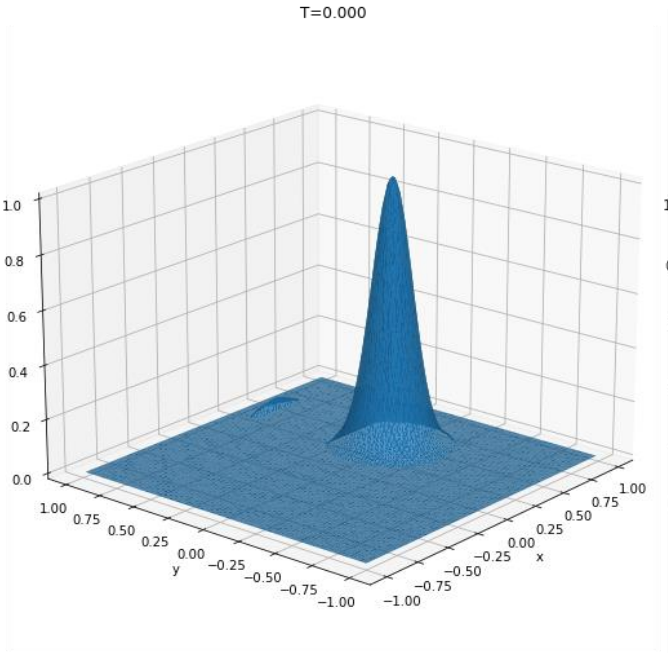
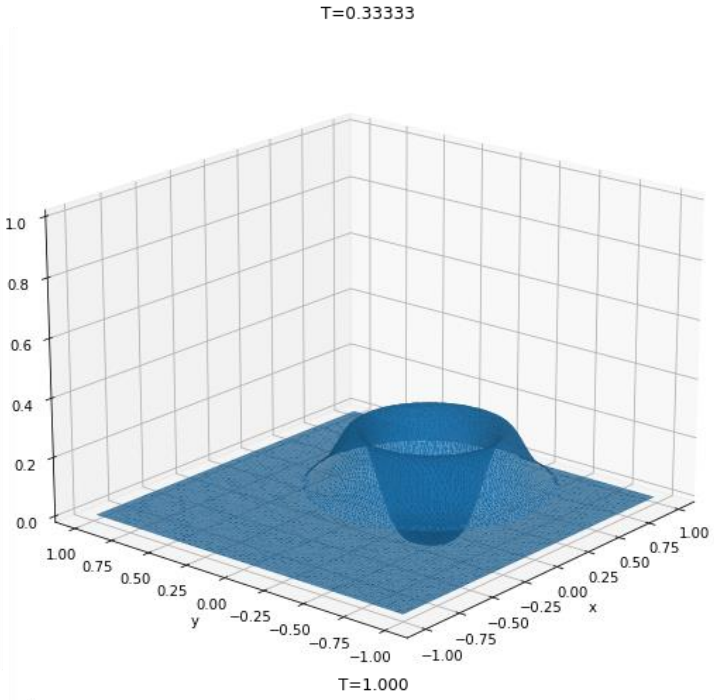
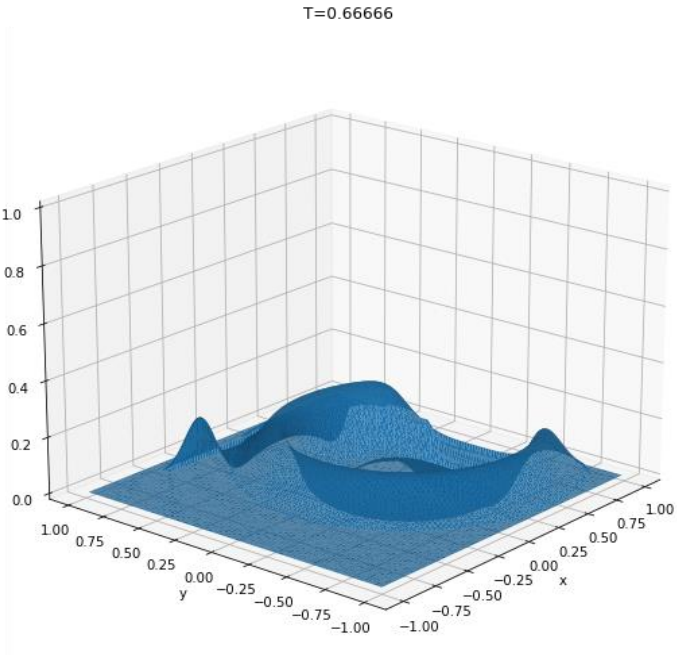
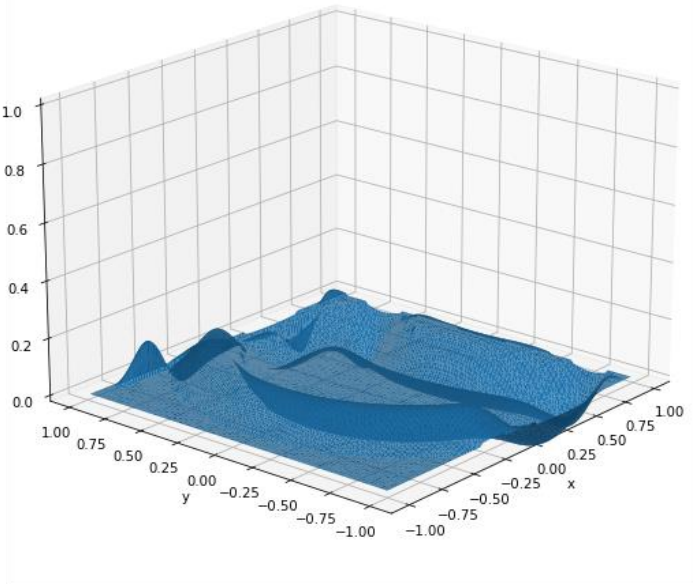
1.Result report:

In this problem, it is a 2-D simulation problem, results for 3 cases are similar, I just show the result from cass2, Since it is hard to output all the numbers (2306*2306）, I just output the average value of larger region, which leads some bias.









2.Code report:

1.Storage and Calculation of the data：

In this problem, it is a 2-D simulation problem, similar to the 1-D cases, I use three 2-D lists for store and calculation of the result in both three cases.

For case 1: (Code name: w5ass1)

```
1.  double inital2[2306][sep + 2], inital[2306][sep + 2], calcu[2306][sep + 2];
```

For case 2: (Code name: w5ass2)

```
1.  double inital2[sep + 2][2306], inital[sep + 2][2306], calcu[sep + 2][2306];
```

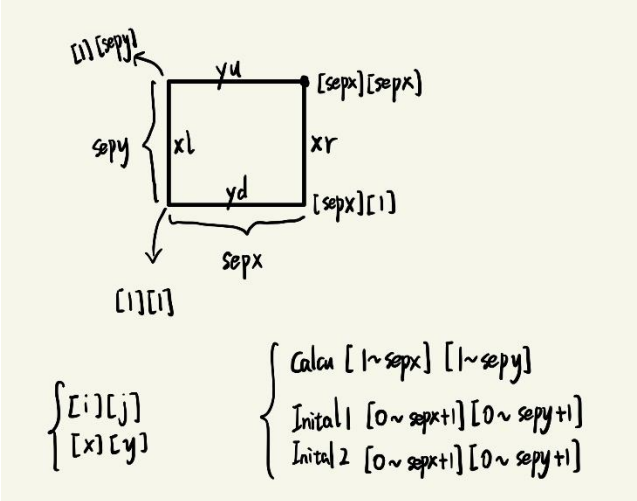
For case 3: (Code name: w5ass3)

```
1.  double inital2[236][236], inital[236][236], calcu[236][236];
```

2. Communication between cores：

In each time step, each core calculate its own grids and it needs to communicate for the next step, the important point is to specific which line each core needs to send and receive.

I demonstrate the case 3, case 1 and 2 are similar and easier.



In case3, I have set the core number as 100 in total, each core will calculate 230 or 234 lines, there are four lists that should be communicated, for the calcu list, it just need to consider [1~sepx][1~sepy] but inital1 and inital2 needs a bit more, that's depends on the simulation scheme we have chosen. So, four lists and named xl(x left), xr,yu,yd. And the key problem is on which cores it needs to communicate. To

solve this problem, one should know which lines this core is calculating, so it is important to know what the starting line, by simple caluculation:

```
1.  //xstart  ystart;
2.  xstart = (rank % 10) * 230;
3.  ystart = (rank / 10) * 230;
```

One we know this, it is easy to communicate between different cores and determined which line it is:

Sending and receiving are quiet similar, since the communication is symmetric.

Sending:

```
1.  //send yd
2.  if ((rank >= 10)) {
3.   MPI_Send(&(yd[0]), 236, MPI_DOUBLE, (rank - 10), k, MPI_COMM_WORLD);
4.  }
5.  //send yu
6.  if ((rank < 90)) {
7.   MPI_Send(&(yu[0]), 236, MPI_DOUBLE, (rank + 10), k, MPI_COMM_WORLD);
8.  }
9.  //send xl
10. if ((rank % 10 > 0)) {
11.  MPI_Send(&(xl[0]), 236, MPI_DOUBLE, (rank - 1), k, MPI_COMM_WORLD);
12. }
13. //send xr
14. if ((rank % 10 < 9)) {
15.  MPI_Send(&(xr[0]), 236, MPI_DOUBLE, (rank + 1), k, MPI_COMM_WORLD);
16. }
```

Receiving:

```
1.  if ((rank >= 10)) {
2.  MPI_Recv(&(yd[0]), 236, MPI_DOUBLE, (rank - 10), k, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3.  }
4.  //rec yu
5.  if ((rank < 90)) {
6.  MPI_Recv(&(yu[0]), 236, MPI_DOUBLE, (rank + 10), k, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7.  }
8.  //rec xl
9.  if ((rank % 10 > 0)) {
10. MPI_Recv(&(xl[0]), 236, MPI_DOUBLE, (rank - 1), k, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
11. }
12. //rec xr
13. if ((rank % 10 < 9)) {
14.  MPI_Recv(&(xr[0]), 236, MPI_DOUBLE, (rank + 1), k, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
15. }
```

3. More details can be found in the codes:

PS: I found it is so hard to output all the data, even using some process like pin pang to ask each core to output the result one by one by itself, so I choose to output the average.

```
pdd = 0;
MPI_Send(&(pdd), 1, MPI_INT, 1, -1, MPI_COMM_WORLD);  // for core 0.
pdd = 1;
if (pdd == 0) {...
..
  pdd = 0;
  if (rank <= 98) {
      MPI_Send(&(pdd), 1, MPI_INT, rank + 1, -1, MPI_COMM_WORLD);
  }
 pdd = 1;
}
```

3.Running time comparison:

1. Comparison under 108/100 cores:

As set up in 1.1, case 1 and case 2 both 108 cores (3 nodes with 36 for each one), case 3 use100 cores, the running time is as follow:

	Begin	End	Time cost (seconds)
Case1	942948	942994	46
Case2	943323	943333	10
Case3	943487	943497	10

We can see the storage method of the data is highly related to the speed, in case one, I use [2306][sep+1] which has a lot rows than columns, since in C++, lists are stored row after row, in the code I use a lot calculation more base on different rows, the time cost for it to search is high.

```
      //make head tail:
For (I = 1; I < m; i++) {
 head[i] = calcu[i][1];
 tail[i] = calcu[i][sepp];
}
```

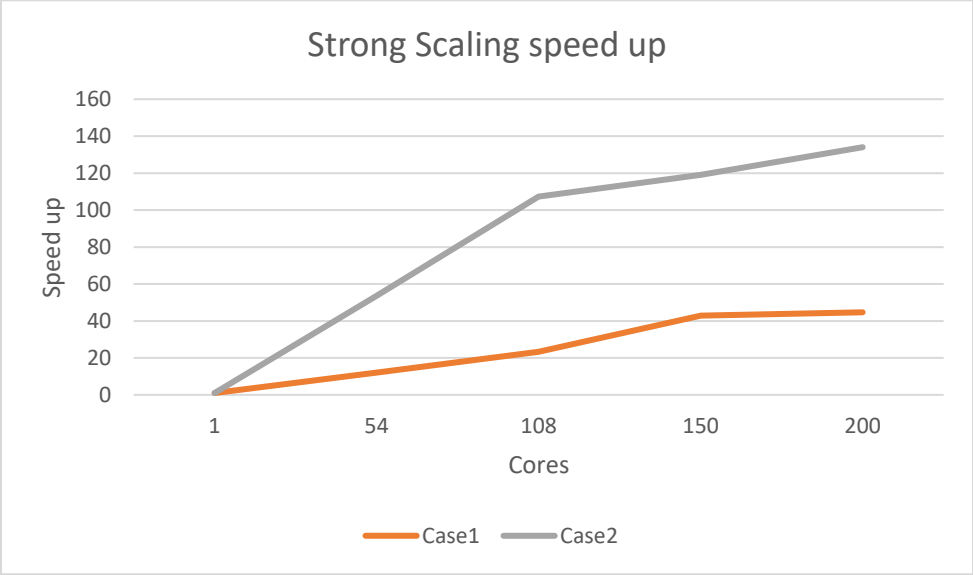
In case 1, things like these are more time-costly than other cases.

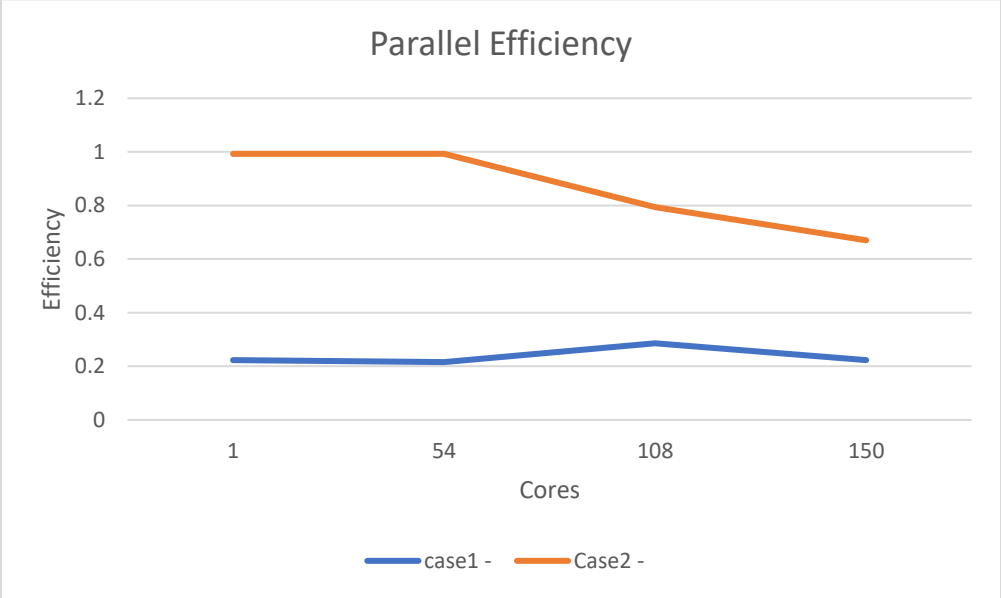
We can see the storage method of the data is highly related to the speed, in case one, we use [2306][sep+1] which has a lot rows than columns, since in C++, lists are **Row-Majored**, in the code I use a lot calculation more base on different rows, the time cost for it to search is much more than the other two cases.

2. Speed up/ efficiency | Strong/Weak scaling：

Core 1 due to the limitation of storage space, I just use similar calculation steps and transform steps, there will be more time consuming when using a big list to store all the data. (Code name: w5asssup)
Estimate running time:

Single core time = 953288-952216=1072 (s)





Cores	case1 Time	case2 Time	case1 Speed up	case2 Speed up	case1 Efficiency	case2 Efficiency
1	1072	1072	1	1	-	-
54	89	20	12.04494	53.6	0.223055	0.992593
108	46	10	23.30435	107.2	0.215781	0.992593
150	25	9	42.88	119.1111	0.285867	0.794074
200	24	8	44.66667	134	0.223333	0.67

As we can see, case 1 has bad speed up and bad parallel efficiency compare to case 2. And case 2 seems to have a good linear-like speed up trend. Which tells us that in C++, the Row-Majored list type is a crucial point to consider when dealing with parallel computing.

Thank you.

Xingyuan Chen.