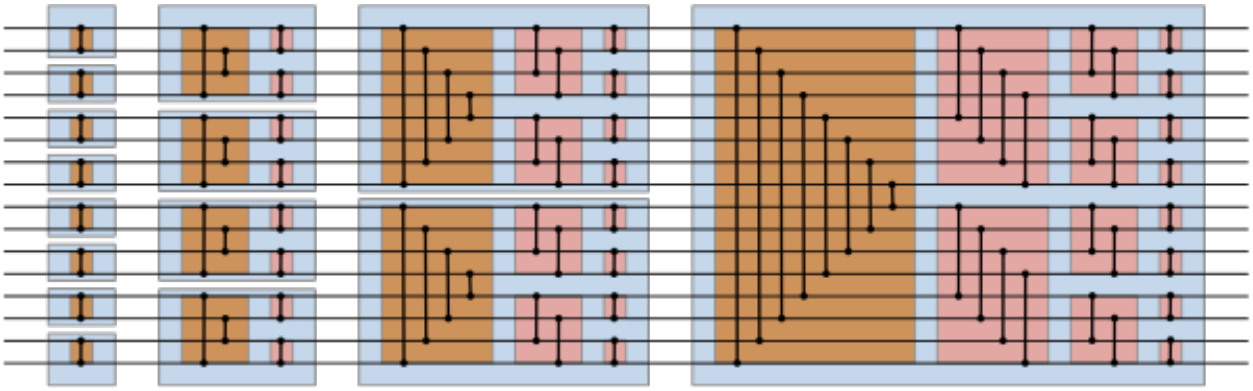
Bitonic Sort

Xiaowei Shen 50206168

1. Bitonic Sort

Bitonic sort is a sort algorithm that performs well on parallel implementation. For n integers (assume that n is a power of 2 and each processor holds 1 integer), the expect time cost on parallel implementation is

$$\Theta(\log^2 n)$$



2. Implementation on large amount of integers

The total number of integers is much greater than the total number of processors. Assume that there are N integers and we have n processors, each processor will hold N/n integers.

the first step is to distribute these data to each processor equally. In this implementation, each processor generates N/n random integers instead of generating N integers in one processor and then distributing them.

The next step is to sort the integers on each processor. In this implementation, each processor uses quick sort to sort the data that belongs to it. And the time cost of this part is

$$\Theta\left(\frac{N}{n}\log\frac{N}{n}\right)$$

Then it is the bitonic sort part. In this part 2 processors merge their local sorted integers together. Then put the smaller half part of integers to one processor and put the larger half part of integers to another processor. This part of running time is

$$\Theta\left(\frac{N}{n}\log^2 n\right)$$

So the total running time is

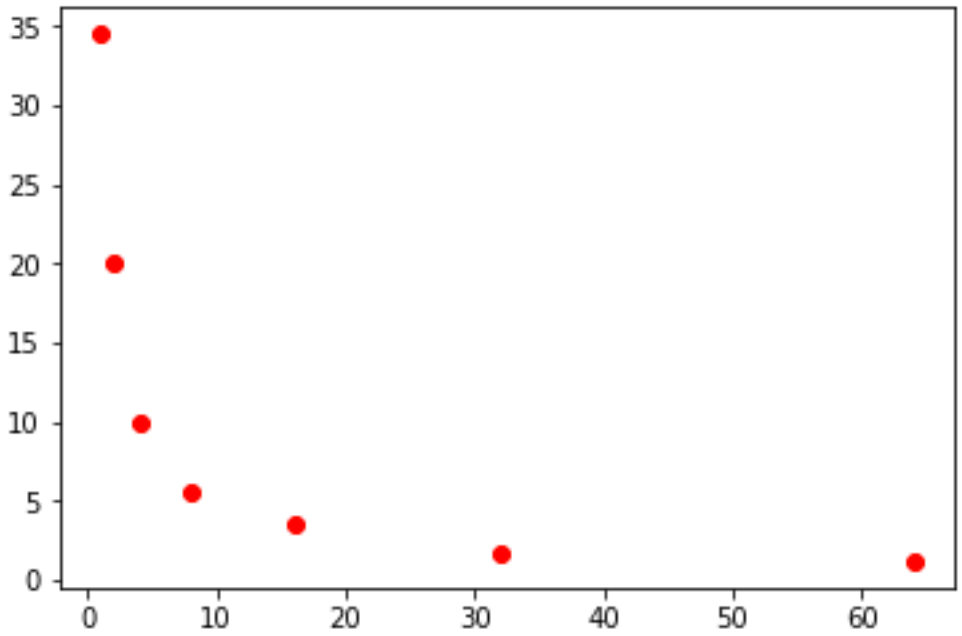
$$\Theta\left(\frac{N}{n}\log\frac{N}{n}+\frac{N}{n}\log^2 n\right)$$

3. Results

a. Change number of processor with fixed total number of integers

In this part the total number of integers is set to 128,000,000. And the number of processor is set from 1 to 64. Each result is the average time of running the algorithm 10 times.

Number of PE	1	2	4	8	16	32	64
Running time(sec)	34.533	20.034	10.01	5.525	3.57	1.736	1.203



From the graph we can find that the running time decreases obviously with the increasing of the number of processors. Consider the running time is

$$\Theta\left(\frac{N}{n}\log\frac{N}{n}+\frac{N}{n}\log^2 n\right)$$

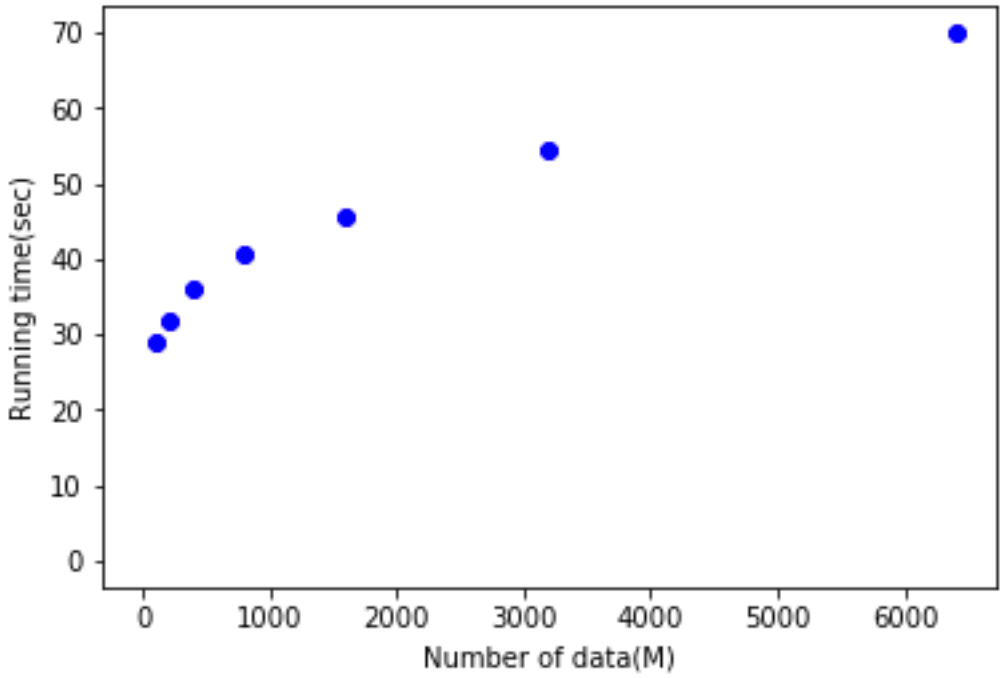
Here the major factor to influence the running time is the number of processor n.

Use large number of processors will save lots of time and it shows the benefits of parallel computing.

b. Change number of processor with fixed number of integers per processor

Each processor holds 100,000,000 integers. Each result is the average time of running the algorithm 10 times.

Number of processors	1	2	4	8	16	32	64
Number of total data	100M	200M	400M	800M	1600M	3200M	6400M
Running time(sec)	28.893	31.982	36.123	40.759	45.66	54.305	69.924



The time complexity of parallel bitonic sort is

$$\Theta\left(\frac{N}{n}\log\frac{N}{n}+\frac{N}{n}\log^2 n\right)$$

When $\frac{N}{n}$ is fixed to 100,000,000, the changing part is

$$\Theta\left(\frac{N}{n}\log^2 n\right)$$

From the graph we can say that the time complexity is O(n). It is much better than the performance of sequential algorithm, which time complexity is $\Theta(n\log n)$.

4. C Code

bitonic.c

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

int ierr, procid, numprocs;
int *nums, *receive_num, *send_num, *tmp;

const int TOTAL_NUM = 1;
const int MAX_NUM = 100;

int comp(const void *elem1, const void* elem2) {
  int f = *((int*)elem1);
  int s = *((int*)elem2);
  if (f > s) return 1;
  if (f < s) return -1;
  return 0;
}

bool _check_numprocs() {
  int count = 0, n = numprocs;
  while (n) {
    if (n & 1) ++count;
    n >>= 1;
  }
  return count == 1;
}

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

void _bitonic(int n, int m) {

  if (m < 2) return;
  int pos = procid % m, group = procid / m, half = m / 2;
        int pairid;
  if (n > m)  pairid = (procid < group * m + half) ? procid + half : procid -
half;
  else pairid = (group+1)*m - 1 - pos;

  MPI_Status status;
  int i, j, k;

  if (procid < group * m + half) {
    //received
    ierr = MPI_Recv(receive_num, TOTAL_NUM, MPI_INT, pairid, 0,
MPI_COMM_WORLD, &status);
```

```
    j = 0; //index of nums
    k = 0; //index of receive_num
    for (i = 0; i < TOTAL_NUM; ++i) {
      if (receive_num[k] < nums[j]) {
        tmp[i] = receive_num[k];
        ++k;
      } else {
        tmp[i] = nums[j];
        ++j;
      }
    }

    for (i = 0; i < TOTAL_NUM; ++i) {
      if (j == TOTAL_NUM || (k < TOTAL_NUM && receive_num[k] < nums[j])) {
        send_num[i] = receive_num[k];
        ++k;
      } else {
        send_num[i] = nums[j];
        ++j;
      }
    }
    ierr = MPI_Send(send_num, TOTAL_NUM, MPI_INT, pairid, 0, MPI_COMM_WORLD);

    for (i = 0; i < TOTAL_NUM; ++i) {
      nums[i] = tmp[i];
    }
  } else {
    //send
    ierr = MPI_Send(nums, TOTAL_NUM, MPI_INT, pairid, 0, MPI_COMM_WORLD);
    ierr = MPI_Recv(receive_num, TOTAL_NUM, MPI_INT, pairid, 0,
MPI_COMM_WORLD, &status);
    for (i = 0; i < TOTAL_NUM; ++i) {
      nums[i] = receive_num[i];
    }

  }

  _bitonic(n, m/2);
}

int main(int argc, char* argv[]) {

  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  if (!_check_numprocs()) {
    if (procid == 0) printf("Num of PEs should be power of 2.\n");
    return -1;
  }

  srand(time(NULL) + procid);

  nums = (int*) malloc(sizeof(int)*TOTAL_NUM);
  int i;
  for (i = 0; i < TOTAL_NUM; ++i) nums[i] = rand() % MAX_NUM;
```

```
  qsort(nums, TOTAL_NUM, sizeof(*nums), comp);
  //sort local data

  receive_num = (int*) malloc(sizeof(int)*TOTAL_NUM);
  send_num = (int*) malloc(sizeof(int)*TOTAL_NUM);
  tmp = (int*) malloc(sizeof(int)*TOTAL_NUM);

  for (i = 2; i <= numprocs; i <<= 1) {
     _bitonic(i, i);
  }

  free(nums);
  free(receive_num);
  free(send_num);
  free(tmp);

  return 0;
}
```