： MPI_Reduce 操作作为一个集体通信操作，需要在所有进程之间进行通信，将局部结果合并为全局结果。在 A 按列切分的情况下，通信开销更大，可能导致性能下降。

3. 并行加速比： 在适当的情况下，MPI 与 CUDA 混合并行矩阵乘法能够实现显著的性能提升。特别是在数据量较大时，加速比明显增加，说明并行计算在大规模问题上有优势。

4. 横向对比： 通过对 A 按列划分所花费的时间与 A 按行划分所花费的时间进行对比，我们可以看到两者的性能差异。这对比也说明了实验代码的正确性，因为在单进程的情况下，两种划分方式的时间比为 1。

5. 优化空间： 实验中我们注意到性能可以通过调整切分方式、优化通信等手段进一步提高。这提示我们在实际应用中需要综合考虑切分方式、通信开销等因素，以达到最佳性能。

总而言之，这次实验加深了我对于排序算法、矩阵乘法、并行计算以及如何在并行的基础上优化的理解，锻炼了我对于 MPI、OpenMP 和 CUDA 等的架构的编程能力，同时也对以后科研工作经常会接触的大模型如何并行化训练有了一定的了解，算是一次很有收获的打基础过程。

项目已开源至 https://github.com/JackIE0niden/ParallelComputing2023。

7. 参考文献

[1] SHOEYBI M, PATWARY M, PURI R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism [J]. arXiv preprint arXiv:190908053, 2019.