KINGDOM OF SAUDI ARABIA
Ministry of Education
Taibah University
College of Computer Science and Engineering
(Girls Section)

جامعة طيبة
كلية علوم و هندسة الحاسبات
(قسم الطالبات)

CS424 – Introduction to Parallel Computing

Odd-Even Transposition Sort Variant of Bubble Sort Algorithm using OpenMP and Java Multithreading

by

Amal Aljabri	3550343
Rawan Qasem	3550277
Raneem Alsaedi	3654599
Maimounah Alhejaili	3654500

Supervised by

Dr.Nahla Abid

Semester II 2019/2020

Table of Contents:

Introduction

Sorting is a basic problem in computer science. It is also an issue with the limitation of algorithmic efficiency. As such, he's a successful candidate to be parallelized. In this report, we will sort a list of n integers values from a file in increasing order. We will do this by using the simplest sorting algorithm which is a variant of bubble sort known as odd-even transposition sort, it has considerably more possibilities for parallelism. In this report, parallelism is achieved for this algorithm by using #pragma parallel in OpenMP and using Java Multithreading.

1 Description of the problem and the sequential solution

The bubble sort algorithm works by comparing each value in the list with the value next to it, and swapping them if required. The bubble sort algorithm repeats this process until it makes a pass all the way through the list without swapping any values. The problem with the bubble sort algorithm appears when myList of values is reverse sorted. This problem has $O(n^2)$ complexity, so the bubble sort algorithm is slow and inefficient and also difficult to parallelize[1]. We will use a variant of bubble sort known as odd-even transposition sort. This algorithm uses one loop instead of using two loops (outer and inner loop as is bubble sort) and this loop will work in two alternate phases, the first phase works with even indices and the second phase works with odd indices. To make this algorithm clearer, we will illustrate the phases using a small example:

```
Start: 8, 9, 5,3
Even phase: Compare-swap (8,9) and (5,3) ---> getting the list 8, 9, 3, 5.
Odd phase: Compare-swap (9,3) ---> getting the list 8, 3, 9, 5.
Even phase: Compare-swap (8,3) and (9,5) ---> getting the list 3, 8, 5, 9.
Odd phase: Compare-swap (8,5) ---> getting the list 3, 5, 8, 9.
```

This example requires four phases to sort a four-element list. In general, this algorithm may require a number of phases equal size number of the list. The following figure 1.1 shows the flowchart of the serial solution for odd-even transposition sort. The following figure 1.2 shows code for a serial odd-even transposition sort function.

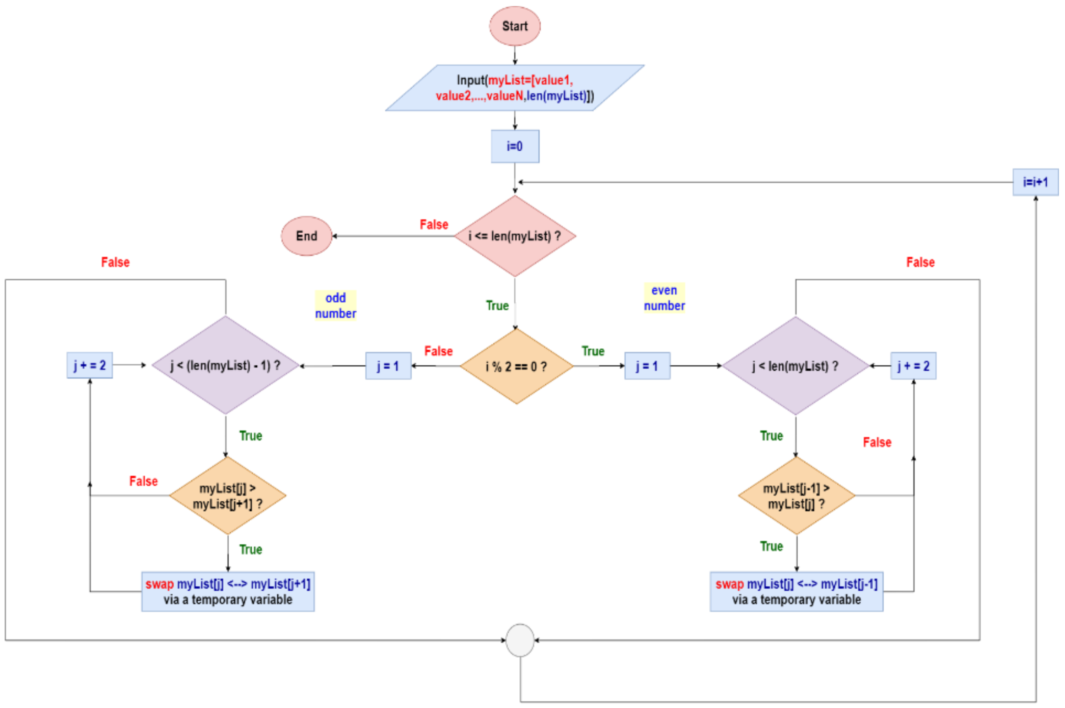
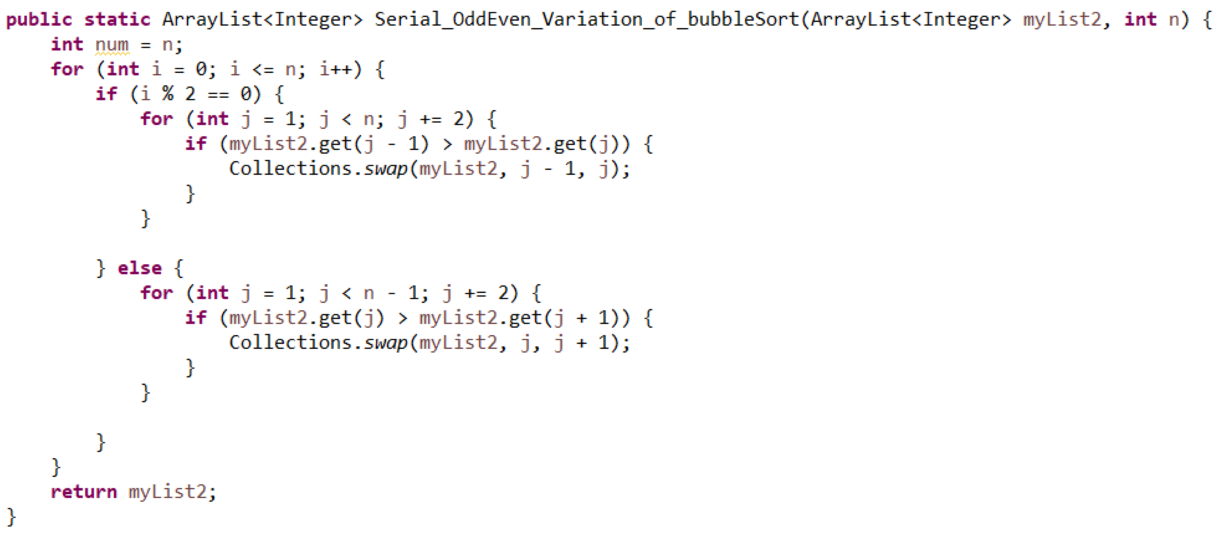

Start
Input(myList=[value1, value2,...,valueN,len(myList)])
i=0
i=i+1
End
False
i <= len(myList) ?
True
odd number
even number
False
False
j += 2
j < (len(myList) - 1) ?
j = 1
False
i % 2 == 0 ?
True
j = 1
j < len(myList) ?
j += 2
True
True
False
False
myList[j] > myList[j+1] ?
myList[j-1] > myList[j] ?
True
True
swap myList[j] <--> myList[j+1] via a temporary variable
swap myList[j] <--> myList[j-1] via a temporary variable

Figure 1.1 The flowchart of the serial solution



```java
public static ArrayList<Integer> Serial_OddEven_Variation_of_bubbleSort(ArrayList<Integer> myList2, int n) {
    int num = n;
    for (int i = 0; i <= n; i++) {
        if (i % 2 == 0) {
            for (int j = 1; j < n; j += 2) {
                if (myList2.get(j - 1) > myList2.get(j)) {
                    Collections.swap(myList2, j - 1, j);
                }
            }

        } else {
            for (int j = 1; j < n - 1; j += 2) {
                if (myList2.get(j) > myList2.get(j + 1)) {
                    Collections.swap(myList2, j, j + 1);
                }
            }

        }
    }
    return myList2;
}
```

Figure 1.2 The code of the serial solution

This algorithm has O(n) complexity and but not the optimal one[2]. We plan to solve the problem and improve a serial odd-even transposition sort function by parallelism. We will achieve parallelism with this algorithm using #pragma parallel in OpenMP and using Java Multithreading.

1.1 OpenMP

Using OpenMP we do fork team of THREAD_COUNT threads before the outer loop by parallel directive. In inner loops we do parallel the inner loops with the existing team of threads by for directive. The following figure 1.3 shows parallel odd-even transposition sort plan with an example.

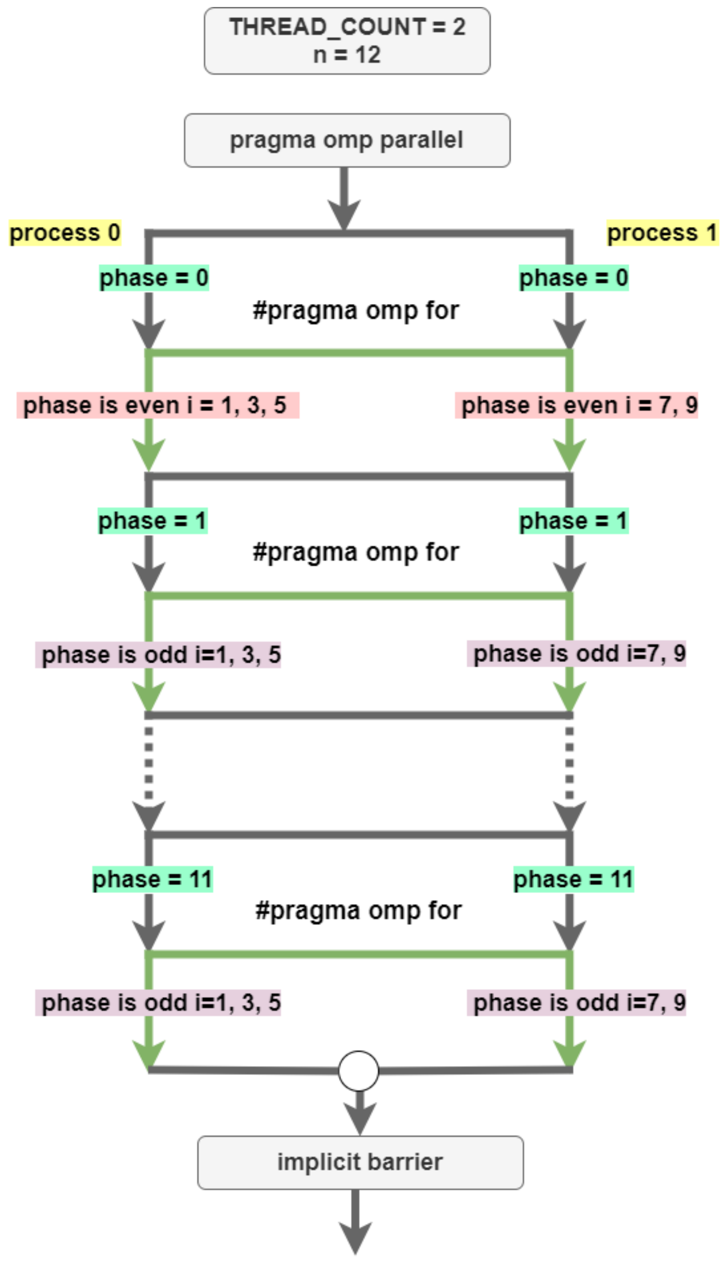


Figure 1.3 The parallel odd-even transposition sort plan

1.2 Java Multithreading

The following figure 1.4 shows parallel odd-even transposition sort plan with an example.

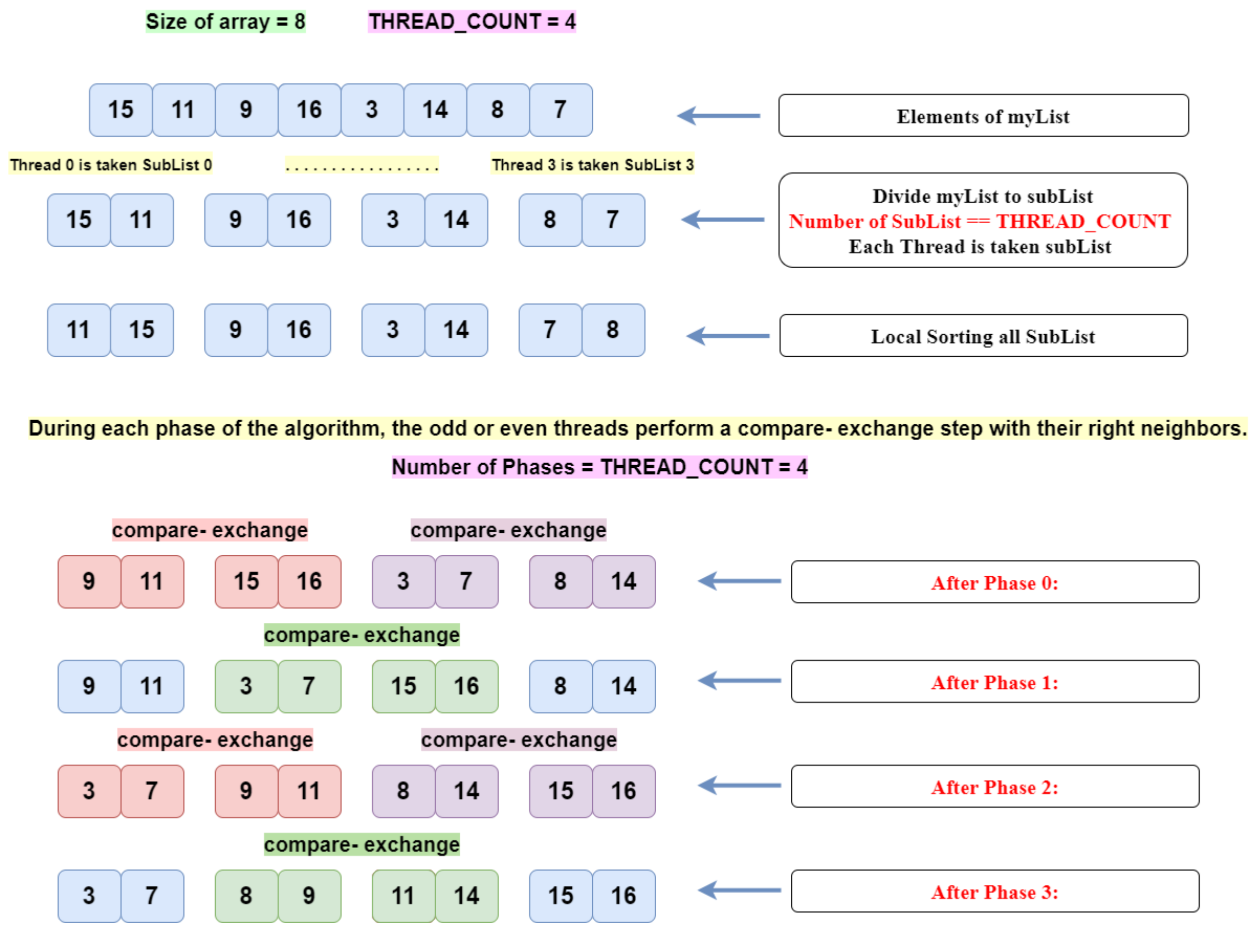


Figure 1.4 The parallel odd-even transposition sort plan

2 Parallel algorithm design

We designed the algorithm in parallel with Foster's design methodology. The steps are as follows[3]:

- **Partitioning:** Dividing computation and data.
- **Communication:** Sharing data between computations.
- **Agglomeration:** Grouping tasks to improve performance.
- **Mapping:** Assigning tasks to processors/threads.

2.1 OpenMP

In OpenMP we using a static array. The design of the algorithm in OpenMP is as follows[3]:

- **Partitioning:**

  Each thread performs n of phases (outer loop).
  For each new phase, the list is partitioned per thread (inner loops).
  In other words, the set of iterations is distributed among the threads.
  Each thread performs Compare-swap on independent set of iterations.

- **Communication:**

  Each thread executes the same code redundantly.

- **Agglomeration:**

  The phases are increased from 0 to n, odd phases compare all odd indices elements and even phases compare all even indices elements.

- **Mapping:**

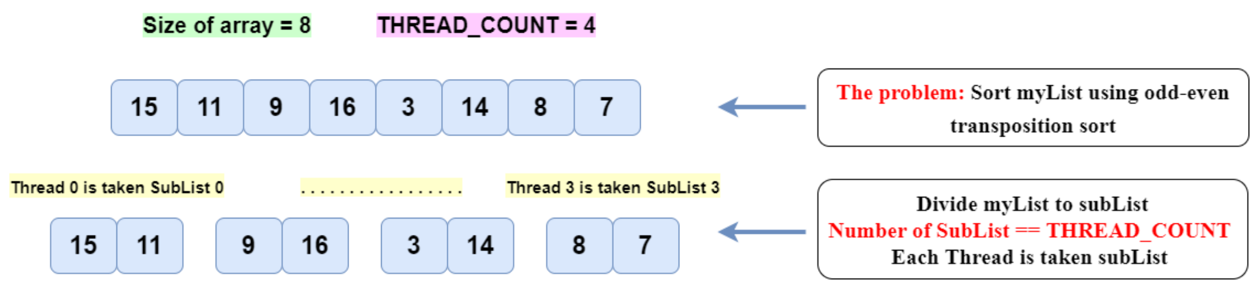
  For each new phase, the list is partitioned per thread so that no thread go performs Compare-swap for the same element at the same time.

2.2 Java Multithreading

In java mutithreading we using Array List. The design of the algorithm in Java Multithreading is as follows[3]:

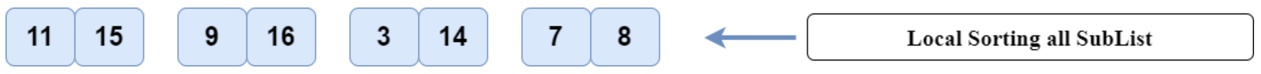
❖ **Partitioning:**

At the beginning we use data parallelism, so we divide the original myList we used to store the integer numbers into multiple subList. each thread will take one subList.


Size of array = 8
THREAD_COUNT = 4
15 11 9 16 3 14 8 7
The problem: Sort myList using odd-even transposition sort
Thread 0 is taken SubList 0
....
Thread 3 is taken SubList 3
15 11 9 16 3 14 8 7
Divide myList to subList
Number of SubList == THREAD_COUNT
Each Thread is taken subList

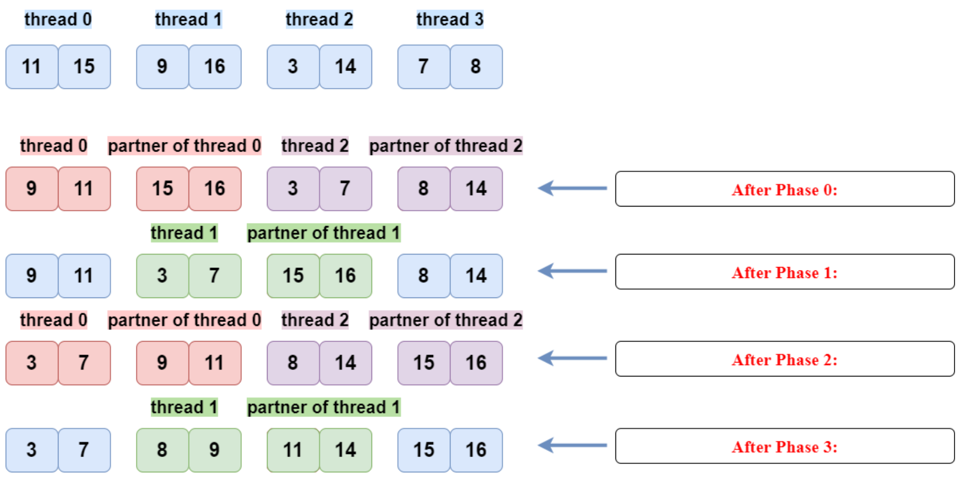
Primitive tasks:

1- **Local sort:** every thread going to sort its elements by using sort method in the Arraylist in the same time. This task you will be parallel.

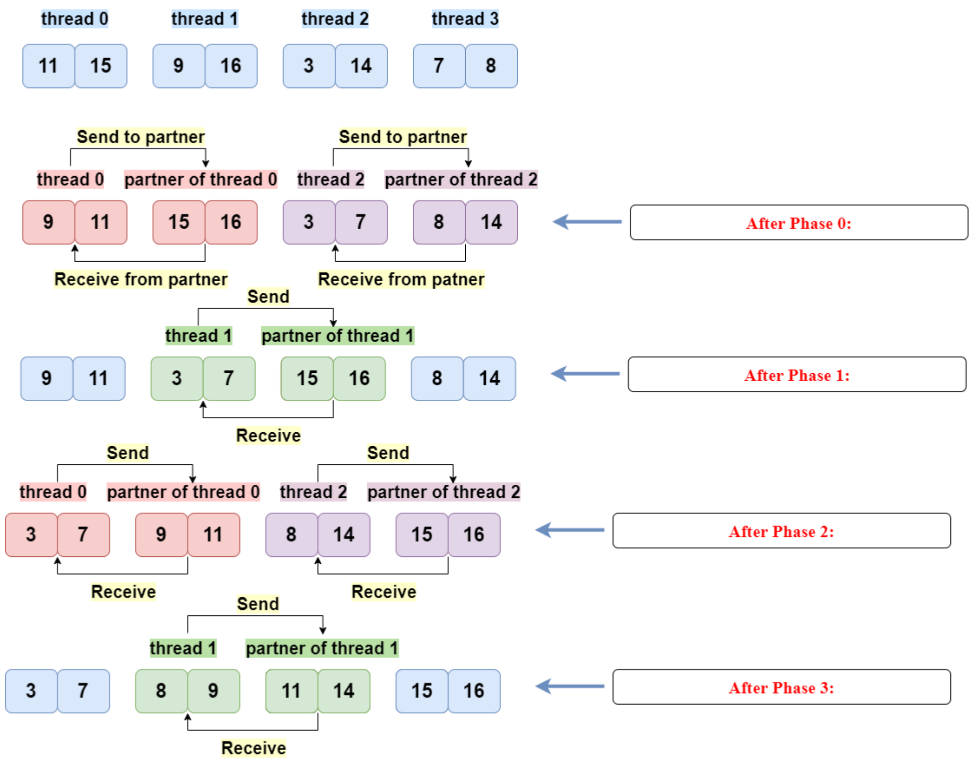


2- **Phases**: number of phases equal number of THREAD_COUNT. The transition from phase to another phase is done sequentially but the distribution of tasks at the same phase is similar to parallelism so that there will be communication between the threads. The tasks in the phase are as follows:

❖ **Compute partner:** If the phase is even and thread rank is even so the partner will be on the right, and in same case of even phase if the thread rank is odd the partner will be on the left. on the other hand, If the phase is odd and thread rank is even so the partner will be on the left, and in same case of odd phase if the thread rank is odd the partner will be on the right.

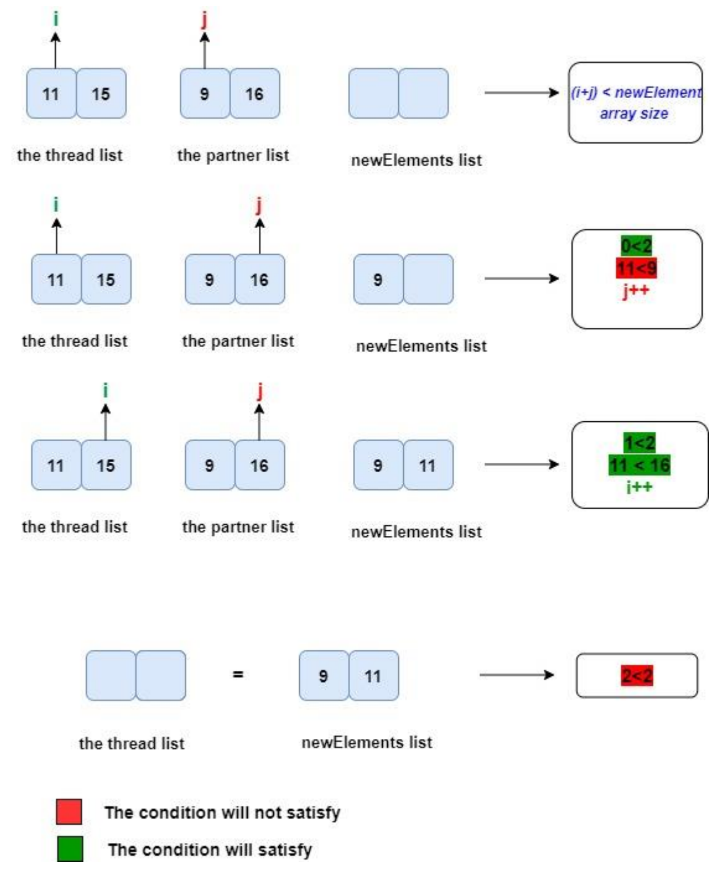


❖ **Send and receive:** Each thread send a copy of itself to partner thread. Each thread receive a copy of partner thread from partner.

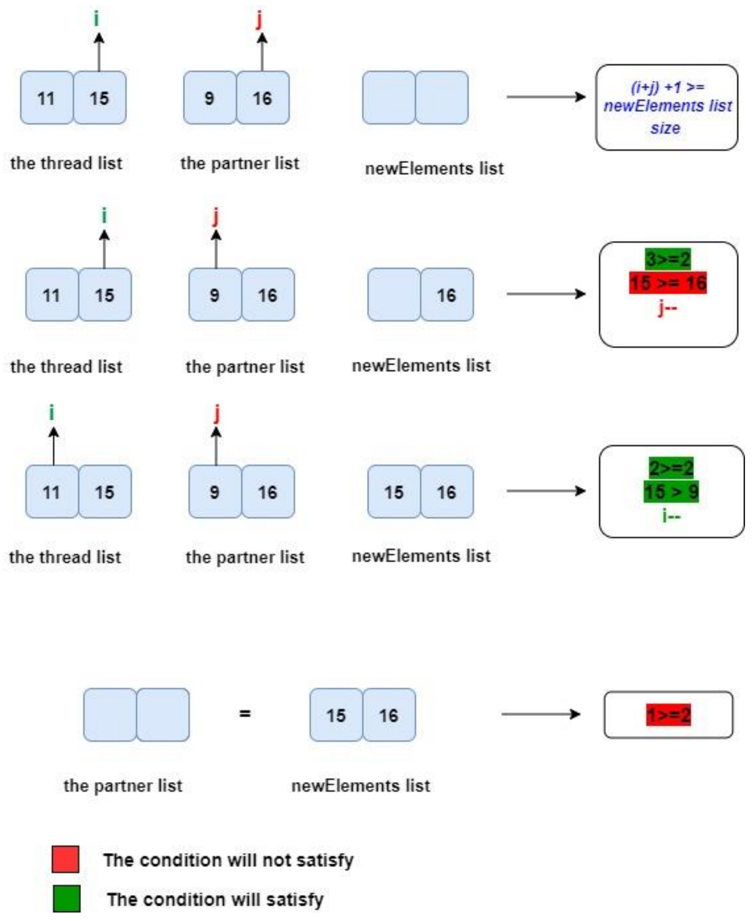


❖ **Compare-swap:** Each thread performs compare-swap with partner thread by select smaller/larger elements. if the thread rank smaller than the partner rank, then the thread rank will select the smaller elements and the partner will select the larger elements. vice versa.

- **Select smaller Elements:** We use two pointers (i , j) and new list (newElements). Pointer i refer to the first key element in thread list and pointer j refer to the first element in partner list. if (i + j) is smaller than the list size of the thread will do the following task until the condition become could not satisfy: if the thread element is a smaller than the partner element, then we add thread key to newElements list and move pointer i to the right. Else, then we add partner element to the list and move pointer j to the right.


i
j
11
15
9
16
the thread list
the partner list
newElements list
(i+j) < newElement array size
0<2
11<9
j++
1<2
11 < 16
i++
2<2
=
🟥 The condition will not satisfy
🟩 The condition will satisfy

- **Select larger keys:** We use two pointers (i , j) and new list (newElements). Pointer i refer to the last element in thread list and pointer j refer to the last element in partner list. if (i + j) +1 is grater than or equal to the list size of the thread will do the following task until the condition become could not satisfy: if the thread element is a grater than the partner element, then we add thread element to newElements list and move pointer i to the left. Else, then we add partner element to the list and move pointer j to the left.


i
j
11
15
9
16
the thread list
the partner list
newElements list
(i+j) +1 >= newElements list size
3>=2
15 >= 16
j--
2>=2
15 > 9
i--
=
1>=2
🟥 The condition will not satisfy
🟩 The condition will satisfy

❖ Communication:

Each thread send a copy of itself to partner thread. Each thread receive a copy of partner thread from partner.

❖ Agglomeration:

The phases are increased from 0 to THREAD_COUNT. Each phase depend to the result of the previous phase. Each thread performs its local sort and then performs tasks in each phase(Compute partner, Send and receive, Compare-swap).

❖ **Mapping:**

Give each thread a sublist that performs tasks on it. It performs the local sort and then performs tasks in phases.

3 Parallel code

In this code, a menu will appear for the user at the beginning when running the program, through which he chooses the appropriate option. The menu is as follows:

- `When click 1: will be arranged myList using Serial Odd-Even Variation to Bubble Sort.`
- `When click 2 will be arranged myList using Parallel Odd-Even Variation to Bubble Sort.`

3.1 OpenMP

For the full source code, please visit GitHub

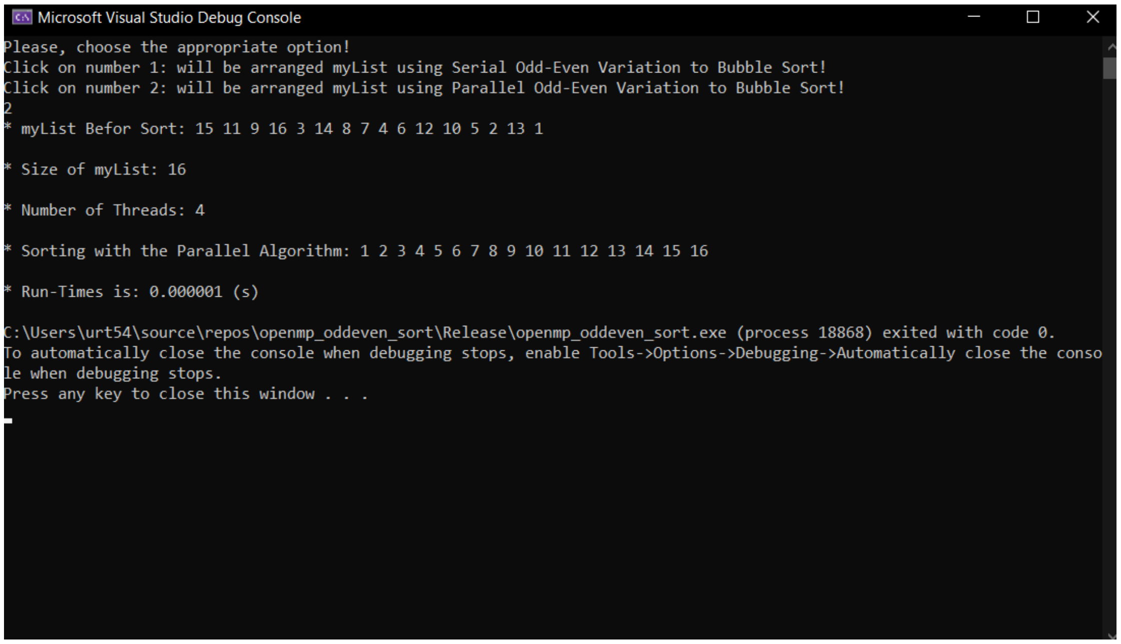
3.2 Java Multithreading

For the full source code, please visit GitHub

4 Sample output

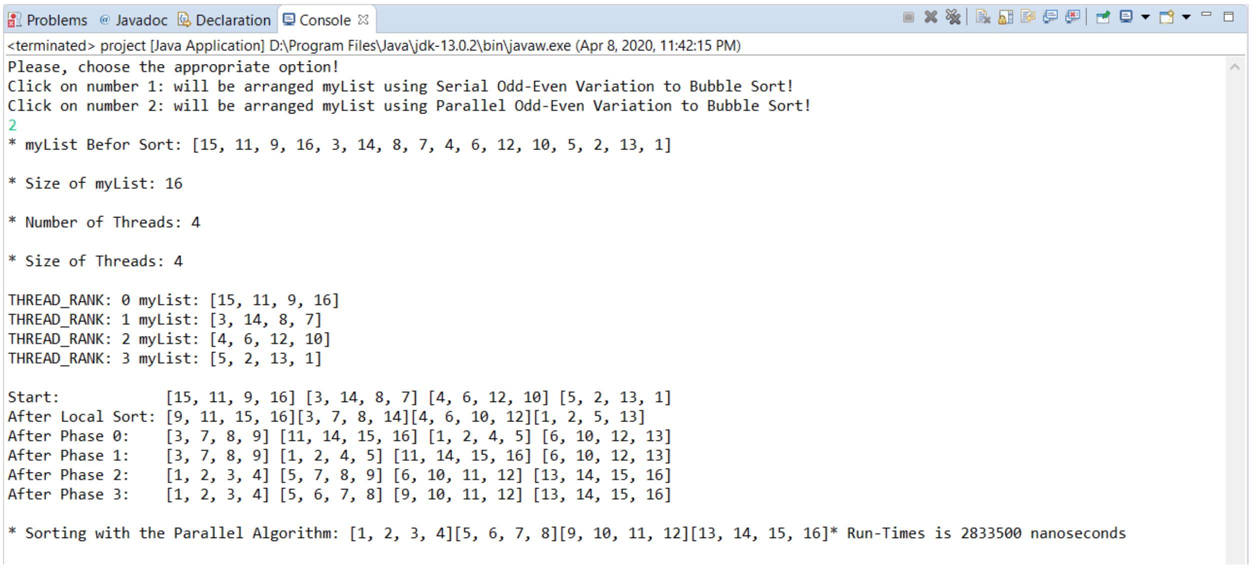
In this section we going to show the output of our work. we tested the parallel program by inserted small size of array.

4.1 OpenMP



```
Please, choose the appropriate option!
Click on number 1: will be arranged myList using Serial Odd-Even Variation to Bubble Sort!
Click on number 2: will be arranged myList using Parallel Odd-Even Variation to Bubble Sort!
2
* myList Befor Sort: 15 11 9 16 3 14 8 7 4 6 12 10 5 2 13 1

* Size of myList: 16

* Number of Threads: 4

* Sorting with the Parallel Algorithm: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

* Run-Times is: 0.000001 (s)

C:\Users\urt54\source\repos\openmp_oddeven_sort\Release\openmp_oddeven_sort.exe (process 18868) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

4.2 Java Multithreading



```
<terminated> project [Java Application] D:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (Apr 8, 2020, 11:42:15 PM)
Please, choose the appropriate option!
Click on number 1: will be arranged myList using Serial Odd-Even Variation to Bubble Sort!
Click on number 2: will be arranged myList using Parallel Odd-Even Variation to Bubble Sort!
2
* myList Befor Sort: [15, 11, 9, 16, 3, 14, 8, 7, 4, 6, 12, 10, 5, 2, 13, 1]

* Size of myList: 16

* Number of Threads: 4

* Size of Threads: 4

THREAD_RANK: 0 myList: [15, 11, 9, 16]
THREAD_RANK: 1 myList: [3, 14, 8, 7]
THREAD_RANK: 2 myList: [4, 6, 12, 10]
THREAD_RANK: 3 myList: [5, 2, 13, 1]

Start:           [15, 11, 9, 16] [3, 14, 8, 7] [4, 6, 12, 10] [5, 2, 13, 1]
After Local Sort: [9, 11, 15, 16][3, 7, 8, 14][4, 6, 10, 12][1, 2, 5, 13]
After Phase 0:    [3, 7, 8, 9] [11, 14, 15, 16] [1, 2, 4, 5] [6, 10, 12, 13]
After Phase 1:    [3, 7, 8, 9] [1, 2, 4, 5] [11, 14, 15, 16] [6, 10, 12, 13]
After Phase 2:    [1, 2, 3, 4] [5, 7, 8, 9] [6, 10, 11, 12] [13, 14, 15, 16]
After Phase 3:    [1, 2, 3, 4] [5, 6, 7, 8] [9, 10, 11, 12] [13, 14, 15, 16]

* Sorting with the Parallel Algorithm: [1, 2, 3, 4][5, 6, 7, 8][9, 10, 11, 12][13, 14, 15, 16]* Run-Times is 2833500 nanoseconds
```

5 Performance evaluation

In this section, we will evaluate performance with a different number of THREAD_COUNT and a different size of myList through the following steps:

1. Runtime's account of serial and parallel code.

   RunTime of paralle code = **End_Time_Parallel − Start_Time_Parallel**

   RunTime of Serial code = **End_Time_Serial − Start_Time_Serial**

2. Measure the relation between the serial and the parallel run-times is the speedup.

$$\mathbf{S(n,p)} = \frac{T_{serial(n)}}{T_{parallel(n,p)}}$$

3. Measure of parallel performance is parallel efficiency.

$$\mathbf{E(n,p)} = \frac{S(n,p)}{p}$$

5.1 OpenMP

The following table 5.1 shows the results of run-times of serial and parallel code (times are in milliseconds).

Table 5.1 The run-times of serial and parallel code (times are in milliseconds)

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	224,000
1	0.701218	2.883491	11.846134	30.677424
2	0.838583	3.425880	11.358377	35.145825
4	0.903003	3.197754	13.072254	31.694612
8	0.819613	3.072128	15.195023	24.681205
16	0.762665	3.224772	13.501419	34.143711

The following table 5.2 shows the speedups of parallel code.

Table 5.2 The speedups of parallel code

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	224,000
1	1	1	1	1
2	0.84	0.84	1.04	0.87
4	0.78	0.90	0.91	0.97
8	0.86	0.94	0.78	1.24
16	0.92	0.89	0.88	0.90

The following table 5.3 shows the efficiencies of parallel code.

Table 5.3 The efficiencies of parallel code

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	224,000
1	1	1	1	1
2	0.5	0.420	0.521	0.436
4	0.194	0.225	0.227	0.242
8	0.108	0.117	0.097	0.155
16	0.057	0.056	0.055	0.056

The scalability in openMP multithreading is weakly scalable because the efficiency decrease while the number of processes increases.

5.2 Java Multithreading

The following table 5.4 shows the results of run-times of serial and parallel code (times are in nanoseconds).

Table 5.4 The run-times of serial and parallel code (times are in nanoseconds)

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	320,000
1	4252223300	17013811100	131972207300	538804444700
2	740308700	2088349900	8685290500	27241524400
4	954267400	3341642400	11695431600	42258946200
8	1901935400	7270067500	18787680600	63753458700
16	3176230600	8938014000	33929118000	111998248200

The following table 5.5 shows the speedups of parallel code.

Table 5.5 The speedups of parallel code

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	320,000
1	1	1	1	1
2	0.57	8.15	15.19	19.8
4	0.45	5	11.3	12.8
8	0.22	2.34	7.02	8.45
16	0.13	1.90	3.9	4.8

The following table 5.6 shows the efficiencies of parallel code.

Table 5.6 The efficiencies of parallel code

	Size of myList			
THREAD_COUNT	40,000	80,000	160,000	320,000
1	1	1	1	1
2	0.285	4.1	7.6	9.9
4	0.133	1.25	2.8	3.2
8	0.027	0.29	0.88	1.06
16	0.008	0.12	0.24	0.3

The scalability in java multithreading is weakly scalable because the efficiency decrease while the number of processes increases.

References

[1] Retrieved from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting Algorithms/sorting.html

[2] Retrieved fro m http://parallelcomp.uw.hu/ch09lev1sec3.html

[3] Pacheco, P., "An Introduction to Parallel Programming", Morgan Kaufman. 2011.

[4] Liang, Y., “Introduction to java programming”, Prentice Hall, 2014.