Assignment4
Student Name: Yi-Ting, Hsieh(a1691807)

Objective:

To gain understanding about how to use OpenCL as well as utilizing GPU hardware to achieve speedup for parallel bubblesort and analyzing the performance compared to sequential bubblesort.

Implementation of parallel version of bubble sort:

1. **Initializes compute device:**

   The first step of host application is to obtain information for each device that will execute the kernel. The parallel bubblesort application accesses the first GPU device associated with the first platform. This is shown in the following code from hostCode.cpp:

   ```
   er = cl::Platform::get(&all_platforms);
   cl::Platform default_platform=all_platform[0];
   err = default_platform.getDivces(CL_DEVICE_TYPE_ALL,&all_devices);
   cl::Device default_device = all_devices[0];
   ```

2. **Defines problem domain:**

   Next, the application creates a context containing only one device – the *default_device* structure created earlier.

   ```
   cl::Context context(default_device);
   ```

   After creating the context, the application creates a program from the source code in the hostCode.cpp file. Specifically, the code reads the kernel by passing the kernel file name to *loadKernel()*, and then calls *Program* to bind it with *sources*.

   ```
   cl::Program::Sources sources;
   std::string kernel_code = loadKernel("kernel_code.cl");
   sources.push_back({kernel_code.c_str(), kernel_code.length()});
   cl::Program program(context, sources);
   ```

   Once the program is created, its source code must be compiled for the devices in the context. The function that accomplishes this is *buld()* , and the following code shows how it is used in the parallel bubblesort application:

   ```
   err = program.build({default_device});
   ```

   After a *program* has been compiled, kernels can be created from its functions. The following code creates *kernel_even_step* from the function called *even_step()* as well as *kernel_odd_step* from the function called *odd_step()*:

```
cl::Kernel kernel_even_step = cl::Kernel(program, "even_step");
cl::Kernel kernel_odd_step = cl::Kernel(program, "odd_step");
```

Before the application can dispatch these kernels, it needs to create a command queue to a target device. We configure this command queue to support profiling, which allows us to measure the time taken for a kernel's execution:

```
cl::CommandQueue queue(context, default_device, CL_QUEUE_PROFILING_ENABLE,
&err);
```

At this point, the application has created all the data structures(device, kernel, program, command queue and context) needed by an OpenCL host application. However, it also needs to know how many *work_group* we have:

```
size_t workgroup_size;
err = clGetDeviceInfo(default_device(), CL_DEVICE_MAX_WORK_GROUP_SIZE,
sizeof(workgroup_size), &workgroup_size, NULL);

int maxWorkGroupSize = workgroup_size;
int numWorkGroups = (N-1) / maxWorkGroupSize + 1;
int globalSizePadded = numWorkGroups * maxWorkGroupSize;
```

3. **Allocates memory in host and on device:**

In this parallel program, the kernel is executed by the input size(n) of work-items divided into n/2 work-groups of 2 work-items each. To perform odd-even bubblesort, we have to distribute work-items to device. On the other hand, our kernel will have each work-group that compare the two work-items and decide whether it should swap the order. These partial order will be put together to form an array for the entire group. In the end, the kernel will return the sorted array. In our hostCode.cpp, the buffer_A stores the entire array, and buffer_B stores the size of the unsorted array.

```
cl::Buffer buffer_A(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(int)*N, array, &err);
cl::Buffer buffer_B(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(int), &N, &err);
```

4. **Copies data from host to device memory:**

Also, we have to set values of kernel's arguments, the first argument for both *kernel_even_step* and *kernel_odd_step* are the unsorted array, and the second argument is the length of the array:

```
kernel_even_step.setArg(0,buffer_A);
kernel_even_step.setArg(1,buffer_B);
kernel_odd_step.setArg(0,buffer_A);
kernel_odd_step.setArg(1,buffer_B);
```

5. **Launches execution "kernel" on the device:**
   Of the OpenCL functions that run on the host, enqueueNDRangeKernel not only deployes kernels to devices, it also identifies how many work-items should be generated to execute the kernel and the number of work-items in each work-group. Since we are using odd-even approach, we have to decide when to perform `even_step` sorting, and when to perform `odd_step` sorting.

```
   if (k % 2 == 0) {
      queue.enqueueNDRangeKernel(kernel_even_step, cl::NullRange,
      globalSizePadded, cl::NullRange, NULL, &event);
   } else {
      queue.enqueueNDRangeKernel(kernel_odd_step, cl::NullRange,
      globalSizePadded, cl::NullRange, NULL, &event);
  }
```

6. **Repeats 5 as needed:**

   Implementing the parallel bubblesort using the odd-even transposition method that implies the existence of n phases, each requiring n/2 compare and exchanges.

```
for (int k=0; k<N; k++) {
   if (k % 2 == 0) {
      queue.enqueueNDRangeKernel(kernel_even_step, cl::NullRange,
      globalSizePadded, cl::NullRange, NULL, &event);
   } else {
      queue.enqueueNDRangeKernel(kernel_odd_step, cl::NullRange,
      globalSizePadded, cl::NullRange, NULL, &event);
  }
 }
```

7. **Copies data from device memory to host:**

   At the end, the kernel will return the sorted array in buffer_A. We copy the sorted array into our res array:

```
int *res = new int[N];
queue.enqueueReadBuffer(buffer_A,CL_TRUE,0,sizeof(int)*N,res);
```

8. **De-allocates all memory and terminates:**

   Once the application perform the bubblesort, it has to de-allocates all memory before it terminates.

```
delete array;
delete res;
```

Analyze:

Machine: Phoenix SuperComputer

Measurements:

First we run our sequential program and parallel program both started the input size(n) at 10000, the reason we don’t start from small size is that the sequential program can esaily solve the array instantly, and the speedup for parallel program will not be that noticable. Then, we increase the size by 10000 and collect the data. Then we keep iterating the same process agiain and again until n is equal to 100,000(Note: we collect 10 times data in each point and calculate the average time required for sorting an array in that size). Once if we gather all the data, we can compare the performance difference between the sequential bubblesort program with the parallel version as well as calculating the speedup.

BubbleSort(in Sequential):

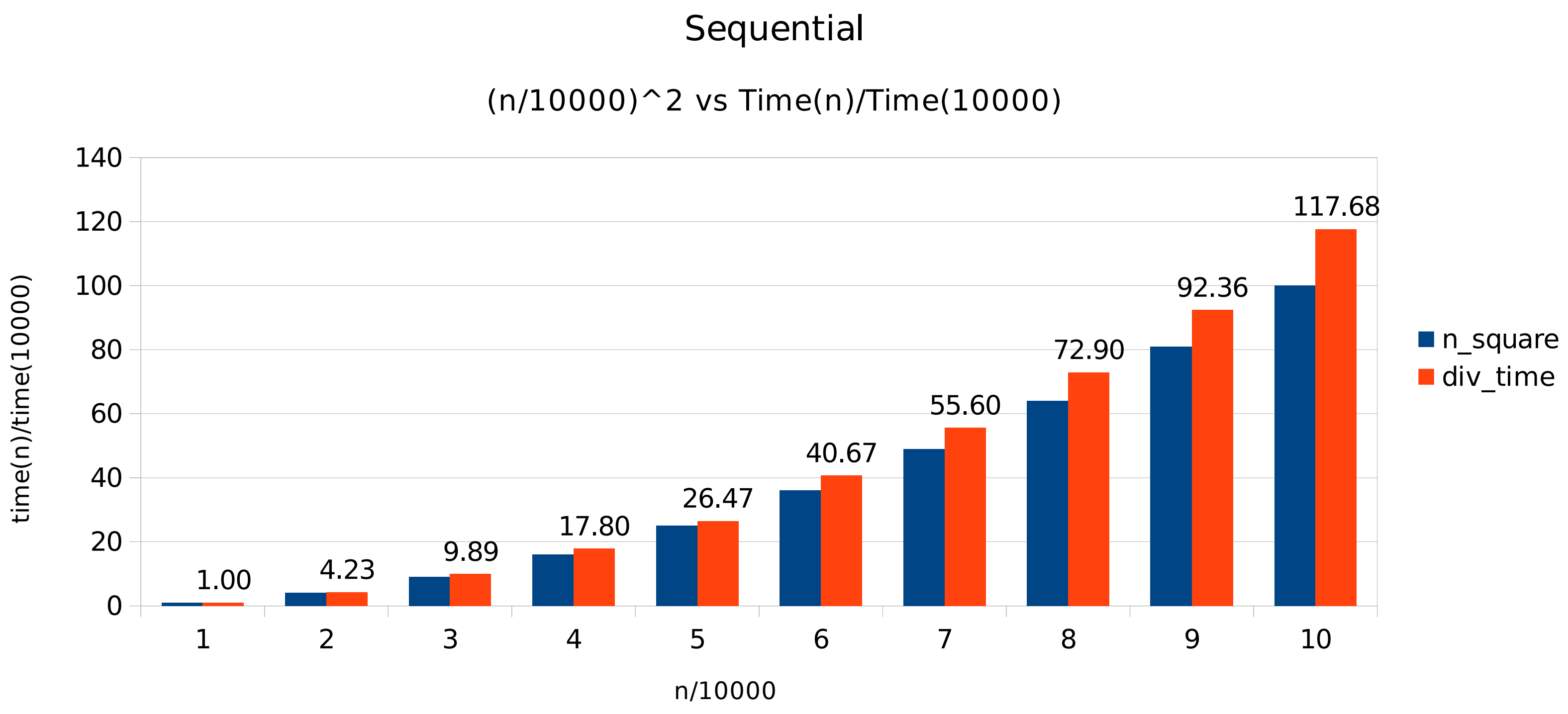
1.) Bubble sort is a simple sorting algorithm, it is also a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Performance results:

n	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	0.35	1.49	3.45	6.23	9.17	14.29	19.47	25.56	32.23	41.22
2	0.35	1.47	3.44	6.23	9.2	14.23	19.42	25.59	32.35	40.73
3	0.35	1.51	3.47	6.24	9.15	14.29	19.4	25.52	32.29	41.19
4	0.35	1.47	3.54	6.22	9.18	14.22	19.43	25.49	32.36	41.17
5	0.35	1.47	3.5	6.21	9.16	14.22	19.51	25.52	32.37	41.22
6	0.35	1.47	3.47	6.22	9.22	14.24	19.48	25.42	32.39	41.22
7	0.35	1.49	3.44	6.23	9.17	14.2	19.48	25.48	32.28	41.34
8	0.35	1.49	3.42	6.26	9.16	14.25	19.48	25.5	32.36	41.26
9	0.35	1.47	3.44	6.22	9.32	14.18	19.44	25.48	32.26	41.24
10	0.35	1.48	3.44	6.23	9.92	14.23	19.49	25.58	32.36	41.28
Avg(sec)	0.35	1.481	3.461	6.229	9.265	14.235	19.46	25.514	32.325	41.187

2.) *In sequential version, bubble sort has $O(n^2)$ complexity.* First, it will go through the list, swap might happen if the adjacent elements are not in order and push the larger number to the back. Once it reaches to the end of the array, we will get the maximum number in that array. Then, we keep iterating the same procedure again and finding the next largest number until we sort out all the array or there is no swap required, the program will learn that the array is completed sorted.

n/10000	1	2	3	4	5	6	7	8	9	10
(n/10000)^2	1	4	9	16	25	36	49	64	81	100
time(n)/time(10000)	1.00	4.23	9.89	17.80	26.47	40.67	55.60	72.90	92.36	117.68



3.) In order to prove the complexity for a sequential bubblesort program is $O(n^2)$, we divide the requried time for the program to sort an random array with the input size(n) by the required time of sorting an array with n equal to 10000. As you can see from the above table and chart, when the $\frac{n}{10000}$ is less or equal to 5, the $\frac{time(n)}{time(10000)}$ doesn’t have much difference with $(\frac{n}{10000})^2$. Even when the n becomes larger, the required time is still less than $2x(\frac{n}{10000})^2$. It also showws that the upper bound f the complexity is in $O(n^2)$.

BubbleSort(in Parallel):

1.) It is not easy to transform the sequential bubblesort to parallel version directly because it compares pairs of consecutive elements. However, we could implement the parallel bubblesort using the odd-even transposition method that implies the existence of n phases, each requiring n/2 compare and exchanges. In the first phase, called even phase, the elements having even indexes are compared with the neighbors from the right and the values are swapped when necessary. In the odd phase, the elements with odd indexes are compared with the elements from the right and the exchanges are performed only if necessary.

PSEUDO CODE for parallel bubblesort:

```
For k = 0 to n-2
      If k is even then
            for i=0 to (n-2)-1 do in parallel
                  if A[2i]>A[2i+1] then
                        exchange A[2i] with A[2i+1]
      else
            for i=0 to (n-2)-2 do in parallel
                  if A[2i+1]>A[2i+2] then
                        exchange A[2i+1] with A[2i+2]
```

2.) The array will be fully sorted after n phases, where n is the number of elements in the array. The inner loop iterations will be performed in parallel on the processors available in the system, which is the Nvidia GPU.

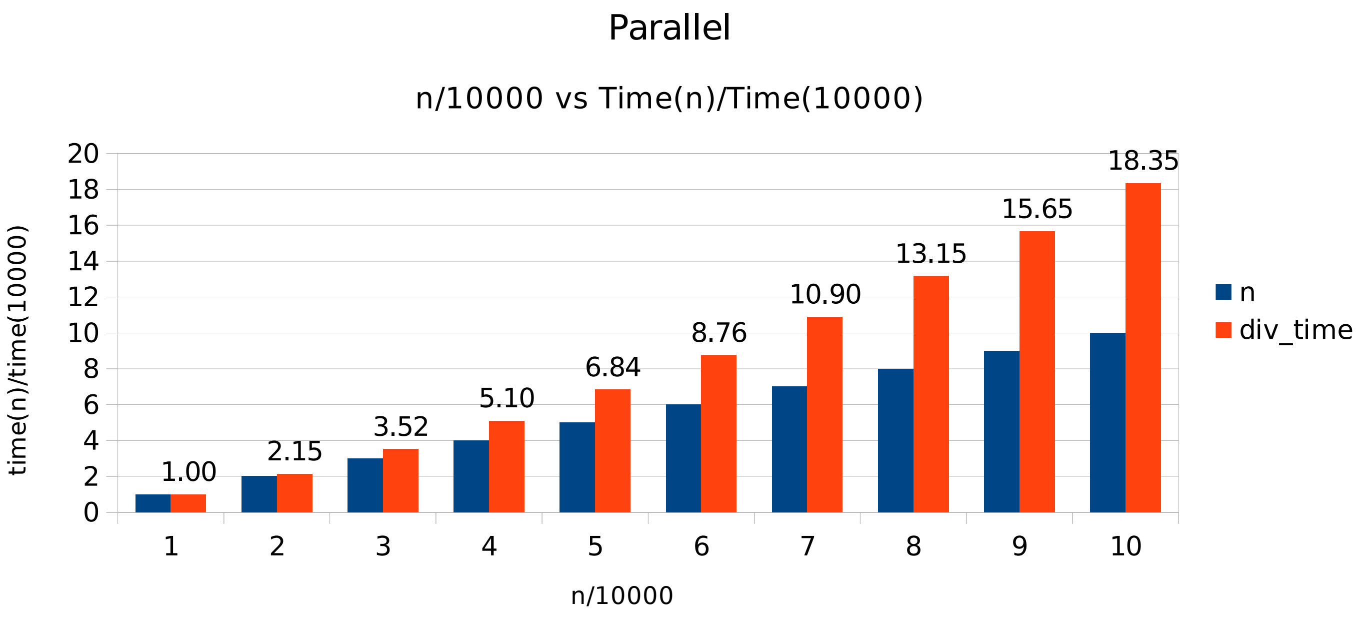
Performance results:

N	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	0.0989	0.2132	0.3497	0.4999	0.6750	0.8613	1.0872	1.2934	1.5412	1.8063
2	0.0987	0.2125	0.3477	0.5074	0.6742	0.8594	1.0889	1.3067	1.5403	1.8019
3	0.0975	0.2119	0.3445	0.5062	0.6739	0.8616	1.0675	1.2953	1.5427	1.8048
4	0.0984	0.2114	0.3505	0.5043	0.6739	0.8599	1.0645	1.2918	1.5414	1.8238
5	0.0986	0.2109	0.3449	0.5061	0.6745	0.8660	1.0720	1.3116	1.5385	1.8076
6	0.0984	0.2120	0.3490	0.5035	0.6718	0.8738	1.0677	1.2855	1.5583	1.8040
7	0.0986	0.2123	0.3492	0.5079	0.6725	0.8611	1.0817	1.2857	1.5412	1.8026
8	0.0985	0.2101	0.3489	0.4964	0.6747	0.8617	1.0763	1.2983	1.5403	1.8136
9	0.0989	0.2108	0.3417	0.4987	0.6780	0.8622	1.0711	1.2993	1.5427	1.8128
10	0.0995	0.2123	0.3482	0.5028	0.6743	0.8665	1.0707	1.3010	1.5414	1.8118
Avg(sec)	0.0986	0.2118	0.3474	0.5033	0.6743	0.8633	1.0748	1.2969	1.5428	1.8089

3.) For a common array, the parallel version of bubble sort will perform n iterations of the main loop and for each iteration a number of n-1 comparisions and (n-1)/2 exchanges will be performed in parallel.
If the number of processors is lower than n/2, every processor will execute (n/2)/p comparisions for each inner loop. In this case, the complexity level of the program could be $O(\frac{n^2}{2p})$.

If the parallel system has a number of processors p higher than n/2, the complexity level of the inner loop is O(1) because all the iterations are performed in parallel. The outer loop will be executed for maximum n times. *So the final complexity level of the algorithm is equal with O(n),* which is much faster compared to the sequential one with $O(n^2)$ complexity.

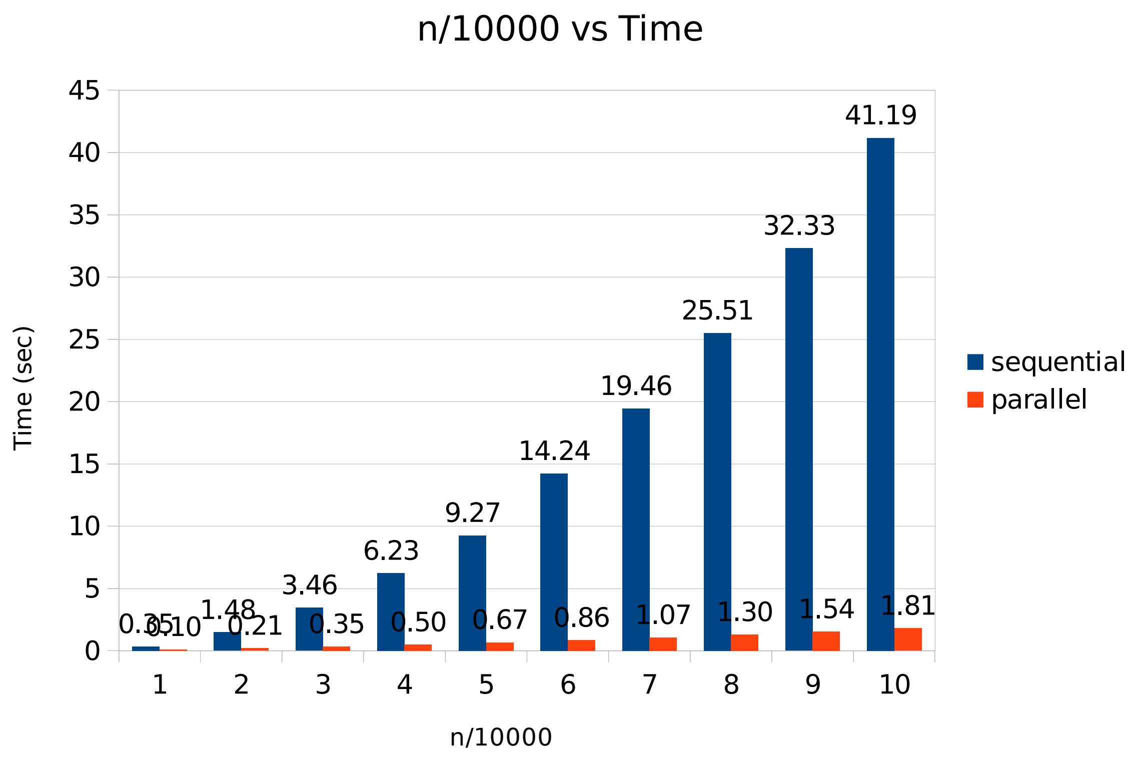
N/10000(base)	1	2	3	4	5	6	7	8	9	10
time(n)/time(10000)	1.00	2.15	3.52	5.10	6.84	8.76	10.90	13.15	15.65	18.35



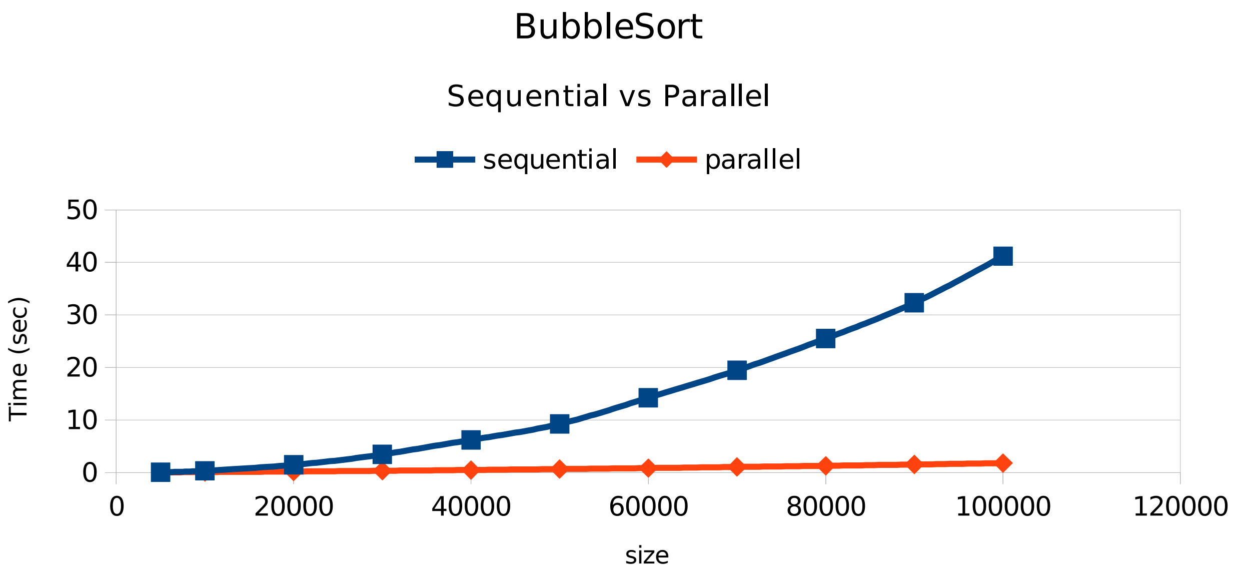
4.) As we mentioned above, the parallel version of bubblesort has the complexity of $O(n)$, we divide the requried time for the program to sort an random array with the input size(n) by the required time of sorting an array with n equal to 10000. From the above table and chart, $\frac{time(n)}{time(10000)}$ is less than $2x(\frac{n}{10000})$. It also shows that the upper bound is $c\,x\frac{time(n)}{time(10000)}$, where c is a constant value. As a result, the parallel version of bubblesort program has $O(n)$ complexity.

Result:

Let’s compare the performance between sequential version of bubblesort and parallel version of bubblesort together. We will get the following diagram: (where n is the size of input array)



At first, when the size of array is equal to 10000, the performances are quite similar for both sequtial program and parallel program. However, when the n increases and becomes larger, we can easily notice that it takes much longer for sequential bubble sort program to finish. On the other hand, the required time for parallel bubble sort does not have much difference, it can easily sort an array in 2 seconds even when the size is 10 times larger than 10000.



According to Amdahl’s law, the performance of speedup equation is:

$$S_p(n)=\frac{T_x(n)}{T_p(n)}$$

And we could use this to calculate how much the parallel version of bubblesort could speedup:

Size of input(n)	bsortseq(sequential)	bsortpar(parallel)	speedup
10000	0.3500	0.0986	3.55
20000	1.4810	0.2118	6.99
30000	3.4610	0.3474	9.96
40000	6.2290	0.5033	12.38
50000	9.2650	0.6743	13.74
60000	14.2350	0.8633	16.49
70000	19.4600	1.0748	18.11
80000	25.5140	1.2969	19.67
90000	32.3250	1.5428	20.95
100000	41.1870	1.8089	22.77

Obviously, as the input size becomes larger, the more speedup we could get. From the above table, we could expect that when the n becomes bigger, the more speedup we could achieve in parallel version of bubblesort.

Improvement:

There is another trick that could slightly improve the performance for our parallel bubblesort. We could use a shared global boolean flag in both odd_step and even_step to inidicate that the sorting operation is complete. By doing so, we may have a chance that not going through all the n iterations of our main loop, but the complexity remains the same.