Parallel Programming

Basic & Advanced ODD-Even Sort

Peter Huang

1、 The flow of Implementation

Basic version:

Step1、Each Processors get data with the number: total data / total processors

Step2、Decide if the first and the last data in each process should exchange data with other neighboring processors.

Judge Pattern: I use data's index in total data to judge it. For example: in Even exchange, if the index of the first data is odd, the first data will receive data form its front processor, which means there are communications between itself and its front neighboring processor. If the index of first data is even, the first data will not exchange data with the front process, which means that the exchange actions happen in its processor only. Besides, if the index of the last data is even, it should send data to its next processor. If the index of last data is odd, the exchange operation will happen within the operation of the processor only.

Step3、Communicate with its front and next processors once the exchanged index is true in the first and last data.

Step4、Compare with the rest of data which also contains the first and last data whose the exchanged index is false.

Step5、Judge if there are any exchange action happens from step 2 to step 4. Recording if there are any exchange action. The program only finishes sorting data by detecting the number of non-exchanging action. If the number of non-exchanging action is equals with 2, I stop exchange action.

Step6、 After step2 through step5, I have successfully exchange one even or odd exchange's action. The program iteratively goes through Step2 ~ Step5

to complete sorting action.

Advanced version:

Step1、Each Processors get data with the number: total data / total processors

Step2、Sort data in each processor by qsort functions

Step3、Send all data to its next processors

Step4、Once the next processor receives data, it will merge data with the order from small integer to the large integer.

Step5、Send back sorting data to the original processor

Step6、Judge if there are any exchange action happens from step 5 by the smallest integer and the largest integer. Recording if there are any exchange action. The program only finishes sorting data by detecting the number of non-exchanging action. If the number of non-exchanging action is equals with 2, we stop exchange action, which means the sorting action is done.

Step7 、After step3 through step6, the program have successfully exchanged one even or odd exchange's action. It iteratively goes through Step3 ~ Step6, until sorting action is done.

2、 Experiment And Analysis

2.1 System Spec

Run programs in parallel programming's testing environment.

2.2 Strong scalability & Time distribution

Which way is used to get the consuming time in each action?

Use clock() And gettimeofday() function to record the consuming time. In any I/O, MPI communicating functions like send or receive, etc., I use the following way to get the consuming time and accumulate each consuming time in each actions. Finally, I can get total consuming time in each action like I/O or Processors communication time.

In the program I use MPI_Barrier(MPI_COMM_WORLD) to make sure that each processor could step in the same line of programs in the Even or Odd exchange action. So by the way, each processor should wait for other processors, if others do not execute the line MPI_Barrier. Hence, the functions in each processor should be synchronized. So the time interval that each actions takes like I/O and communication time in each processor should be closed with other processors, which means no matter processors you choose to recording the time interval should not have too much difference.

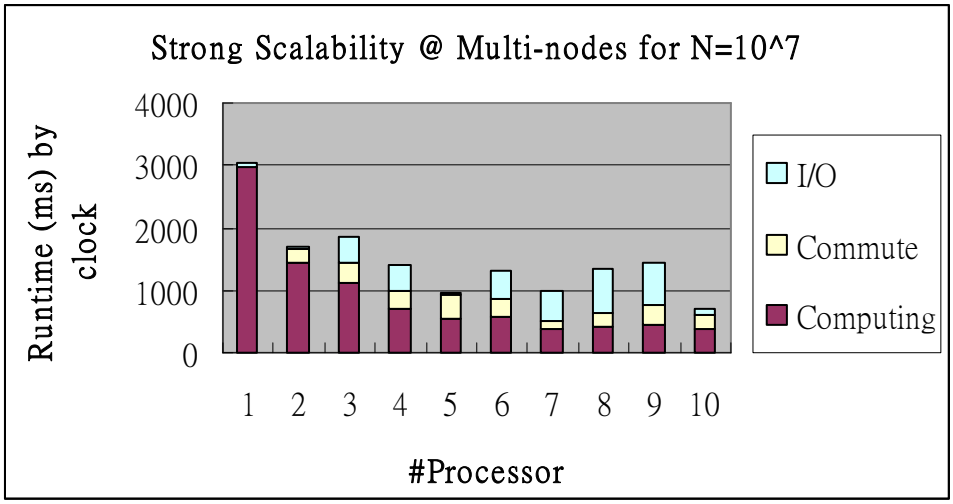
The following is the way by which I get the consuming time of func_A().

```
clock_t startTime = clock();
func_A();
clock_t endTime = clock();
double consumingTime = (double)( endTime - startTime);
-----------------------------------------------------------------------------------------------
struct timeval startTime, endTime;
gettimeofday(&startTime,NULL);
func_A();
gettimeofday(&endTime,NULL);
double consumingTime    = endT.tv_sec-startT.tv_sec +
(endT.tv_usec-startT.tv_usec) /1000000.0; // sec
```

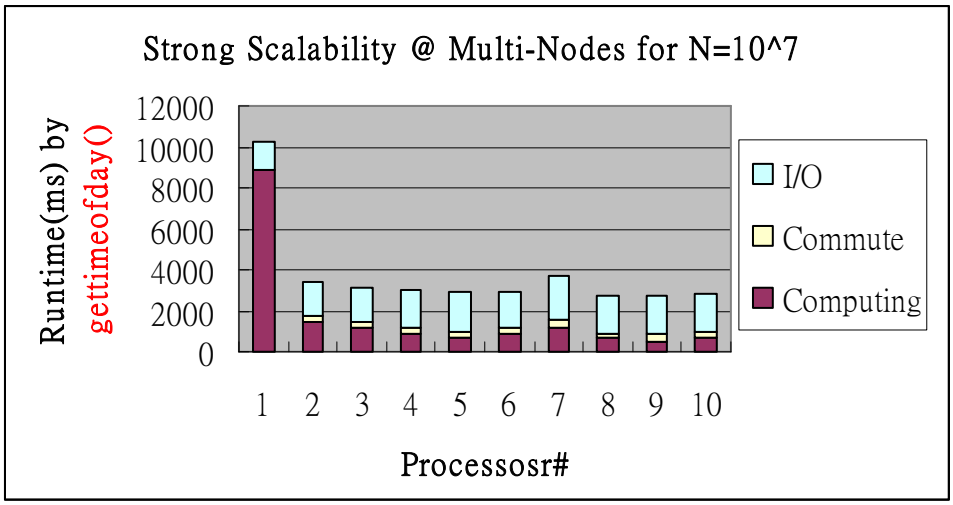
{Multi-nodes , single-node } X {basic, advanced version}

[Advanced version] **N = 10^7**

Multi-Node: I choose different nodes to run the program. (There are 10 nodes in parallel programming's test machine.)



(ms) / #No	1	2	3	4	5	6	7	8	9	10
Computing	2960	1430	1120	690	530	580	390	420	430	390
Communicating	0	220	320	310	390	280	130	200	340	210
I/O	80	50	410	400	50	460	460	710	650	80



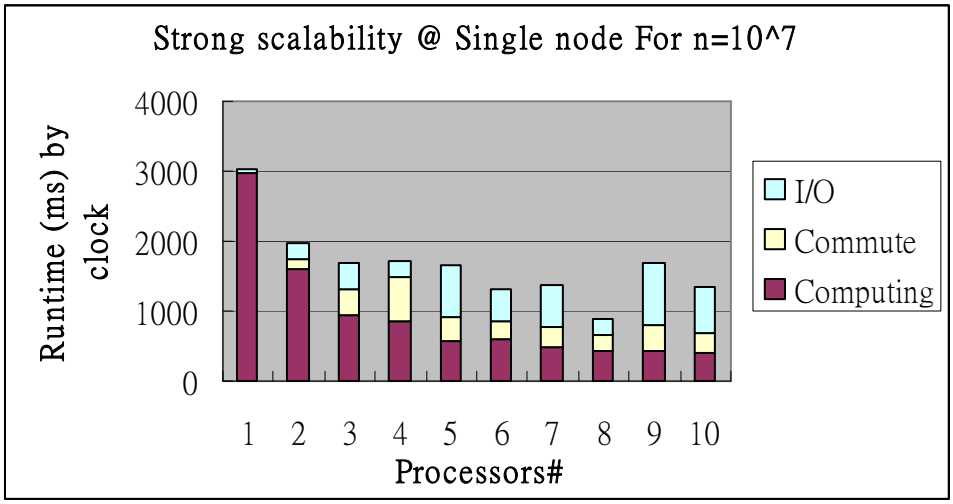
(ms) / #No	1	2	3	4	5	6	7	8	9	10
Computing	8868	1486	1161	829	634	835	1162	673	497	687
Communicating	0	221	320	322	376	292	397	187	339	236
I/O	1367	1694	1678	1855	1873	1834	2169	1854	1915	1910

Try to comparing the gettimeofday() and clock() functions:

gettimeofday function will get real escaped time in the computers, which means that it will record the real time period. However, the clock function will get the cpu's clock signal to measure the time. By the experiments, we can see the I/O & computing time has increased largely by using gettimeofday measurement. My thought is that the largely increased time when using gettimeofday is due to the system switches to do other jobs or our programs has been switch out form running stage. So in the some time the cpu is allocated to other jobs. Hence, if we just records by clock(), it will depend on cpu's clock signal to tell us how long the cpu needs to take in some functions in our programs. However, to be honest with you, it's just my idea but I don't know if my thought is right or not.

[Advanced version]

Single-node: I choose only one node to run the program. (There are 12 processor in each nodes in parallel programming's test machine. Because of only 10 multi-node, I just run 10 processors for matching multi-node experiment.)



(ms) / #No	1	2	3	4	5	6	7	8	9	10
Computing	2960	1610	940	860	560	590	480	420	430	400
Communicating	0	140	370	630	340	260	290	230	350	270
I/O	80	220	370	230	760	470	600	240	900	650

By the above bar chart and statistics data. Comparing the single & multiple version, we will know that the commuting time between Single node and Multiple node has not too much difference, which means the consuming communicating time between Nodes and Processes is very closed without too much time latency between them.

[Advanced Version -- Time Complexity Analysis]

Phase1: T-startup , MPI_IO loads n data (n = N / # Processors)

Phase2: do sort in each processor O(n* log*n)

Phase3: send data to neighboring processor and receive data from neighboring processor

Phase4: Merge data, take O (2 * n) ➔ It should be the bottleneck when n is larger in each processors.

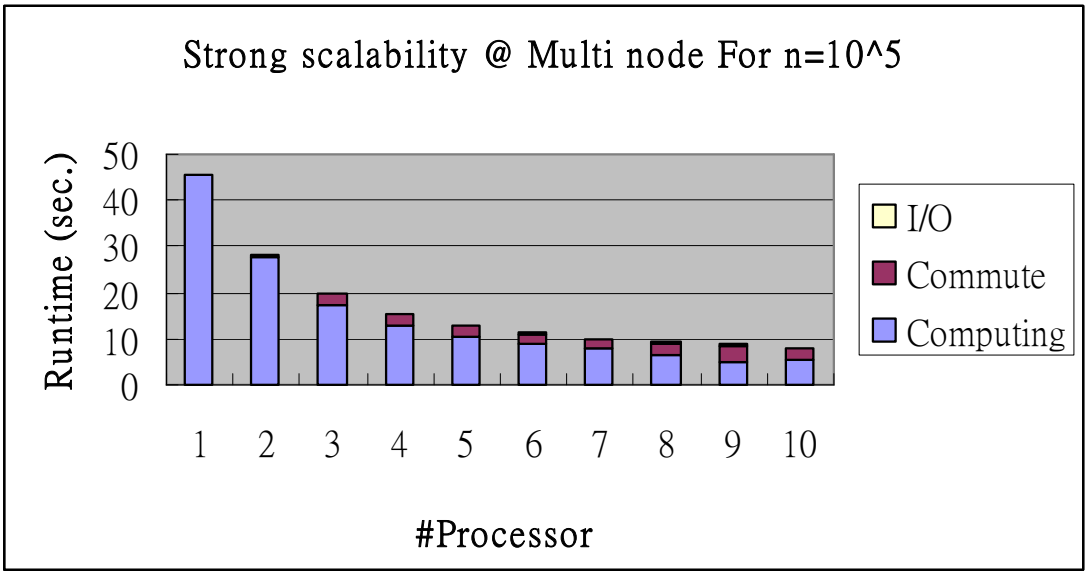
Phase5: get result, take O (n) , because receiving processor should go through the amount of n data to get sorted data.

Phase6: Other communications from Master involving get the exchange status for each processors by MPI_Reduce and notify each processors if sorting action should be done by MPI_Bcast.

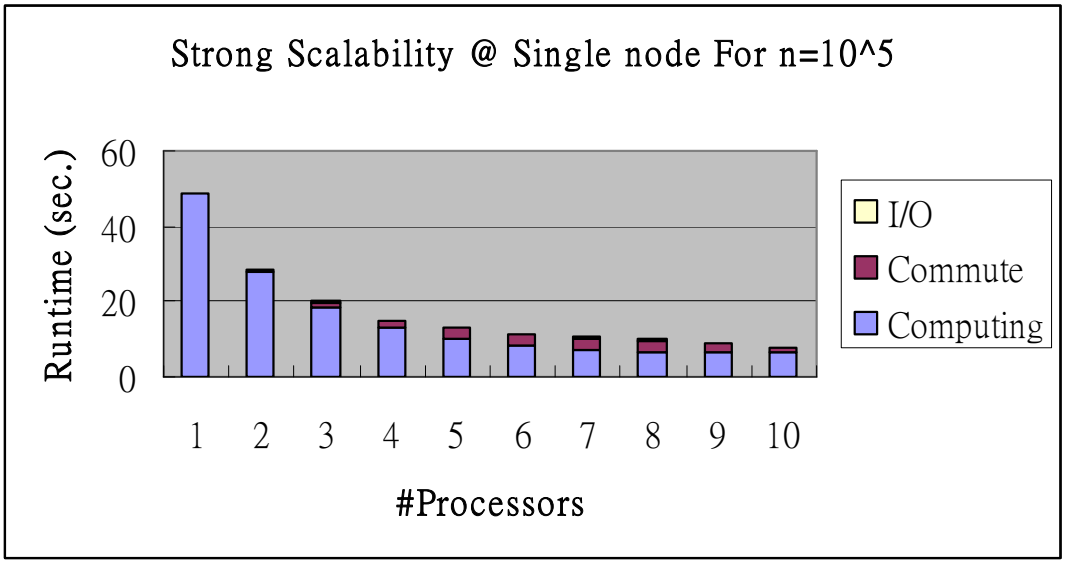
Phase3~Phase6 will be executed with many iterations, if Master do not send the “sorting done signal” to them.

Phase7: Save N/P data with MPI_IO

[Basic version] **N = 10^5**



(sec) / #No	1	2	3	4	5	6	7	8	9	10
Computing	45.73	27.81	17.1	13.1	10.43	8.79	7.7	6.43	5.04	5.63
Communicating	0	0.4	2.93	2.02	2.42	2.16	2.28	2.62	3.59	2.39
I/O	0	0.02	0.01	0.01	0.07	0.19	0.05	0.18	0.16	0.06



(sec) / #No	1	2	3	4	5	6	7	8	9	10
Computing	48.96	27.71	18.53	13	10.27	8.44	7.1	6.43	6.66	6.35
Communicating	0	0.59	1.34	2.08	2.58	2.56	3.18	2.83	2.06	1.61
I/O	0	0.01	0.08	0.04	0.11	0.25	0.5	0.68	0.15	0.03

[Basic Version -- Time Complexity Analysis]

Phase1: T-startup , MPI_IO loads n data (n = N / # Processors)

Phase2: send data to next neighboring processor and receive data from front neighboring processor If the exchange index is ture. → It should be the second or minor bottleneck because the processor should always communicating with other neighboring processors.

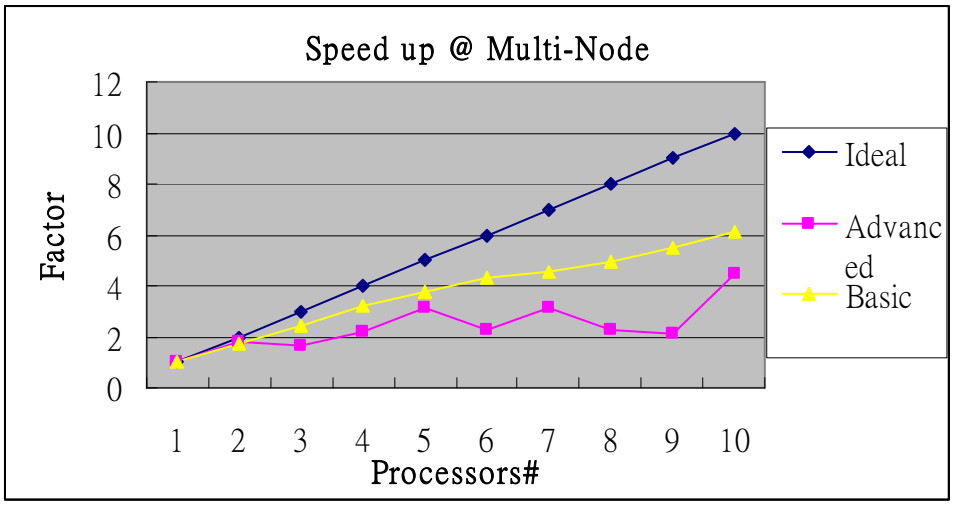
Phase3: Exchange the data which the processor holds on, except the first or last data which is marked “exchange with other processor”. →It should be the first or major bottleneck when n is larger in each processor. O(n)

Phase4: Other communications from Master involving get the exchange status for each processors by MPI_Reduce and notify each processors if sorting action should be done by MPI_Bcast.

Phase2~Phase4 will be executed with y many iterations, if Master do not send the “sorting done signal” to them.

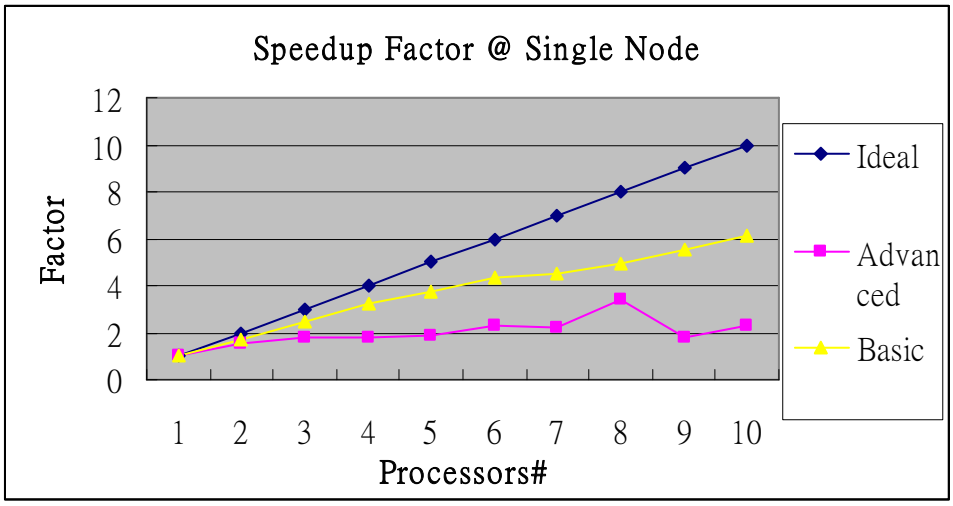
Phase5: Save N/P data with MPI_IO

2.3 Speedup factor



Ideal	1	2	3	4	5
Advanced	1	1.788235	1.643243	2.171429	3.134021
Basic	1	1.729424	2.454135	3.238095	3.777778

Ideal	6	7	8	9	10
Advanced	2.30303	3.102041	2.285714	2.140845	4.470588
Basic	4.352	4.541744	4.925553	5.519729	6.12766



Ideal	1	2	3	4	5
Advanced	1	1.543147	1.809524	1.767442	1.831325
Basic	1	1.729424	2.454135	3.238095	3.777778

Ideal	6	7	8	9	10
Advanced	2.30303	2.218978	3.41573	1.809524	2.30303
Basic	4.352	4.541744	4.925553	5.519729	6.12766

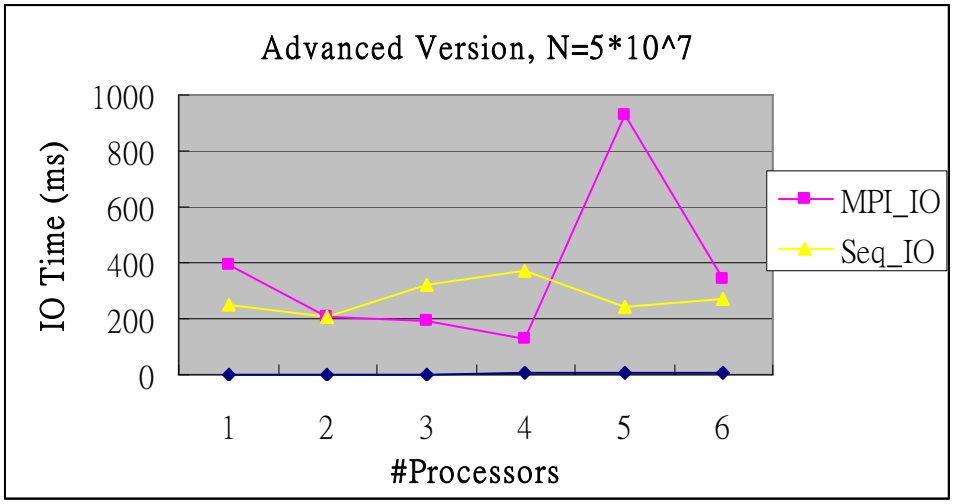
[Speedup Factor Analysis]

[Basic Version] In basic version, I get better speedup, when compared with advanced version. Because in basic version, If we only have the one process, the process should go through all data and swap them in each even or odd exchange action, the consuming time is bound by the amount of data. However, once I assign more processors, the amount of data will be cut down largely, which mean each process should go through the amount of n data only. (n = N / #Processors)

[Advanced Version] In advanced version, I get worse speedup, because when only one processor does even-odd sort, the processor will adopt qsort to sort data. The time complexity will be bound in O(N*log*N) without any communicating time, when processor is the only one. And, when processors is increased to more than one, each processor will do one time qsort in the beginning phase. The time is O(n* log*n). However, I send all data to its neighboring processors, so once neighboring processor receiving data, the processors should merge 2 times of n data containing itself data and sending data. So the time complexity will be add with O(2 * n). Based on the analysis, without considering the communicating time, my parallel advanced version should at least take O(n*log*n + 2*n). Finally, you will see the N*log*N dominates my programs, which is why I hardly get optimal speedup factor in advanced version.

2.4 Performance of different I/O ways

Running in single node by advanced version.



(ms) / #No	1	2	3	4	5	6
MPI_IO	390	210	190	130	930	340
Seq_IO	250	210	320	370	240	270

[Result] In the I/O experiment I found that there are not too much different when we reading data from MPI_IO and Sequential IO, although the communication time is increased in Sequential IO. However, the I/O action between two of them are not obviously different.

2.5 Compare two implementation

Basic Version. N = 10^5

(sec) / #No	1	2	3	4
Computing	48.96	27.71	18.53	13
Communicating	0	0.59	1.34	2.08
I/O	0	0.01	0.08	0.04

Advanced Version. N = 10^5

(sec) / #No	1	2	3	4
Computing	0.03	0.01	0.06	0.01
Communicating	0	0.03	0.12	0.02
I/O	0.01	0.06	0.11	0.02

[Analysis between basic and Advance version]

By the table, you will see the I/O is the least difference in basic and advanced version. And, the computing time decides how long the program will take. Besides, the communication time has the secondly obvious influenced on basic and advanced version. So we can analyze them individually.

[Computing] In advance version, the program use qsort to sorting data, once it do even or odd exchange, all the data in each process is order well. However, in basic version, the programs only can swap by its neighboring, like bubble sort. So the best case should take O(n). However, the average case will be O(n^2). So if we just focus on each processor action and not consider to exchange with other neighbor, the advanced version can finish it by O(nlogn). And the basic version only can finish sorting by O(n^2) in average case.

[Communicating] And then Now we add communicating action in it, because advanced version is processor-based, it can exchange all data with different processor, which mean once you send all the data to your neighboring processor, the receiving processor can sort them again, and receive sorted data back. And then all the data in the two processors should be sorted. However, in basic version, the data is element-based, which means it only can exchange data with its neighboring data. So there must have more

communicating actions happening than advanced version, because in basic version, it can exchange ONE data to its neighboring processor only.

By these two individual analysis, no matter computing time or communicating time the basic version take, there is larger efforts in basic verison than advanced version. Hence, it is straightforward to evaluate the time complexity is worse in basic version.

2.6 Others

[Advanced Version with worse performance when N is not large enough]

N= 10^5 , Running in single node

(ms) / #No	1	2	3	4
Computing	20	10	60	40
Commute	0	30	120	60
I/O	20	60	110	20
Total	40	100	290	120

N=10^6, Running in single node

(ms) / #No	1	2	3	4	5
Computing	240	140	200	110	100
Commute	0	20	120	190	210
I/O	10	40	50	130	60
Total	250	200	370	430	370

[Result] It gets worse performance in advanced version, when N is not large enough. When n is small, one processor can use qsort to sort data. Time complexity will be O(N * log * N) ,without any communicating time in each nodes or processors.

[Compare MPI_Send with MPI_Isend]

In Basic Version sending data in single nodes, when N=10^5 ,

MPI_Send (Blocking communicating mechanism)

(sec) / #No	2	3	4
Computing	27.81	17.1	13.1
Communicating	0.4	2.93	2.02
I/O	0.02	0.01	0.01
Total	28.23	20.04	15.13

MPI_Isend (Non-blocking communicating mechanism)

(sec) / #No	2	3	4
Computing	24.24	16.37	11.14
Communicating	15.79	8.45	6.05
I/O	0.05	0.03	0.06
Total	40.08	24.85	17.25

[Result] When switching to non-blocking communicating from MPI_Send to MPI_Isend, the communicating time has been largely cut down.

[Compare static exchange time with dynamic exchange time]

Advanced version sending data in single nodes, when N = 5 * 10^3

[Dynamic Exchange Time] : The program will judge if the sorting action should be done, once there are continuous 2 times non-swap actions happening.

(ms) / #No	2	3	4	5
Computing	0	11	0	5
Commute	7	10	30	70
I/O	78	230	130	110
Total	85	251	160	185
Exchange time	2	4	3	4

[Static Exchange Time] : The program will depends on the N's size to do the time of even and odd exchange. For example. If N=5000, the exchange action will be done with 5000 times.

(ms) / #No	2	3	4	5
Computing	150	750	1080	1700
Commute	80	1330	1990	1460
I/O	50	260	20	20
Total	280	2340	3090	3180
Exchange Time	5000	5000	5000	5000

[Result] If the program can detect that the sorting action is done earlier, the program will get better performance involving reducing the computing and communicating time.

3 Experience and conclusion

The homework lets me to know how to communicate with other processors. The following things is what I have learned in the homework.

1. When data cannot be distributed equally to other processors, we can use dummy data to represent the invalid data. When the programs meet them, it can ignore them. Then, we can use scatter function to send data to others processor without using scatterv func. Because using scatterv func, we should do extra effort to calculate different buffer size for each processors.

2. The MPI/IO have its itself mechanism to recording data. When data is large than one limit, the data content is totally different with the data that we using traditional IO like fwrite func. However, if you read by fread func, we will get wrong result. But if you read with MPI/IO, the result is correct.

3. The performance depends on how proportion part of our programs can be parallelized. However, even the high proportion of programs can be parallelized independently. We should take care of reducing the number of the communication of MPI. For example. If we need send data to all of processors, we should use scatter functions to cut off the number of MPI communication.