并行与分布式计算作业

第六次作业

姓名：施天予

班级：计科2班

学号：19335174

一、问题描述

1. 找出给出的7个MPI程序中的bug，并将程序修改正确，比较修改前后的程序运行结果。MPI程序见附件。
2. 利用MPI程序实现 n 个整形数的排序过程，排序算法不限， MPI的进程不限，可以使用单机运行多个MPI进程。提示： 在排序的过程中需要MPI集合通信如MPI_Allgather等。

二、解决方案

修改MPI程序的bug

mpi_bug1

两个进程 tag = rank，发送与接收的tag不一致，导致消息无法发送成功，进程阻塞

```
    if (rank == 0) {
        if (numtasks > 2)
            printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
        dest = rank + 1;
        source = dest;
        tag = rank;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        printf("Sent to task %d...\n",dest);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        printf("Received from task %d...\n",source);
    }
    else if (rank == 1) {
        dest = rank - 1;
        source = dest;
        tag = rank;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        printf("Received from task %d...\n",source);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        printf("Sent to task %d...\n",dest);
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug1
Task 0 starting...
Sent to task 1...
Task 1 starting...
```

设置 tag = 0，消息发送成功

```
    if (rank == 0) {
```

```
        if (numtasks > 2)
            printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
        dest = rank + 1;
        source = dest;
        tag = 0;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        printf("Sent to task %d...\n",dest);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        printf("Received from task %d...\n",source);
    }
    else if (rank == 1) {
        dest = rank - 1;
        source = dest;
        tag = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        printf("Received from task %d...\n",source);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        printf("Sent to task %d...\n",dest);
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug1
Task 0 starting...
Sent to task 1...
Received from task 1...
Task 0: Received 1 char(s) from task 1 with tag 0
Task 1 starting...
Received from task 0...
Sent to task 0...
Task 1: Received 1 char(s) from task 0 with tag 0
```

mpi_bug2

变量alpha是int类型，变量beta是float类型，发送消息时它们的数据类型MPI_INT和MPI_FLOAT不一致

```
    int numtasks, rank, tag=1, alpha, i;
    float beta;
    if (rank == 0) {
        if (numtasks > 2)
            printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
        for (i=0; i<10; i++) {
            alpha = i*10;
            MPI_Isend(&alpha, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &reqs[i]);
            MPI_Wait(&reqs[i], &stats[i]);
            printf("Task %d sent = %d\n",rank,alpha);
        }
    }
    if (rank == 1) {
        for (i=0; i<10; i++) {
            MPI_Irecv(&beta, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &reqs[i]);
            MPI_Wait(&reqs[i], &stats[i]);
```

```
            printf("Task %d received = %f\n",rank,beta);
        }
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug2
Task 0 sent = 0
Task 0 sent = 10
Task 0 sent = 20
Task 0 sent = 30
Task 0 sent = 40
Task 0 sent = 50
Task 0 sent = 60
Task 0 sent = 70
Task 0 sent = 80
Task 0 sent = 90
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
Task 1 received = 0.000000
```

将alpha和beta改为float类型，设置MPI_Isend和MPI_Irecv的数据类型是MPI_FLOAT

```
    int numtasks, rank, tag=1, i;
    float alpha, beta;
    if (rank == 0) {
        if (numtasks > 2)
            printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
        for (i=0; i<10; i++) {
            alpha = i*10;
            MPI_Isend(&alpha, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &reqs[i]);
            MPI_Wait(&reqs[i], &stats[i]);
            printf("Task %d sent = %f\n",rank,alpha);
        }
    }
    if (rank == 1) {
        for (i=0; i<10; i++) {
            MPI_Irecv(&beta, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &reqs[i]);
            MPI_Wait(&reqs[i], &stats[i]);
            printf("Task %d received = %f\n",rank,beta);
        }
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug2
Task 0 sent = 0.000000
Task 0 sent = 10.000000
Task 0 sent = 20.000000
Task 0 sent = 30.000000
Task 0 sent = 40.000000
Task 0 sent = 50.000000
Task 0 sent = 60.000000
Task 0 sent = 70.000000
Task 0 sent = 80.000000
Task 0 sent = 90.000000
Task 1 received = 0.000000
Task 1 received = 10.000000
Task 1 received = 20.000000
Task 1 received = 30.000000
Task 1 received = 40.000000
Task 1 received = 50.000000
Task 1 received = 60.000000
Task 1 received = 70.000000
Task 1 received = 80.000000
Task 1 received = 90.000000
```

mpi_bug3

未使用MPI_Init()和MPI_Finalize()进行初始化和终止化

```
    /***** Initializations *****/
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    ...
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug3
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH
```

在开头添加MPI_Init()，结尾添加MPI_Finalize()

```
    /***** Initializations *****/
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    ...
    MPI_Finalize();
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug3
MPI task 0 has started...
MPI task 1 has started...
MPI task 2 has started...
MPI task 3 has started...
Initialized array sum = 1.335708e+14
Sent 4000000 elements to task 1 offset= 4000000
Sent 4000000 elements to task 2 offset= 8000000
Sent 4000000 elements to task 3 offset= 12000000
Task 1 mysum = 4.884048e+13
Task 2 mysum = 7.983003e+13
Task 0 mysum = 1.598859e+13
Task 3 mysum = 1.161867e+14
Sample results:
  0.000000e+00  2.000000e+00  4.000000e+00  6.000000e+00  8.000000e+00
  8.000000e+06  8.000002e+06  8.000004e+06  8.000006e+06  8.000008e+06
  1.600000e+07  1.600000e+07  1.600000e+07  1.600001e+07  1.600001e+07
  2.400000e+07  2.400000e+07  2.400000e+07  2.400001e+07  2.400001e+07
*** Final sum= 2.608458e+14 ***
```

mpi_bug4

在MASTER进程中未使用MPI_Reduce，导致Final sum结果错误

```
    /* Get final sum and print sample results */
    printf("Sample results: \n");
    offset = 0;
    for (i=0; i<numtasks; i++) {
        for (j=0; j<5; j++)
            printf("  %e",data[offset+j]);
            printf("\n");
            offset = offset + chunksize;
    }
    printf("*** Final sum= %e ***\n",sum);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug4
MPI task 0 has started...
MPI task 1 has started...
MPI task 2 has started...
MPI task 3 has started...
Initialized array sum = 1.335708e+14
Sent 4000000 elements to task 1 offset= 4000000
Sent 4000000 elements to task 2 offset= 8000000
Task 1 mysum = 4.884048e+13
Sent 4000000 elements to task 3 offset= 12000000
Task 2 mysum = 7.983003e+13
Task 3 mysum = 1.161867e+14
Task 0 mysum = 1.598859e+13
Sample results:
  0.000000e+00  2.000000e+00  4.000000e+00  6.000000e+00  8.000000e+00
  8.000000e+06  8.000002e+06  8.000004e+06  8.000006e+06  8.000008e+06
  1.600000e+07  1.600000e+07  1.600000e+07  1.600001e+07  1.600001e+07
  2.400000e+07  2.400000e+07  2.400000e+07  2.400001e+07  2.400001e+07
*** Final sum= 1.335708e+14 ***
```

在MASTER进程中添加MPI_Reduce，Final sum结果正确

```
    /* Get final sum and print sample results */
    MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD);
    printf("Sample results: \n");
    offset = 0;
    for (i=0; i<numtasks; i++) {
        for (j=0; j<5; j++)
            printf("  %e",data[offset+j]);
            printf("\n");
            offset = offset + chunksize;
    }
    printf("*** Final sum= %e ***\n",sum);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug4
MPI task 1 has started...
MPI task 3 has started...
MPI task 0 has started...
MPI task 2 has started...
Initialized array sum = 1.335708e+14
Sent 4000000 elements to task 1 offset= 4000000
Task 1 mysum = 4.884048e+13
Sent 4000000 elements to task 2 offset= 8000000
Task 2 mysum = 7.983003e+13
Sent 4000000 elements to task 3 offset= 12000000
Task 3 mysum = 1.161867e+14
Task 0 mysum = 1.598859e+13
Sample results:
  0.000000e+00  2.000000e+00  4.000000e+00  6.000000e+00  8.000000e+00
  8.000000e+06  8.000002e+06  8.000004e+06  8.000006e+06  8.000008e+06
  1.600000e+07  1.600000e+07  1.600000e+07  1.600001e+07  1.600001e+07
  2.400000e+07  2.400000e+07  2.400000e+07  2.400001e+07  2.400001e+07
*** Final sum= 2.608458e+14 ***
```

mpi_bug5

0号进程使用MPI_Send向1号进程不停发送，默认是缓冲模式，导致1号进程最后会缓冲区耗尽，不是“线程安全”的。因此每次的Time是不均匀的。

```
    if (rank == 0) {
        /* Initialize send data */
        for(i=0; i<MSGSIZE; i++)
          data[i] =  'x';
        start = MPI_Wtime();
        while (1) {
            MPI_Send(data, MSGSIZE, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
            count++;
            if (count % 10 == 0) {
                end = MPI_Wtime();
                printf("Count= %d  Time= %f sec.\n", count, end-start);
                start = MPI_Wtime();
            }
        }
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug5
mpi_bug5 has started...
Count= 10  Time= 0.000005 sec.
Count= 20  Time= 0.000003 sec.
Count= 30  Time= 0.000003 sec.
Count= 40  Time= 0.000004 sec.
Count= 50  Time= 0.000003 sec.
Count= 60  Time= 0.000004 sec.
Count= 70  Time= 0.032887 sec.
Count= 80  Time= 0.070880 sec.
Count= 90  Time= 0.062464 sec.
Count= 100  Time= 0.074145 sec.
Count= 110  Time= 0.061287 sec.
Count= 120  Time= 0.062539 sec.
Count= 130  Time= 0.069653 sec.
Count= 140  Time= 0.063584 sec.
Count= 150  Time= 0.067359 sec.
Count= 160  Time= 0.061187 sec.
Count= 170  Time= 0.061724 sec.
Count= 180  Time= 0.071414 sec.
Count= 190  Time= 0.065007 sec.
Count= 200  Time= 0.061337 sec.
Count= 210  Time= 0.061488 sec.
Count= 220  Time= 0.063138 sec.
Count= 230  Time= 0.065432 sec.
Count= 240  Time= 0.061484 sec.
Count= 250  Time= 0.061072 sec.
Count= 260  Time= 0.061335 sec.
```

将MPI_Send改为MPI_Ssend（同步模式），保证了对应的MPI_Recv开始前发送端一直阻塞，是“线程安全”的。因此每次的Time都大致相同。

```
    if (rank == 0) {
        /* Initialize send data */
        for(i=0; i<MSGSIZE; i++)
          data[i] =  'x';
        start = MPI_Wtime();
        while (1) {
            MPI_Ssend(data, MSGSIZE, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
            count++;
            if (count % 10 == 0) {
                end = MPI_Wtime();
                printf("Count= %d  Time= %f sec.\n", count, end-start);
                start = MPI_Wtime();
            }
        }
    }
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 2 ./mpi_bug5
mpi_bug5 has started...
Count= 10  Time= 0.064866 sec.
Count= 20  Time= 0.062370 sec.
Count= 30  Time= 0.068363 sec.
Count= 40  Time= 0.063164 sec.
Count= 50  Time= 0.066751 sec.
Count= 60  Time= 0.061074 sec.
Count= 70  Time= 0.061657 sec.
Count= 80  Time= 0.061410 sec.
Count= 90  Time= 0.061951 sec.
Count= 100  Time= 0.062274 sec.
Count= 110  Time= 0.061599 sec.
Count= 120  Time= 0.061462 sec.
Count= 130  Time= 0.061577 sec.
Count= 140  Time= 0.062166 sec.
Count= 150  Time= 0.064699 sec.
Count= 160  Time= 0.062542 sec.
Count= 170  Time= 0.062068 sec.
Count= 180  Time= 0.061367 sec.
Count= 190  Time= 0.063559 sec.
Count= 200  Time= 0.064562 sec.
Count= 210  Time= 0.063454 sec.
Count= 220  Time= 0.062775 sec.
```

mpi_bug6

rank=1的offset=REPS使reqs的索引重叠，导致MPI_Waitall报错；rank=2的nreqs=REPS错误，因为MPI_Send是阻塞式的，不需要请求数量，导致MPI_Waitall报错。

```
    if (rank < 2) {
        nreqs = REPS*2;
        if (rank == 0) {
            src = 1;
            offset = 0;
        }
        if (rank == 1) {
            src = 0;
            offset = REPS;
        }
        dest = src;
        for (i=0; i<REPS; i++) {
            MPI_Isend(&rank, 1, MPI_INT, dest, tag1, COMM, &reqs[offset]);
            MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset+1]);
            offset += 2;
            if ((i+1)%DISP == 0)
                printf("Task %d has done %d isends/irecvs\n", rank, i+1);
        }
    }
    if (rank > 1) {
        nreqs = REPS;
        if (rank == 2) {
            dest = 3;
            for (i=0; i<REPS; i++) {
                MPI_Send(&rank, 1, MPI_INT, dest, tag1, COMM);
```

```
                if ((i+1)%DISP == 0)
                    printf("Task %d has done %d sends\n", rank, i+1);
            }
        }
        if (rank == 3) {
            src = 2;
            offset = 0;
            for (i=0; i<REPS; i++) {
                MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset]);
                offset += 1;
                if ((i+1)%DISP == 0)
                    printf("Task %d has done %d irecvs\n", rank, i+1);
            }
        }
    }
    MPI_Waitall(nreqs, reqs, stats);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug6
Starting isend/irecv send/irecv test...
Task 0 starting...
Task 1 starting...
Task 2 starting...
Task 3 starting...
Task 1 has done 100 isends/irecvs
Task 1 has done 200 isends/irecvs
Task 1 has done 300 isends/irecvs
Task 1 has done 400 isends/irecvs
Task 3 has done 100 irecvs
Task 3 has done 200 irecvs
Task 3 has done 300 irecvs
Task 1 has done 500 isends/irecvs
Task 3 has done 400 irecvs
Task 3 has done 500 irecvs
Task 3 has done 600 irecvs
Task 3 has done 700 irecvs
Task 3 has done 800 irecvs
Task 3 has done 900 irecvs
Task 3 has done 1000 irecvs
Task 0 has done 100 isends/irecvs
Task 2 has done 100 sends
Task 2 has done 200 sends
Task 2 has done 300 sends
Task 2 has done 400 sends
Task 2 has done 500 sends
Task 2 has done 600 sends
Task 2 has done 700 sends
Task 2 has done 800 sends
Task 2 has done 900 sends
Task 0 has done 200 isends/irecvs
Task 0 has done 300 isends/irecvs
Task 0 has done 400 isends/irecvs
Task 0 has done 500 isends/irecvs
Task 2 has done 1000 sends
Fatal error in PMPI_Waitall: Request pending due to failure, error stack:
PMPI_Waitall(356): MPI_Waitall(count=1000, req_array=0x7ffdc54c9dd0, status_array=0x7ffdc54cbd10) failed
PMPI_Waitall(332): The supplied request in array element 0 was invalid (kind=0)
```

修改rank=1的offset=0，修改rank=2的nreqs=0

```
    if (rank < 2) {
        nreqs = REPS*2;
        if (rank == 0) {
            src = 1;
            offset = 0;
        }
        if (rank == 1) {
            src = 0;
```

```
            offset = 0;
        }
        dest = src;
        for (i=0; i<REPS; i++) {
            MPI_Isend(&rank, 1, MPI_INT, dest, tag1, COMM, &reqs[offset]);
            MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset+1]);
            offset += 2;
            if ((i+1)%DISP == 0)
                printf("Task %d has done %d isends/irecvs\n", rank, i+1);
        }
    }
    if (rank > 1) {
        if (rank == 2) {
            nreqs = 0;
            dest = 3;
            for (i=0; i<REPS; i++) {
                MPI_Send(&rank, 1, MPI_INT, dest, tag1, COMM);
                if ((i+1)%DISP == 0)
                    printf("Task %d has done %d sends\n", rank, i+1);
            }
        }
        if (rank == 3) {
            nreqs = REPS;
            src = 2;
            offset = 0;
            for (i=0; i<REPS; i++) {
                MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset]);
                offset += 1;
                if ((i+1)%DISP == 0)
                    printf("Task %d has done %d irecvs\n", rank, i+1);
            }
        }
    }
    MPI_Waitall(nreqs, reqs, stats);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug6
Starting isend/irecv send/irecv test...
Task 0 starting...
Task 1 starting...
Task 2 starting...
Task 3 starting...
Task 1 has done 100 isends/irecvs
Task 3 has done 100 irecvs
Task 3 has done 200 irecvs
Task 3 has done 300 irecvs
Task 3 has done 400 irecvs
Task 3 has done 500 irecvs
Task 3 has done 600 irecvs
Task 3 has done 700 irecvs
Task 3 has done 800 irecvs
Task 3 has done 900 irecvs
Task 3 has done 1000 irecvs
Task 1 has done 200 isends/irecvs
Task 1 has done 300 isends/irecvs
Task 1 has done 400 isends/irecvs
Task 1 has done 500 isends/irecvs
Task 2 has done 100 sends
Task 2 has done 200 sends
Task 2 has done 300 sends
Task 2 has done 400 sends
Task 2 has done 500 sends
Task 0 has done 100 isends/irecvs
Task 2 has done 600 sends
Task 2 has done 700 sends
Task 2 has done 800 sends
Task 2 has done 900 sends
Task 2 has done 1000 sends
Task 0 has done 200 isends/irecvs
Task 0 has done 300 isends/irecvs
Task 0 has done 400 isends/irecvs
Task 1 has done 600 isends/irecvs
Task 1 has done 700 isends/irecvs
Task 1 has done 800 isends/irecvs
Task 1 has done 900 isends/irecvs
Task 1 has done 1000 isends/irecvs
Task 0 has done 500 isends/irecvs
Task 0 has done 600 isends/irecvs
Task 0 has done 700 isends/irecvs
Task 0 has done 800 isends/irecvs
Task 0 has done 900 isends/irecvs
Task 0 has done 1000 isends/irecvs
Task 0 time(wall)= 0.002597 sec
Task 1 time(wall)= 0.002593 sec
Task 2 time(wall)= 0.000807 sec
Task 3 time(wall)= 0.000813 sec
```

mpi_bug7

buffer只有 1 个条目，而不是等于 rank 的条目数

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
count = taskid;
MPI_Bcast(&buffer, count, MPI_INT, root, MPI_COMM_WORLD);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug7
Task 0 on ubuntu starting...
Root: Number of MPI tasks is: 4
Task 1 on ubuntu starting...
Task 2 on ubuntu starting...
Task 3 on ubuntu starting...
```

在 MPI_Bcast 中将count参数改为 1

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
count = taskid;
MPI_Bcast(&buffer, 1, MPI_INT, root, MPI_COMM_WORLD);
```

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./mpi_bug7
Task 0 on ubuntu starting...
Root: Number of MPI tasks is: 4
Task 1 on ubuntu starting...
Task 2 on ubuntu starting...
Task 3 on ubuntu starting...
sty@ubuntu:~/Parallel/hw6/new$
```

MPI奇偶交换排序

串行算法

这次实验我实现的是奇偶交换排序，参考了《并行程序设计导论》中的写法。此算法由一系列阶段组成，在偶数阶段，比较-交换由以下数对执行：

（a[0]，a[1]），（a[2]，a[3]），…

奇数阶段由以下数对进行比较-交换：

（a[1]，a[2]），（a[3]，a[4]），…

我们至多用n个阶段就能排序n个元素。串行奇偶交换排序的代码如下：

```
void odd_even_sort(int *a, int n) {
    int phase, i, temp;
    for (phase = 0; phase < n; ++phase) {
        if (phase % 2 == 0) {
            for (int i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        }
        else {
            for (int i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
    }
}
```

并行算法

因为在一个阶段内所有的比较-交换能同时进行，奇偶交换排序非常易于并行化。根据阶段，进程i可以将当前自己拥有a[i]的值发送给进程i-1或者进程i+1。同时，它也应该接收存储在进程i-1或者i+1的值，然后决定保留两个值中的哪个来为下一阶段做准备。如果由p个进程运行并行奇偶交换排序算法，则p个阶段后，排序完毕。

时间	进程			
	0	1	2	3
开始	15，11，9，16	3，14，8，7	4，6，12，10	5，2，13，1
局部排序后	9，11，15，16	3，7，8，14	4，6，10，12	1，2，5，13
阶段 0 后	3，7，8，9	11，14，15，16	1，2，4，5	6，10，12，13
阶段 1 后	3，7，8，9	1，2，4，5	11，14，15，16	6，10，12，13
阶段 2 后	1，2，3，4	5，7，8，9	6，10，11，12	13，14，15，16
阶段 3 后	1，2，3，4	5，6，7，8	9，10，11，12	13，14，15，16

首先获取进程数comSize和进程号comRank，通过0号进程读取元素个数n，随机生成n个小于n的整型数

```
    MPI_Init(&argc, &argv);
    int comSize, comRank, n, *globalData;
    MPI_Comm_size(MPI_COMM_WORLD, &comSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &comRank);

    if (comRank == 0) {
        printf("Input array size:");
        scanf("%d", &n);
        globalData = malloc(n * sizeof(int));
        for (int i = 0; i < n; ++i)
            globalData[i] = rand() % n;
    }
```

接下来使用MPI_Bcast将元素个数n广播到所有进程，再把n个元素组成的数组globalData通过MPI_Scatterv将数组分成几段分发给各个进程。这里有几个参数需要说明：

- globalData：n个元素的数组
- localData：每个进程分得的数组
- sendcounts：每个进程需要发送的数据量，如 sendcount[1] = 3 代表需要发送3个元素到1号进程
- displs：每个进程发送的数据的开始索引，如 displs[1] = 5 代表需要从globalData[5]开始发送数据到1号进程

```
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int *sendcounts = malloc(comSize * sizeof(int));
    int *displs = malloc(comSize * sizeof(int));
    for (int i = 0, cnt = (n + comSize - 1) / comSize; i < comSize; ++i) {
        displs[i] = i * cnt;
        sendcounts[i] = n - displs[i] < cnt ? n - displs[i] : cnt;
    }
    int *localData = malloc((sendcounts[comRank] + sendcounts[0]) * sizeof(int));
    MPI_Scatterv(globalData, sendcounts, displs, MPI_INT, localData,
sendcounts[comRank], MPI_INT, 0, MPI_COMM_WORLD);
    qsort(localData, sendcounts[comRank], sizeof(int), &cmp);
```

奇偶交换排序，首先要根据阶段的奇偶性确定partner的进程号，然后每个进程与它的partner进程使用MPI_Sendrecv交换localData，使用qsort将两个进程内的元素进行排序，再把前一半元素交给进程号小的进程，后一半元素交给进程号大的进程。经过几个阶段的比较-交换，最后通过MPI_Gatherv将所有进程的数据聚集到0号进程，整个排序过程就完成了。

```
    for (int i = 0; i < comSize; ++i) {
        int partner = comRank + ((i + comRank) % 2 ? 1 : -1);
        if (0 <= partner && partner < comSize) {
            MPI_Sendrecv(localData, sendcounts[comRank], MPI_INT, partner, i,
localData + sendcounts[comRank], sendcounts[partner], MPI_INT, partner, i,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            qsort(localData, sendcounts[comRank] + sendcounts[partner], sizeof(int),
&cmp);
            if (comRank > partner)
                for (int i = 0; i < sendcounts[comRank]; ++i)
                    localData[i] = localData[i + sendcounts[partner]];
        }
    }
    MPI_Gatherv(localData, sendcounts[comRank], MPI_INT, globalData, sendcounts,
displs, MPI_INT, 0, MPI_COMM_WORLD);
    free(localData), free(sendcounts), free(displs);
    MPI_Finalize();
}
```

正确性检验

通过对100个数进行排序，分别检验串行算法和MPI并行算法的正确性

串行算法

```
sty@ubuntu:~/Parallel/hw6/new$ ./sort_serial
Input array size:100
Running time: 0.000029 second
2 3 5 5 8 11 11 12 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 26 27 27 29 2
9 29 29 30 32 34 35 35 36 36 37 39 39 40 42 43 45 46 49 50 51 54 56 56 57 58 59
60 62 62 62 63 64 67 67 67 67 68 68 69 70 70 72 73 73 76 76 77 78 80 81 82 82 83
 84 84 84 86 86 86 87 88 90 91 92 93 93 94 95 96 98 99
```

并行算法

```
sty@ubuntu:~/Parallel/hw6/new$ mpiexec -n 4 ./sort_mpi
Input array size:100
Running time: 0.000056 second
2 3 5 5 8 11 11 12 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 26 27 27 29 2
9 29 29 30 32 34 35 35 36 36 37 39 39 40 42 43 45 46 49 50 51 54 56 56 57 58 59
60 62 62 62 63 64 67 67 67 67 68 68 69 70 70 72 73 73 76 76 77 78 80 81 82 82 83
 84 84 84 86 86 86 87 88 90 91 92 93 93 94 95 96 98 99
```

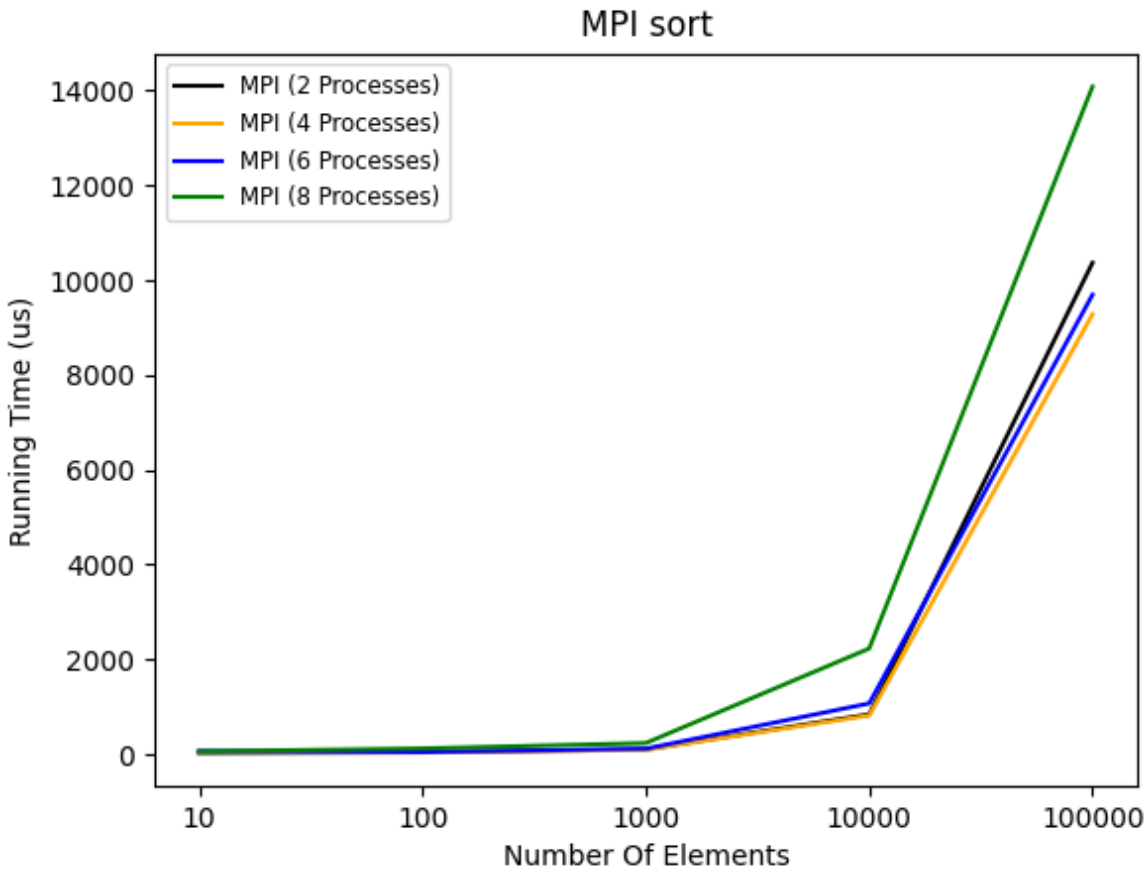
发现奇偶交换排序的串行算法和并行算法结果都是正确的

结果分析

分别对串行算法和MPI的不同进程数进行运行时间的比较，其中列索引是排序的元素个数，时间单位是微秒。得到结果如下

	10	**100**	**1000**	**10000**	**100000**
Serial	2	37	2745	176617	19852328
MPI (2 Processes)	37	46	102	842	10368
MPI (4 Processes)	48	39	95	818	9281
MPI (6 Processes)	69	46	118	1073	9694
MPI (8 Processes)	61	123	237	2230	14084

对MPI不同进程数排序算法的运行时间，绘图如下：



实验发现，我实现的MPI奇偶交换排序在4个进程时加速效果达到最佳，与串行算法相比，其在100000个元素排序时加速比可以达到2139，大大减小了程序运行的时间；然而MPI在排序数量较小时效果不如串行算法，因为其不同进程间的通信开销导致运行时间的增加，但排序数量较大时通信开销相比排序的时间就显得微乎其微了。

三、遇到的问题及解决方法

这次实验相对来说还比较轻松。debug部分主要是bug5和bug6略有困难：bug5开始没看懂它的意思，后来发现是发送太快导致来不及接收后问题就解决了；bug6开始不理解MPI_Waitall为何报错，仔细观察后发现是因为MPI_Send本身就是阻塞式的，所以可以不需要wait。然而其实整个debug的过程我花费时间最多的还是修改程序的缩进，它本身乱七八糟的缩进让人十分难受。

MPI排序算法包括Scatter分发数据、Gather整合数据等多种集合通信操作，通过自己对MPI程序的编写，让我对MPI各种函数的使用更加熟练了，也对MPI集合通信的原理有了更深刻的理解，收回颇丰。