*JKAU: Comp. IT.,* **Vol. 2**, pp: 27 - 44 (2013 A.D./ 1435 A.H.)

DOI: 10.4197 / Comp. 2-2

# A Novel Approach of Selection Sort Algorithm with Parallel Computing and Dynamic Programing Concepts

**Khaled Thabit** and **Afnan Bawazir**

*Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, P.O.Box. 80221, Jeddah 21589, Kingdom of Saudi Arabia.*
*afnan-bawazir@hotmail.com*

*Abstract.* Many research works have been conducted to find out better enhancement for Selection Sort Algorithm, such as bidirectional selection sort "Friend Sort Algorithm" which can position two elements in each round. We have improved this algorithm by using the concept of parallel computing. This algorithm is called Min-Max Bidirectional Parallel Selection Sort (MMBPSS). Also this paper proposes to use dynamic programming (stack) to reduce sorting time by increasing the amount of space. The basic idea behind using stack is to eliminate unnecessary iteration. This algorithm is called Dynamic Selection Sort "DSS". To fuse advantages of "DSS" with advantages of "MMBPSS", we suggest a new third algorithm called Min-Max Bidirectional Parallel Dynamic Selection Sort "MMBPDSS". It can position two elements: minimum and maximum from two directions using Dynamic Selection Sort algorithm in each round in parallel, thus reducing the number of loop required for sorting. Results obtained after implementation are provided in graphical form with an objective to show that "MMBPDSS" is saving almost 50% of classical selection sorting time and ensure accuracy.

*Keywords*: *High Performance Computing, Selection Sort, Bidirectional selection sort, parallel computing, Dynamic Programming.*

## 1. Introduction

Sorting is a technique by which elements are arranged in a particular order following some characteristic or law[1]. Data can be in numerical or character form. There are a lot of sorting techniques, currently used in industrial applications and academic researches, to arrange the data of various forms and from different areas. Sorting is of considerable importance as the human is possessed in keeping the ordered information/knowledge. To search the information efficiently the arrangement of data is very important. To facilitate the human, computers consume a massive amount of time in ordering the data[2]. There are a lot of sorting algorithms used nowadays such as Bubble Sort, Insertion Sort, Selection Sort and Cocktail sort. Every kind of sort has its own pros and cons, and the pattern of input data is a major factor for its performance. This paper focuses on Selection Sort algorithm which has performance advantages over more complicated ones in certain situations, especially where auxiliary memory is limited. It does many comparisons and least amount of data swapping. Selection Sort algorithm is inefficient on large lists, because it has O (n2) complexity, and generally performs worse than the similar Insertion Sort. Many research works have been conducted to find out better enhancement for Selection Sort [1, 3-5] that speed up the sorting process such as bidirectional Selection Sort Algorithm, which can position two items in each pass thus reducing the number of loops required for sorting. This algorithm also called "Friends Sort"[3]. Lakra and Divy[4] suggested "Double Selection Sort" which makes sorting an efficient and convenient way for larger data set by saving almost 25% to 35% than the classic Selection Sort Algorithm. We have improved "Friends Sort" algorithm by making it working in parallel. This algorithm is called Min-Max Bidirectional Parallel Selection Sort (MMBPSS). Also other study proposes an improvement for Selection Sort Algorithm by using dynamic programming technique (Stack). The key idea behind using stack is to eliminate unnecessary passes by reducing the number of comparison. A Stack is used to store the location of previous max element found, and instead of starting from the beginning each time, the largest element is found and placed at the end of the array. This algorithm is called Dynamic Selection Sort "DSS". We suggest a new third algorithm called Min-Max Bidirectional Parallel Dynamic Selection Sort "MMBPDSS" which combine DSS and MMBPSS. Our hypothesis "MMBPDSS" makes sorting an efficient and convenient way for smaller and larger data set by saving almost 50% than the classic Selection Sort and Friend Sort algorithms [3] due to the parallel implementation of the algorithm.

The paper is organized as follows: a brief review of selection sorting algorithm are discussed in Section 2, while section 3 contains the proposed algorithm "Min-Max Bidirectional Parallel Selection Sort" and explained it in more details, the steps, and procedure with an example. Section 4 presents second proposed algorithm “Dynamic Selection Sort” in details with procedures using example. Furthermore, Section 5 contains third proposed algorithm “Min-Max Bidirectional Parallel Dynamic Selection Sort” in details with steps and procedure using example. This paper further progress in Section 6 by testing and analyzing the proposed algorithm’s results with the classical Selection Sorting and the new Friend Sorting technique[3]. Finally the paper concluded in Section-7.

## 2. Brief Review of Selection Sorting Algorithm

This section presents a review of Selection Sort including history of formation, methodologies and algorithms.

### *2.1 Selection Sort*

Selection Sort is a well-known sorting technique that scans an array to find the maximum item, puts it at the last location in the array, and then scans the array for the second maximum item, puts it before the last location, then third maximum and so forth, until reaches the smallest item to be put at the first location of the array. It has $O(n^2)$ complexity, inefficient for larger lists or arrays and its performance is worse than that of Insertion Sort. In certain situations, it has a prominent efficiency than some other convoluted algorithms. The number of passes, of the Selection Sort for a given list, is equal to the number of elements in that list.[6] The number of interchanges and assignments depends on the original order of the items in the list/array, but the sum of these operations does not exceed a factor of $n^2$[7].

### *2.2 Procedure for Selection Sort*

```
Procedure Selection-Sort (List: List of items to be sorted)
        Length ← length (List);
        For i ← Length -1 to 1 do
                Max ← i;
                For j ← i - 1 to 0 do
                        if( List[ j ] > List[ Max ] )
                                Max ← j
                        End if
```

```
            End for
            Swap (List[i], List [Max]);
    End for
    End Procedure
```

### *2.3 Min-Max Bidirectional Parallel Selection Sort*

Min-Max Bidirectional Parallel Selection Sort (MMBPSS) is an improvement on the idea of traditional Bidirectional Selection Sort and Friend Sort Algorithms[3] which can position two elements in each round parallel, thus reduces the number of loop required for sorting. The basic design idea of the (MMBPSS) is as follows: it divides the list into two parts, minimum and maximum values from each part are selected in each sort round. Then both values of minimum and maximum from each part are compared to determine the minimum and the maximum of the whole array, and they are placed at their proper locations. The Steps of the proposed algorithm are as follows:

1. Divide the array into two.

Now: working in parallel from 2 to 7:

2. Find minimum and maximum values from each part.

3. Take minimum1 of the first part, compare it with minimum2 of the second part.

4. Swap and put them at their exact location.

5. Take maximum1 of the first part, compare it with maximum2 of the second part.

6. Swap and put them at their exact location.

7. Repeat these steps for the whole array.

### *2.4 Procedure for MMBPSS*

```
Procedure MMBPSS (List: List of items to be sorted)
        Length ← length (List);
        Max, Min;
Mid = Length/2;
Start = 0, end = Length-1;

            For i← Start to end do in parallel
                        () =>
```

```
          For j← Start to mid-1
                      if( List[ j ]  <  List[ Min1 ] )
                            Min1 ← j
                      End if
                      if( List[ j ]  >  List[ Max1 ] )
                            Max1 ← End
                      End if
               End for
                  () =>
        For K← mid to end
                      if( List[ k ]  <  List[ Min2 ] )
                            Min2 ← k
                      End if
                      if( List[ k ]  >  List[ Max ] )
                            Max2 ← k
                      End if
If (List [max1] >= List [max2])
       Max = max1;

      Else
        Max = max2;
End if
Swap (List[i], List [max]);

      If (List [min1] <= List [min2])
      Min = min1;
      Else
      Min = min2;
End if  Swap (List[ i ], List [ min ] );
      End for
End Procedure
```

## *2.5 Example for MMBPSS*

Let us take an array as an example see (Figure. 1) to apply (MMBPSS) on it:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **5** | **33** | **8** | **41** | **19** | **2** | **50** | **1** | **7** | **20** |

**Fig. 1. Unsorted Array**

Index of Mid=5

Each for loop work at one part to find min & max in parallel,

See (Figure. 2)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **5** | **33** | **8** | **41** | **19** | **2** | **50** | **1** | **7** | **20** |

**Fig. 2. Divide Array into two parts**.

First iteration: see (Figure. 3)

Min1= 5 Min2=1

Max1=41 Max2=50

Then compare them:

Min 1> Min 2 $\rightarrow$ Min= Min 2= 1

Max 1< Max 2 → Max= Max2 = 50

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **33** | **8** | **41** | **19** | **2** | **20** | **5** | **7** | **50** |

**Fig. 3. Array after the first iteration.**

Second iteration: see (Figure. 4)

Min1= 8 Min2=2

Max1=41 Max2=20

Then compare them:

Min 1> Min 2 → Min= Min 2= 2

Max 1> Max 2 → Max= Max1 = 41

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **8** | **7** | **19** | **33** | **20** | **5** | **41** | **50** |

**Fig. 4. Array after the second iteration**.

Third iteration: see (Figure. 5)

Min1= 7 Min2=5

Max1=19 Max2=33

Then compare them:

Min 1> Min 2 → Min= Min 2= 5

Max 1< Max 2 → Max= Max2 = 33

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **5** | **7** | **19** | **8** | **20** | **33** | **41** | **50** |

**Fig. 5. Array after third iteration**.

Fourth iteration: see (Figure. 6)

Min1= 7 Min2=8

Max1=19 Max2=20

Then compare them:

Min 1< Min 2 → Min= Min 1= 7

Max 1< Max 2 → Max= Max2 = 20

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **5** | **7** | **19** | **8** | **20** | **33** | **41** | **50** |

**Fig. 6. Array after fourth iteration.**

Fifth iteration: see (Figure. 7)

Min1= 19 Min2=8

Max1=19 Max2=8

Then compare them:

Min 1> Min 2 → Min= Min 2= 8

Max 1> Max 2 → Max= Max1 = 19

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **5** | **7** | **8** | **19** | **20** | **33** | **41** | **50** |

**Fig. 7. Array after fifth iteration.**

Finally, the array is sorted. See (Figure. 7)

### *2.6 Dynamic Selection Sort*

Dynamic Selection Sort (DSS) is an improvement on the idea of Selection Sort but it used dynamic programming (stack) to reduce sorting time by increasing the amount of space. The basic idea behind using stack is to eliminate unnecessary iterations. A stack is used to store the location of previous largest element found, instead of starting from the beginning after the largest element is found and placed at the end of the array, we pop the stack and start at the location of the previous max, so we can decrease the number of comparison required for sorting operation. Here we used two Stacks, one to store the location of the previous index largest element, another one to store value. Once the location and value of the previous largest element is popped off from the stack and compared with the elements of the array from that location till a new largest element is found or the new end of the array is reached. If a larger element is encountered then the location and value of the previous largest element is pushed into the stack. The new largest element is again compared to the remaining elements of the array. This process is repeated until the array is sorted.

### *2.7 Procedure for DSS*

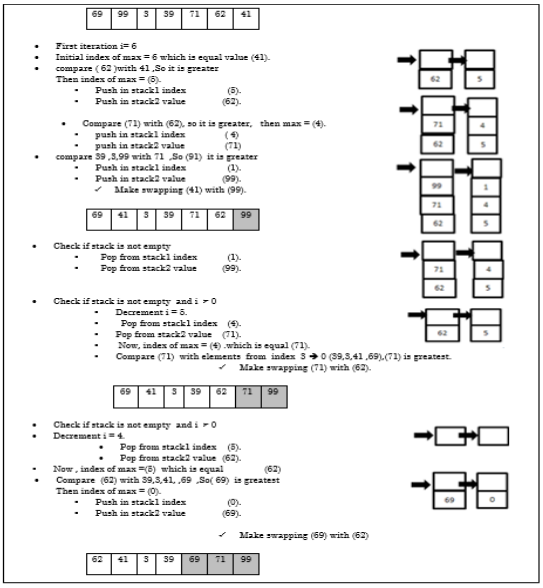
**Procedure DynamicSelectionSort (List: List of items to be sorted)**

```
    Length ← length (List); Max, Location, Value, Stack1, Stack2;
    For i ← Length -1 to 1 do
        Max ← i;
        For j ← i - 1 to 0 do
            if( List[ j ] > List[ Max ] )
                Max ← j
                Push Max in Stack1
                Value ← List [Max]
                Push Value in Stack 2
            End if
        End for
        Swap (List[ i ], List[Max] );
        Pop the first element from both the stack //this element
already has been swapped
        While (Stacks are not empty && i > 0)
            i ← i - 1
            Location ← pop element from Stack1, Max ←
Location
            Value ← pop element from Stack2.
            Swapped ← false
```

```
                For n ← Location - 1 to 0 && i - 1 do
                    Swapped ← true;
                    if ( List[ n ]  > Value )
                        Max ← n
                        Push Max in Stack1
                        Value ← List [Max]
                        Push Value in Stack 2
                    End if
                End for
                If (Swapped )
                    Swap (List[ i ], List[Max] )
                    Pop the first element from both the stack
                Else
                    i ← i + 1
            End while // Stack count loop
        End for // outer for loop
End Procedure
```

## *2.8 Example for DSS*

Let us take an array as an example (Fig.8 and 9) to apply (DSS) on it:



| | | | | | | |
|---|---|---|---|---|---|---|
| 69 | 99 | 3 | 39 | 71 | 62 | 41 |

- First iteration i= 6
- Initial index of max = 6 which is equal value (41).
- compare ( 62 )with 41 ,So it is greater
  Then index of max = (5).
  - Push in stack1 index (5).
  - Push in stack2 value (62).
- Compare (71) with (62), so it is greater, then max = (4).
  - push in stack1 index ( 4)
  - push in stack2 value (71)
- compare 39 ,3,99 with 71 ,So (91) it is greater
  - Push in stack1 index (1).
  - Push in stack2 value (99).
  - ✓ Make swapping (41) with (99).

| | | | | | | |
|---|---|---|---|---|---|---|
| 69 | 41 | 3 | 39 | 71 | 62 | 99 |

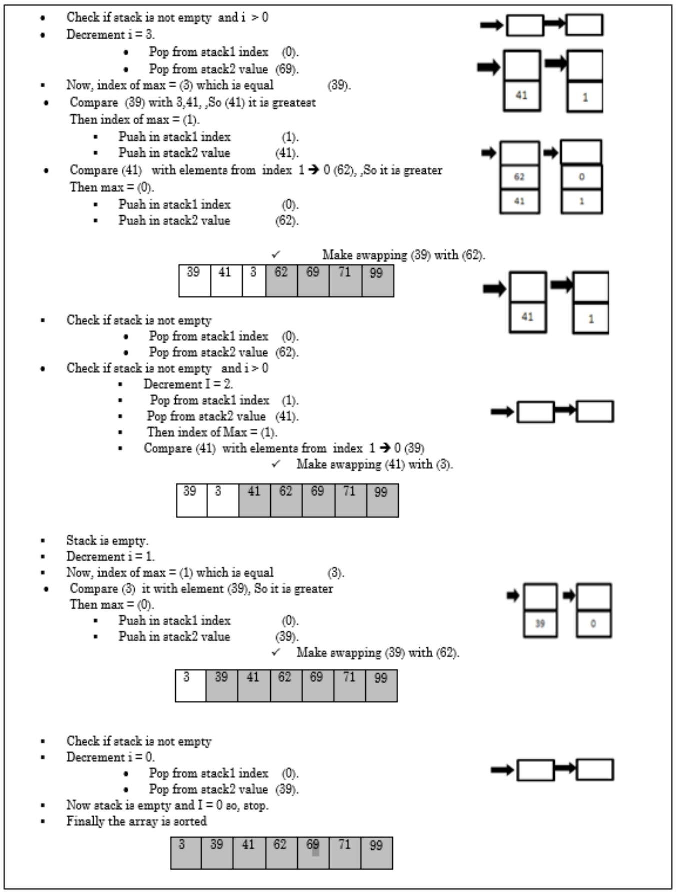
- Check if stack is not empty
  - Pop from stack1 index (1).
  - Pop from stack2 value (99).
- Check if stack is not empty and i ≻ 0
  - Decrement i = 5.
  - Pop from stack1 index (4).
  - Pop from stack2 value (71).
  - Now, index of max = (4) .which is equal (71).
  - Compare (71) with elements from index 3 ➔ 0 (39,3,41 ,69),(71) is greatest.
    - ✓ Make swapping (71) with (62).

| | | | | | | |
|---|---|---|---|---|---|---|
| 69 | 41 | 3 | 39 | 62 | 71 | 99 |

- Check if stack is not empty and i ≻ 0
- Decrement i = 4.
  - Pop from stack1 index (5).
  - Pop from stack2 value (62).
- Now , index of max =(5) which is equal (62)
- Compare (62) with 39,3,41, ,69 ,So( 69) is greatest
  Then index of max = (0).
  - Push in stack1 index (0).
  - Push in stack2 value (69).
  - ✓ Make swapping (69) with (62)

| | | | | | | |
|---|---|---|---|---|---|---|
| 62 | 41 | 3 | 39 | 69 | 71 | 99 |

Fig. 8. DSS algorithm by way of an example.



- Check if stack is not empty and i > 0
- Decrement i = 3.
    - Pop from stack1 index (0).
    - Pop from stack2 value (69).
- Now, index of max = (3) which is equal (39).
- Compare (39) with 3,41, ,So (41) it is greatest

  Then index of max = (1).
    - Push in stack1 index (1).
    - Push in stack2 value (41).
- Compare (41) with elements from index 1 ➔ 0 (62), ,So it is greater

  Then max = (0).
    - Push in stack1 index (0).
    - Push in stack2 value (62).

✓ Make swapping (39) with (62).

| 39 | 41 | 3 | 62 | 69 | 71 | 99 |
|---|---|---|---|---|---|---|

- Check if stack is not empty
    - Pop from stack1 index (0).
    - Pop from stack2 value (62).
- Check if stack is not empty and i > 0
    - Decrement I = 2.
    - Pop from stack1 index (1).
    - Pop from stack2 value (41).
    - Then index of Max = (1).
    - Compare (41) with elements from index 1 ➔ 0 (39)

      ✓ Make swapping (41) with (3).

| 39 | 3 | 41 | 62 | 69 | 71 | 99 |
|---|---|---|---|---|---|---|

- Stack is empty.
- Decrement i = 1.
- Now, index of max = (1) which is equal (3).
- Compare (3) it with element (39), So it is greater

  Then max = (0).
    - Push in stack1 index (0).
    - Push in stack2 value (39).

      ✓ Make swapping (39) with (62).

| 3 | 39 | 41 | 62 | 69 | 71 | 99 |
|---|---|---|---|---|---|---|

- Check if stack is not empty
- Decrement i = 0.
    - Pop from stack1 index (0).
    - Pop from stack2 value (39).
- Now stack is empty and I = 0 so, stop.
- Finally the array is sorted

| 3 | 39 | 41 | 62 | 69 | 71 | 99 |
|---|---|---|---|---|---|---|

**Fig. 9. DSS algorithm by way of an example "cont".**

## 3 Min Max Bidirectional Parallel Dynamic Selection Sort

Min-Max Bidirectional Parallel Dynamic Selection Sort (MMBPDSS) is an improvement on the idea of Dynamic Selection Sort which can position two elements, minimum and maximum, from two directions in each round in parallel. It thus reduces the number of loop required for sorting. The Steps of the (MMBPDSS) are as follows

1. First thread starts from the beginning of the array which finds the smallest element (using 2 stacks to store the minimum and its location).

2. The second thread starts from the end of the array searching for the largest element (using 2 stacks to store the maximum and its location).
3. Then concatenate the first half from first thread with the second half from the second thread.

### *3.1 Procedure for MMBPDSS*

**ProcedureMinMaxBidirectionalParallelDynamicSelectionSort (List: List of items to be sorted)**
**First Thread: //** sorting smallest elements

```
Length ← length (List); Min, Location, Value, Stack1, Stack2, mid = Length / 2;
    For i ← 0 to mid-1 do
        Min ← i;
        For j ← i - 1 to 0 do
            if( List[ j ] < List[ Min ] )
                Min ← j
                Push Min in Stack1
                Value ← List [Min]
                Push Value in Stack 2
            End if
        End for
        Swap (List[i], List [Min]);
        Pop the first element from both the stack //this element already has been swapped
        While (Stacks are not empty && i <= mid)
            i ← i + 1
            Location ← pop element from Stack1, Min← Location
            Value ← pop element from Stack2.
            Swapped ← false
            For n ← Location - 1 to 0 && i - 1 do
                Swapped ← true;
                If (List[n] < Value)
                    Min ← n
                    Push Min in Stack1
                    Value ← List [Min]
                    Push Value in Stack 2
                End if
            End for
            If (Swapped)
                Swap (List[i], List [Min])
```

```
                        Pop the first element from both the stack
                    Else
                        i ← i - 1
                End while // Stack count loop
        End for // outer for loop
End first thread.
```

**Second Thread: //** sorting largest elements

```
Length ← length (List); Max, Location, Value, Stack1, Stack2, mid = Length
/ 2;
        For i ← mid to 1 do
                Max ← i;
                For j ← i - 1 to 0 do
                    if( List[ j ] > List[ Max ] )
                        Max ← j
                        Push Max in Stack1
                        Value ← List [Max]
                        Push Value in Stack 2
                    End if
                End for
                Swap (List[i], List [Max]);
                Pop the first element from both the stack //this element
already has been swapped
                While (Stacks are not empty && i >= mid)
                    i ← i - 1
                    Location ← pop element from Stack1, Max ←
Location
                    Value ← pop element from Stack2.
                    Swapped ← false
                    For n ← Location - 1 to 0 && i - 1 do
                        Swapped ← true;
                        If (List[n] > Value)
                            Max ← n
                            Push Max in Stack1
                            Value ← List [Max]
                            Push Value in Stack 2
                        End if
                    End for
                    If (Swapped)
                        Swap (List[i], List [Max])
                        Pop the first element from both the stack
                    Else
                        i ← i + 1
                End while // Stack count loop
        End for // outer for loop
```

*End second thread.*
**End Procedure.**

### *3.2 Example for MMBPDSS*

Let us take an array as an example (Figure. 10) to apply (MMBPDSS) on it:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 10 | 6 | 1 | 8 | 9 | 12 | 11 | 7 | 3 | 4 |

Working in parallel
First thread sorting smallest element using Dynamic Selection Sort

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 6 | 7 | 9 | 12 | 11 | 7 | 10 | 8 |

Second thread sorting largest element using Dynamic Selection Sort

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 6 | 4 | 1 | 3 | 7 | 8 | 9 | 10 | 11 | 12 |

Then concatenate first half form first thread with second half from second thread.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 6 | 7 | 7 | 8 | 9 | 10 | 11 | 12 |

## 4. RESULTS AND DISCUSSION

To prove efficiency of MMBPSS, DSS and MMBPDSS performance, they were implemented along with Classical Selection Sort algorithm and with the new Friend Sorting Algorithm[3]. The calculation of average execution time, total comparison and swapping frequencies are conducted for random sample lists with different sizes, 30 times for all mentioned algorithms in the paper. We have conducted those algorithms by using basic laptop with the following specification Intel Core2Duo processor 2.53 GHZ machine with 4 GB.

The results are accomplished in three ways:
1. Comparison of execution time of MMBPSS, DSS and MMBPDSS algorithms with the classic Selection Sort algorithm and with the new Friend Sorting algorithm.
2. Comparison of total frequency of MMBPSS, DSS and MMBPDSS with the classic Selection Sort algorithm and new Friend Sorting algorithm.
3. Comparison of total swapping frequency of MMBPSS, DSS and MMBPDSS with the classic Selection Sort algorithm and with the new friend sorting algorithm.

## *4.1 Comparison of Execution Time*

Table 1 shows the comparison of the MMBPSS, DSS and MMBPDSS algorithm with the classic Selection Sort algorithm and with the new Friend Sorting algorithm with respect to the average execution time each algorithm takes to perform sorting.

**Table 1. Time Comparison**

| Array size | Average of Execution Time (Test on different array of size)in millisecond | | | | |
|---|---|---|---|---|---|
| | Selection Sort | New Friend Sorting Algorithm | Min-Max Bidirectional Parallel Selection Sort | Dynamic Selection Sort | Min-Max Bidirectional Parallel Dynamic Selection Sort |
| 1000 | 13.44133333 | 11.99764 | 36.87646667 | 10.43051 | 11.25 |
| 3000 | 44 | 45 | 117 | 37 | 31 |
| 5000 | 206.7171 | 169.3317 | 192.08728 | 140.6054 | 87.25 |
| 7000 | 213.9861 | 195.2054 | 198.8224 | 164.0168 | 125.4608 |
| 10000 | 693.21615 | 665.3608 | 528.2458 | 394.7421 | 299.2226 |
| 30000 | 4224.902 | 4244.429 | 3630.884 | 3498.265 | 2657.741 |
| 50000 | 11916.28 | 12019.771 | 9281.721 | 8580.618 | 6537.066 |
| 70000 | 18795.62 | 19479.03 | 16521.67 | 14323.28 | 12189.42 |
| 100000 | 53993.07 | 52417.33 | 31804.16 | 35361.93 | 27735.59 |

Graphical view for Table.1 is presented in Figure. 11.

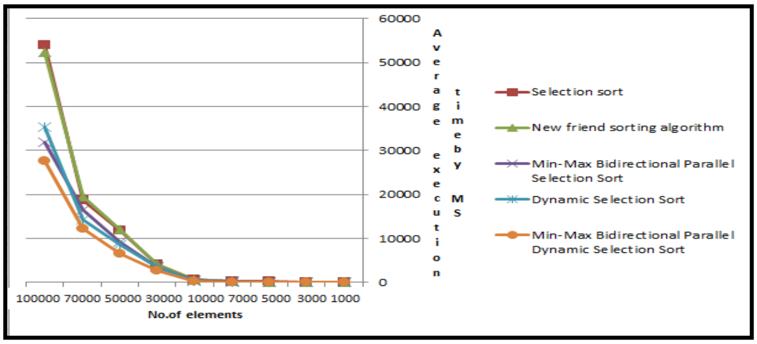


**Fig. 11. Time Comparison.**

It can be observed from Fig. 11 that the performance of the new Friend Sorting algorithm is less efficient when the array size is smaller than 30000 but after that its efficiency degrades and it is equally efficient to the classic Selection Sort but MMBPSS is more efficient when the array size is over 35000 elements. There is an additional overhead when applying MMBPSS on smaller array size. DDS reduces the execution time compared to the classic Selection Sort, the new Friend Sorting algorithm and MMBPSS, while "MMBPDSS" is better than DDS and saves almost 50% of the classical Selection Sorting. It really reaches the optimization purpose.

### *4.2 Comparison of Total Comparison Frequency*

Table.2 shows the comparison of the MMBPSS, DSS and MMBPDSS algorithms with the classic Selection Sort algorithm and with the new Friend Sorting algorithm with respect to average of comparison numbers each algorithm takes to perform sorting.

**Table 2. Total Comparison Frequency.**

| Array size | Average of comparison numbers (Test on different array of size) | | | | |
|---|---|---|---|---|---|
| | Selection sort | New friend sorting algorithm | Min-Max Bidirectional Parallel Selection Sort | Dynamic Selection Sort | Min-Max Bidirectional Parallel Dynamic Selection Sort |
| 1000 | 499500 | 500500 | 10871 | 415369 | 621162 |
| 3000 | 4498500 | 4501500 | 35597 | 3683523 | 3929732 |
| 5000 | 12497500 | 12502500 | 69541 | 10096142 | 9745052 |
| 7000 | 24496500 | 24503500 | 97718 | 19993926 | 21634554 |
| 10000 | 49995000 | 50005000 | 151299.3 | 40832313 | 39379276 |
| 30000 | 449985000 | 450015000 | 490405 | 368623040 | 303589600 |
| 50000 | 1249975000 | 1250025000 | 880382 | 1031800452 | 1018170886 |
| 70000 | 2449965000 | 2450035000 | 1319601 | 2032017833 | 2256552865 |
| 100000 | 4999950000 | 5000050000 | 1926387 | 4186028005 | 4102486376 |

Graphical view for Table.2 is presented in Figure. 12.

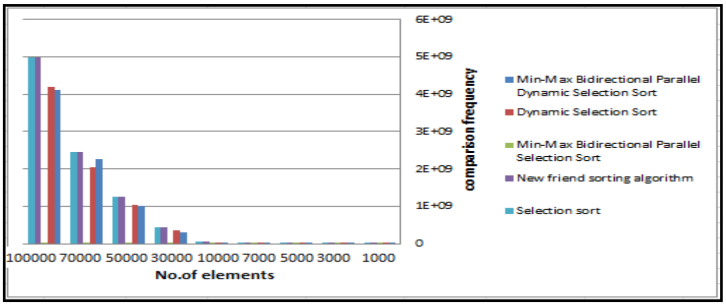


**Fig. 12. Total Comparison Frequency.**

It can be observed from the above graph that the total comparison frequency of Selection Sort and the new Friend Sorting algorithms are the same, while DSS and MMBPDSS reduce the total comparison frequency but MMBPSS perform the least number of comparisons in sorting procedure.

### *4.3 Comparison of Total Swapping Frequency*

Table.3 shows the comparison of the MMBPSS, DSS and MMBPDSS algorithms with the classic Selection Sort and the new Friend Sorting algorithms with respect to average of swapping frequency each algorithm takes to perform sorting.

**Table 3. Total Swapping Frequency**

| **Array size** | Average of swapping numbers (Test on different array of size) | | | | |
|---|---|---|---|---|---|
| | Selection sort | New friend sorting algorithm | Min-Max Bidirectional Parallel Selection Sort | Dynamic Selection Sort | Min-Max Bidirectional Parallel Dynamic Selection Sort |
| 1000 | 999 | 1000 | 994 | 999 | 1003 |
| 3000 | 2999 | 3000 | 2993 | 2999 | 3000 |
| 5000 | 4999 | 5000 | 4991 | 4999 | 5002 |
| 7000 | 6999 | 7000 | 6991 | 6999 | 6999 |
| 10000 | 9999 | 10000 | 9988 | 9999 | 10002 |
| 30000 | 29999 | 30000 | 29990 | 29999 | 30000 |
| 50000 | 49999 | 50000 | 49989 | 49999 | 50002 |
| 70000 | 69999 | 70000 | 69987 | 69999 | 70000 |
| 100000 | 99999 | 100000 | 99980 | 99999 | 100000 |

Graphical view for Table.3 is presented in Figure. 13.

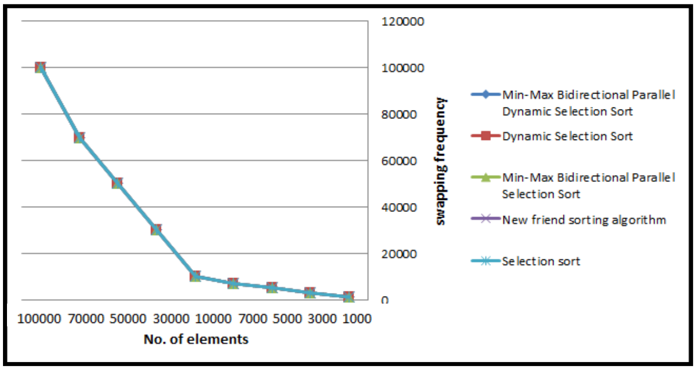


**Fig. 13. Total Comparison Frequency.**

It can be observed from the Figure. 13 that the classic Selection Sort, the new Friend Sorting Algorithms, MMBPSS, DSS, MMBPDSS perform the same number of swaps as the number of elements to perform sorting.

## 5. CONCLUSION

In this study, we present three new sorting techniques: "MMBSS","DSS" and "MMBPDSS" for selection sort that are tested and analyzed against the classical Selection Sorting and the new Friend Sorting techniques[3] to provide their efficiency. The graphs show that "MMBPDSS" save almost 50% of the classical Selection Sorting with 100% accuracy of order which get the benefit from effective utilization of CPU by using parallel computing with cost of increasing amount of space.

### References

[1] **Min, W.,** “Design and analysis on bidirectional selection sort algorithm,” in *Education Technology and Computer (ICETC), 2nd International Conference* on, Vol. **4**, pp: V4–380, (2010).

[2] **Bailey**, **D. A.,** Java Structure: *Data Structure in Java for Principled Programmer*, 2nd ed. McGraw-Hill, (2003).

[3] **Iqbal, S. Z., Gull, H.** and **Muzaffar, A. W.,** A new friends sort algorithm. *In Computer Science and Information Technology, 2nd IEEE International Conference on*. pp: 326-329, ICCSIT, (2009).

[4] **Lakra, S.** and **Divy**, "Improving the performance of selection sort using a modified double-ended selection sorting", *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, Volume 2, Issue 5, and May (2013).

[5] **Agarwal, A., Pardesi, V.** and **Agarwal**, **N.,** " A New Approach To Sorting: Min-Max Sorting Algorithm", *International Journal of Engineering Research & Technology (IJERT)* Vol. **2** Issue 5, May (2013).

[6] **Donald, E. K.,** The art of computer programming, *Sorting and searching*, **3**, 426-458.

[7] **Lipschutz**, **S.,** Theory and Problems of Data Structures, Schaum's Outline Series: *International Edition*, McGraw (1999).

# تحسين خوارزمية ترتيب "الاختيار" باستخدام مفهومي الحوسبة المتوازية والبرمجة الديناميكية

**خالد ثابت، وأفنان باوزير**

*قسم علوم الحاسبات، كلية الحاسبات وتقنية المعلومات*

*جامعة الملك عبدالعزيز، جدة، المملكة العربية السعودية*

afnan-bawazir@hotmail.com

*المستخلص*. قد أجريت العديد من الأعمال البحثية لاكتشاف أفضل تحسين لخوارزمية ترتيب "الاختيار"، مثل خوارزمية ترتيب الاختيار الثنائية الاتجاه كخوارزمية "ترتيب الاختيار الصديقة" والتي يمكن وضع عنصرين في كل جولة، لقد قمنا بتحسين هذه الخوارزمية باستخدام مفهوم الحوسبة المتوازية، هذه الخوارزمية تسمى أصغر–أكبر ثنائي الاتجاه المتوازية للترتيب بواسطة الاختيار (MMBPSS). كما تقترح هذه الورقة استخدام البرمجة الديناميكية (المكدس) لتقليل وقت الفرز عن طريق زيادة مقدار مساحة الذاكرة. الفكرة الأساسية وراء استخدام المكدس هو القضاء على التكرار الذي لا داعي له في البحث عن العنصر الكبير والصغير. هذه الخوارزمية تسمى ترتيب الاختيار المتغيرة (DSS) ولدمج مزايا (DSS) مع مزايا "MMBPSS"، اقترحنا خوارزمية جديدة ثالثة تسمى أصغر–أكبر ثنائي الاتجاه المتوازية المتغيرة للترتيب بواسطة الاختيار "MMBPDSS". والتي تمكن من وضع عنصرين من عناصر الحد الأدنى والحد الأقصى من اتجاهين باستخدام خوارزمية الاختيار المتغيره في كل جولة بالتوازي، وبالتالي تقليل عدد الجولات المطلوبة للترتيب. وتم تقديم النتائج التي تم الحصول عليها بعد التنفيذ على شكل رسوم بيانية مع الهدف لاظهار "MMBPDSS" هي أفضل بـ50٪ من خوارزمية ترتيب الاختيار العادية.