Parallel Computing for Sorting Algorithms

*Zainab T. Baqer**

Received 20, December, 2012
Accepted 11, March, 2014

Abstract:

The expanding use of multi-processor supercomputers has made a significant impact on the speed and size of many problems. The adaptation of standard Message Passing Interface protocol (MPI) has enabled programmers to write portable and efficient codes across a wide variety of parallel architectures. Sorting is one of the most common operations performed by a computer. Because sorted data are easier to manipulate than randomly ordered data, many algorithms require sorted data. Sorting is of additional importance to parallel computing because of its close relation to the task of routing data among processes, which is an essential part of many parallel algorithms.
In this paper, sequential sorting algorithms, the parallel implementation of many sorting methods in a variety of ways using MPICH.NT.1.2.3 library under C++ programming language and comparisons between the parallel and sequential implementations are presented. Then, these methods are used in the image processing field. It have been built a median filter based on these submitted algorithms. As the parallel platform is unavailable, the time is computed in terms of a number of computations steps and communications steps.

Key words: parallel, sorting and median filter.

1. Introduction

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors (fig.1) [1], networks of workstations, multiple-processor workstations, and embedded systems. The performance of microprocessors, memories and networks has been improved over 25 to 40 years [2]. Parallel computing has been considered to be "the high end of computing", and has been used to model difficult problems

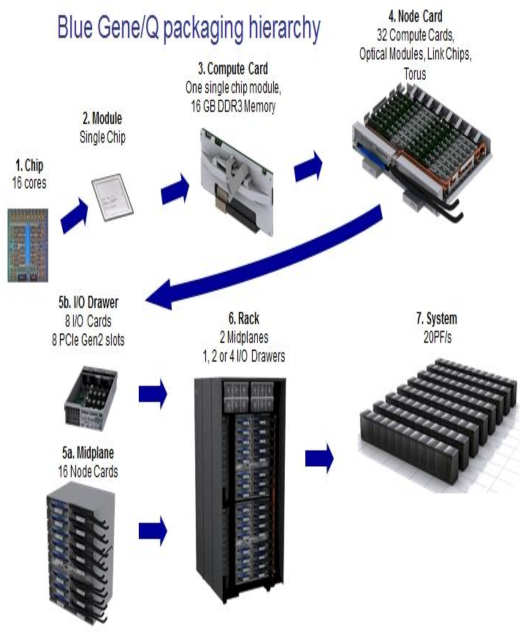


Fig.1 One example of Parallel System, IBM Blue Gene / Q Super Computer [1]

*Electrical Engineering Dept. /Baghdad University, IEEE member

in many areas of science and engineering such as: Atmosphere, Earth, Environment, Physics-applied, nuclear, particle, condensed matter, Bioscience, Biotechnology, Genetics, Chemistry, Molecular Sciences, Geology, Seismology, Mechanical Engineering-from prosthetics to spacecraft, Electrical Engineering, Circuit Design, Microelectronics, Computer Science, Mathematics, Image Processing and so on. In this paper, the parallelism is used in sorting algorithms and as an application in median filter.

2. Related Work

There exist large bodies of research on parallelizing the sorting algorithms such as [3… 7]. In [3] the dual core Window-based platform was used to study the effect of parallel processes number and also the number of cores on the performance of some sorting algorithms. The authors [4] presented a 2D median filter. It had been implemented in three parallel programming models. The authors [5] used field-programmable gate arrays in sorting networks. In [6] the histogram sort, sample sort and radix sort were implemented using two modern supercomputers. In [7] a novel merge-based external sorting algorithm for one or more CUDA- enabled GPUs had been presented.

3. Bitonic Sort

A bitonic sorting network sorts n elements in (log^2 n) time [8]. A bitonic sequence has two tones increasing and decreasing, or vice versa. Any cyclic rotation of such networks is also considered bitonic. <1; 2; 4; 7; 6; 0> is a bitonic sequence, because it first increases and then decreases. To sort any random sequence using bitonic sort, the sequence first is converted to a bitonic sequence. The functions sort_up and sort_down sort the sequences into an increasing and decreasing order respectively using any type of sorting as shown in algorithm1.

Algorithm 1: Bitonic Sort

```
Bitonic Sort  // Sort the sequence A

1. begin
2.  i=0
      // first A is converted to length of $2^i$
3.  no. of element= length (A)
4.  while (no. of element > $2^i$)
5.     i + +
6. for(x=0;x<$2^i$ --no. of elemen; x + +)
7.    A[no. of element + x] = 0
8.  y = 0
9.  x = length (A) / 4
10.  while ( x $\geq$ 1 )
       {
11.    for ( i = x;  i < 0 ;  i - -)
       {
12.        sort_up ( A, index, index + $\frac{4*2^y}{2} - 1$)
13.    sort_down( A,index+$\frac{4*2^y}{2}$, $index + 4*2^y$ -1)
14.    $index = index + 4*2^y$
       }
15.   y + +
16.   x = x / 2
      }
17. x = 1
18. n = length ( A )
19. while ( n / 2 $\geq$ 1 )
    {
20.   index = 0
21.   for ( i = 0, i < x , i + +)
22.     for ( j = 0, j < x , j + +)
23.      if  A[ index + j] > A [index + n / 2 + j ]
           {
24.          temp = A[ index + j]
25.         A[ index + j] = A [index + n / 2 + j ]
26.         A [index + n / 2 + j ] = temp
           }
27.   index = index + n
28.   x = x * 2
29.   n = n / 2
    }
30. end Bitonic Sort
```

4. Mapping Bitonic Sort to a Hypercube

In the implementation of the parallel program a number of processes (process is a set of executable instructions (program)) are created. More than one process can be executed on a single processor. An important feature for the MPI is ***the possibility of using MPI on virtually any computer, even a serial one***. Message passing systems generally associate only one process per processor. The basis of the MPI parallel model is that each processor has its own private memory and private arrays. This is true for both shared and distributed memory architectures. It is possible to test the parallel algorithms which are presented in this work on a single processor using MPI. It is possible to execute these programs using different number of processors. Algorithm 2 shows the implementation of bitonic sort using hypercube interconnection system. Figure (2) illustrates the communication during the last stage of the bitonic sort algorithm. More information on bitonic sort can be found in [9].

Algorithm 2: Parallel formulation of bitonic sort on a hypercube with $n = 2d$ processes. In this algorithm, *label* is the process's label and d is the dimension of the hypercube.

Parallel Bitonic_ Sort(sequence)

```
1.PARALLEL BITONIC_SORT(label, d)
  // sort a sequence on process with id = label in a d-dimensional hypercube
2. begin
3.Get the information about the Communicator
4.Compute the number of processes and determine the process label
5.Set up the Topology
6. Get process label in the new topology
7.Get the coordinates
8. Save row, column,…. coordinate
9. for i = 0 ; i < d ; i + +
10.    for j = i ; j > 0;  j - -
11.  if (i + 1)st bit of label ≠ jth bit of label then
12.        comp_exchange max(j);
13.     else    comp_exchange min(j);
15. end Parallel Bitonic_Sort
```

During each step of the algorithm, every process performs a compare-exchange operation. The algorithm performs a total steps of:

$$(1+\log n)(\log n)/2 \qquad (1)$$

thus,
the parallel run time is: $T_p = \theta(\log^2 n)$ (2)

This parallel formulation of bitonic sort is cost optimal with respect to the sequential implementation of bitonic sort (that is, the process-time product is

$$\theta(n\log^2 n) \qquad (3)$$

but it is not cost-optimal [9] with respect to an optimal comparison-based sorting algorithm, which has a serial time complexity of

$$\theta(n\log n). \qquad (4)$$

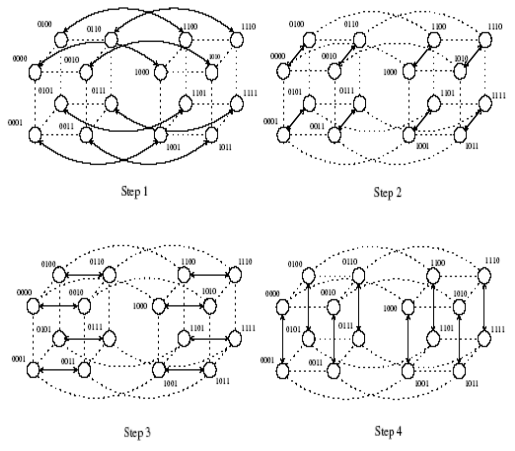


Fig.2 Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube process; each connection represents a compare-exchange between processes.

Mesh

There are several ways for mapping the sequence onto the mesh processes (Fig.3):

(a) row-major mapping,
(b) row-major snakelike mapping, and
(c) row-major shuffled mapping.

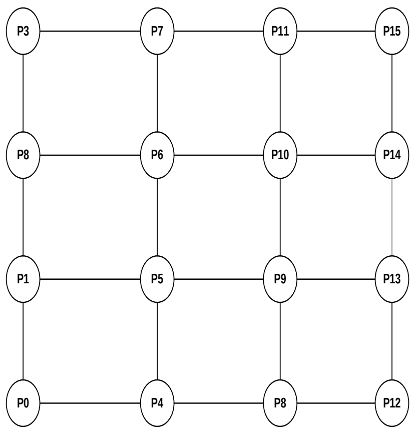


Fig.3 2-D Mesh Processes (16 nodes).

For row-major shuffled mapping, the parallel run time is:

$$T_P = \theta(\log^2 n) + \theta(\sqrt{n}) \qquad (5)$$

More information on this subject can be found in [8]. This is not a cost-optimal formulation, because the process-time product is: $\theta(n^{1.5})$ but the sequential complexity of sorting is $\theta\,(n \log n)$ (4)

5. A Block of Elements per Process

In the parallel formulations of the bitonic sort algorithm presented so far, it was assumed that there were as many processes as elements to be sorted. In this part it is considered that the case in which the number of elements to be sorted is greater than the number of processes.

Let p be the number of processes and n be the number of elements to be sorted, such that $p < n$. Each process is assigned a block of n/p elements and cooperates with the other processes to sort them. One way to obtain a parallel formulation is to think of each process as consisting of n/p smaller processes. In other words, imagine emulating n/p processes by using a single process. The run time of this formulation will be greater by a factor of n/p because each process is doing the work of n/p processes. This virtual process approach leads to a poor parallel implementation of bitonic sort for the reason that for the case of a hypercube with p processes. Its run time will be

$$\theta((nlog^2n)/p) \qquad (6)$$

which is not cost-optimal because the process-time product is explained in equation 3 as $\theta(nlog^2n)$. An alternate way of dealing with blocks of elements is to use the compare-split operation. (n/p)-element blocks are being sorted using compare-split operations. The problem of sorting the p blocks is identical to that of performing a bitonic sort on the p blocks using compare-split operations instead of compare-exchange operations. Since the total number of blocks is p, the bitonic sort algorithm has a total of

$(1 + \log p)(\log p) / 2$ steps (7)

Because compare-split operations preserve the initial sorted order of the elements in each block, at the end of these steps the *n* elements will be sorted. The main difference between this formulation and the one that uses virtual processes is that the *n*/*p* elements assigned to each process are initially sorted locally, using a fast sequential sorting algorithm. This initial local sort makes the new formulation more efficient and cost-optimal.

5.1 Hypercube

The block-based algorithm for a hypercube with *p* processes is similar to the one-element-per-process case, but now there are *p* blocks of size *n*/*p*, instead of *p* elements.

The parallel run time of this formulation is:

$$T_p = \overbrace{\theta(\frac{n}{p}\log\frac{n}{p})}^{local\ sort} + \overbrace{\theta(\frac{n}{p}\log^2 \mathrm{p})}^{comparisions} + \overbrace{\theta(\frac{n}{p}\log^2 \mathrm{p})}^{ccommunication} \qquad (8)$$

Because the sequential complexity of the best sorting algorithm is θ (*n* log *n*), the speedup and efficiency are as follows:

$$S = \frac{\theta(n \log n)}{\theta\left(\left(\frac{n}{p}\right)\log\left(\frac{n}{p}\right)\right)+\theta\left(\left(\frac{n}{p}\right)\log^2 p\right)} \qquad (9)$$

$$E = \frac{1}{1-\theta((\log p)/(\log n))+\theta((\log^2 p)/(\log n))}$$
(10)

5.2 Mesh The block-based mesh formulation is also similar to the one-element-per-process case. The parallel run time of this formulation is:

$$T_p = \overbrace{\theta(\frac{n}{p}\log\frac{n}{p})}^{local\ sort} + \overbrace{\theta(\frac{n}{p}\log^2 \mathrm{p})}^{comparisions} + \overbrace{\theta(\frac{n}{\sqrt{p}})}^{ccommunication} \qquad (11)$$

The efficiency and speedup as follows:

$$S = \frac{\theta(n \log n)}{\theta\left(\left(\frac{n}{p}\right)\log\left(\frac{n}{p}\right)\right)+\theta\left(\left(\frac{n}{p}\right)\log^2 p\right)+\theta(n/\sqrt{p})} \qquad (12)$$

$$E = \frac{1}{1-\theta((\log^2 p)/(\log n))+\theta((\log^2 p)/(\log n))+\theta(\sqrt{p}/\log n)}$$
(13)

(13)

By comparing the communication overhead of this mesh-based parallel bitonic sort to the communication overhead of the hypercube-based formulation, it can be seen that it is higher by a factor of

$$\theta(\sqrt{p}\,/\log^2 p) \qquad (14)$$

From the analysis for hypercube and mesh, it can be seen that the parallel formulations of bitonic sort are neither very efficient nor very scalable. This is because the sequential algorithm is suboptimal. Good speedups are possible on a large number of processes only if the number of elements to be sorted is very large.

6. Quicksort

Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead and optimal average complexity.

Algorithm 3 has an average complexity of: θ (*n log n*). (15)

Algorithm 3: Quick Sort

```
QuickSort (A,q,r)

1. begin
2.  if ( q > = r ) end
3.  Partition ( A, q, r +1, pivot)
4.  QuickSort (A, q, pivot -1)
5.  QuickSort ( A, pivot + 1, r )
6. end QuickSort
```

Algorithm 4 describes the partition algorithm. The operation of quicksort is illustrated in **Fig.4**. The complexity of partitioning a sequence of size *k* is

$$\theta(k). \qquad (16)$$

More information on quick sort can be shown in [9].

Algorithm 4: Partition

```
Partition  ( A)

1. Partition ( A, left, right, pivot)
2. begin
3.   pivot = A[ left ]
4.   LeftToRight = left +1
5.  RightToLeft = right -1
6.  notCrossed = true
7. while ( notCrossed )  {
8.     while ( A [LeftToRight ] < pivot )
9.       LeftToRight + +
10.         while ( A [ RightToLeft ] > pivot )
11.           RightToLeft - -
12.              If ( LeftToRight < = RightToLeft ) {
13.        temp = A[ RightToLeft ]
14.            A [ RightToLeft ] = A [ LeftToRight ]
15.        A [ LeftToRight ] = temp
16.        LeftToRight + +
17.        RightToLeft - -
           }
18.    Else notCrossed = false
       }
19.   A [ left ] = A [ RightToLeft ]
20.  end  Partition Algorithm
```

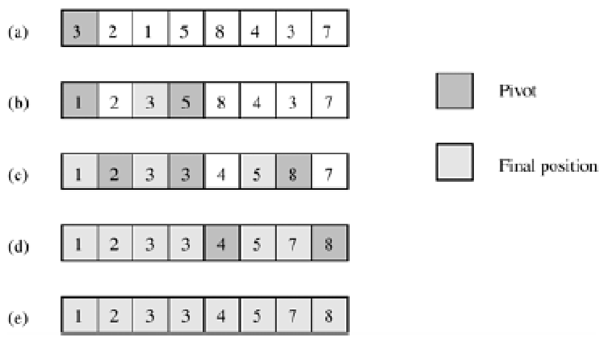


Fig.4 Example of the quicksort algorithm sorting a sequence of size *n* = 8.

7. Parallelizing Quicksort

There are different techniques to parallelize the quicksort method. The following sections describe several of them.

7.1 Shared Address Space Formulation

The implementation of this algorithm is shown in Algorithm 5. **Fig. 5** and **Fig.6** describe the operation of this algorithm [9].

Algorithm 5 : Parallel QuickSort Algorithm

```
Parallel QuickSort (A)

1. ParallelQuickSort ( A, q, r )
   // Sort the sequence A[q….r] on a number of processes
2.begin
3.   Create a number of processes  P
     // The formulation is a shared address type
4.  Partition the sequence A into blocks of size n/p
5. block Ai assigns to process Pi
6. Master Process select a pivot element
7. Master Process broadcast pivot to all the processes
8. Rearrange  (Ai, Si, Li, pivot )
   // each process arrange its block into two sub blocks  Si  with elements smaller than the pivot and Li with elements greater than pivot
9. Store the Si block at the beginning of
```

```
A and the Li at the end of A
10. Master Process divide the processes into two groups
11. If the process in the 1st group
        ParallelQuickSort (S, left of S, right of S)
12. If the process in the 2nd group
        ParallelQuickSort (L, left of L, right of L)
13. end ParallelQuickSort
```

The overall complexity of the parallel algorithm is:

$$T_p = \overbrace{\theta(\frac{n}{p}\log\frac{n}{p})}^{local\ sort} + \overbrace{\theta(\frac{n}{p}\log\ \ p)}^{array\ splits} + \theta\ (\log^2 p)$$

7.2 QuickSort on a Hypercube
This parallel quicksort algorithm takes advantage of the topology of a *p*-process hypercube connected parallel computer. If *n* be the number of elements to be sorted and $p = 2^d$ be the number of processes in a *d*-dimensional hypercube. Each process is assigned a block of *n*/*p* elements, and the labels of the processes define the global order of the sorted sequence. This formulation is shown in **Algorithm 6**. Median filter is one of the applications that use sorting algorithms in its implementation. The following section describes it briefly. The parallel implementations of this filter are described in the following sections.

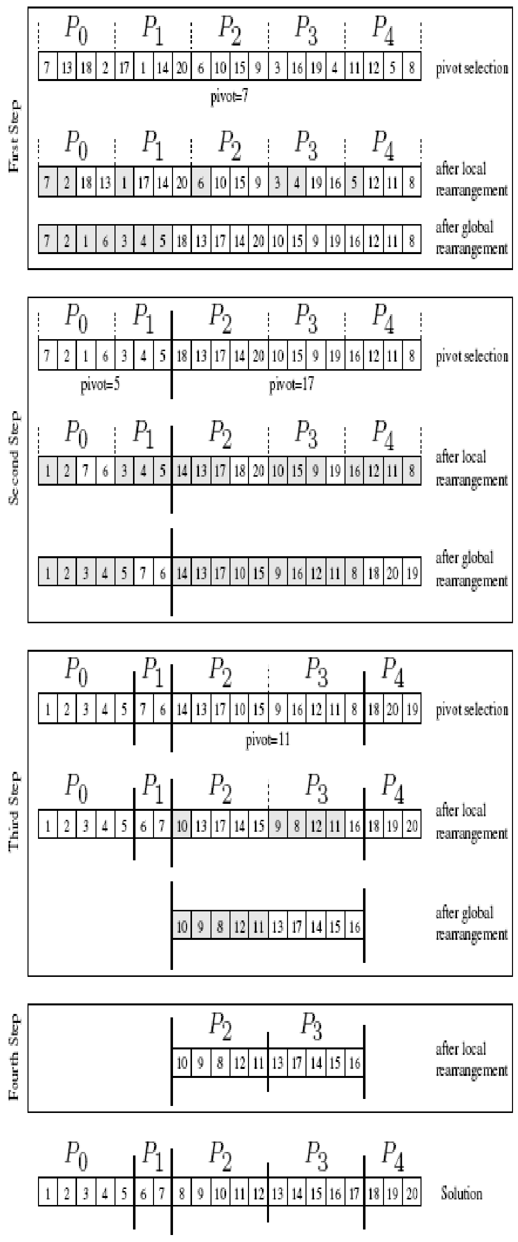


Fig.5 An example of the execution of an efficient shared-address-space quicksort algorithm.

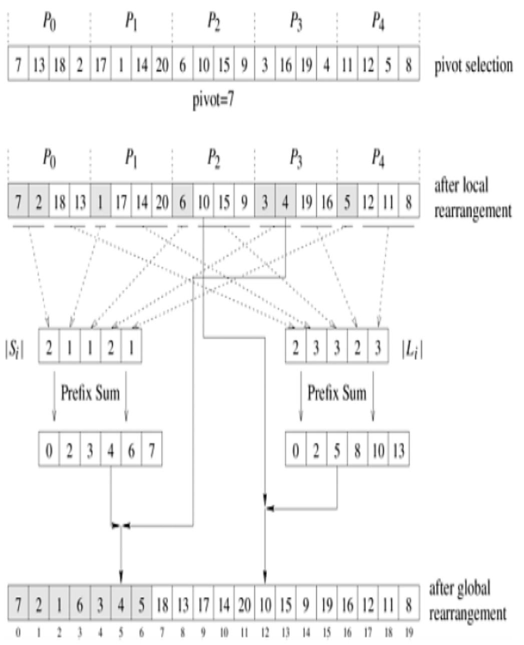

P_0 P_1 P_2 P_3 P_4
pivot selection
pivot=7
after local rearrangement
$|S_i|$
$|L_i|$
Prefix Sum
Prefix Sum
after global rearrangement

Fig.6 Efficient global rearrangement of the array.

Algorithm 6: QuickSort on a Hypercube

```
Parallel QuickSort Hypercube (B,n)

1.ParallelQuickSortHyperCube (B, n)
   // sort sequence B of size n on d dimensional hypercube
2. begin
3. id := process's label;
4. for i = 1 to d do
5.    {
6.       x = pivot
7.       partition B into B1 and B2 such that
           $B1 \leq x < B2$
8.       if ith bit is 0 then {
9.            send B2 to the process along
              the i th communication link
10.           C = subsequence received along
              the i th communication link
11.           $B = B1 \cup C$
                            }
12.    else {
13.            send B1 to the process along
               the ith communication link
14.            C = subsequence received along the
               ith communication link
15.            $B = B2 \cup C$
               }
16.    }
20.        sort B using sequential quicksort // described in Algorithm 5
21. end ParallelQuickSortHyperCube
```

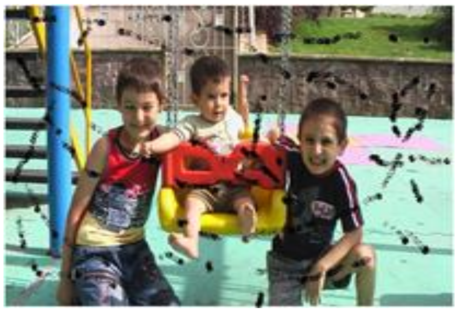
8. Median Filter

In image processing it is usually necessary to perform a high degree of noise reduction in an image before performing higher-level processing steps. The **median filter** is a non-linear digital filtering technique, often used to remove noise from images or other signals. Median filters are particularly effective in the presence of *impulse noise* [10] [11], also called *salt-and-pepper noise* because of its appearance as white and black dots superimposed on an image (**Fig.7**).

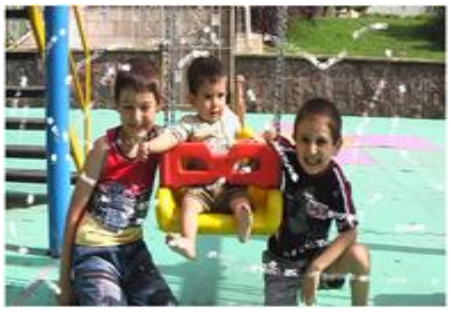
The operation of the filter is shown in Fig.8. The implemented algorithm is shown in Algorithm 16. Any other sorting method like ***quick-sort*** can also be used.

The sorting operation has to be done for each pixel; the median operation is a bit slower than other algorithms. Higher the value of n (or *median_extent*), more values would have to be sorted (n^2) and so slower will be the operation. The median filter algorithms can be implemented in parallel. There are two choices: one can use the parallel implementation for sorting algorithms described previously. The other choice is: since the window of the filter slides on the entire image and in each step the computations is performed independently, this computations can be parallelized using more than one processor (Algorithm 8). In this work a number of processes are created in order to simulate the processors.

(a) original image



(b) image corrupted by pepper noise



(c) image corrupted by salt noise

Fig.7 An image corrupted by salt-and-pepper noise by pepper noise

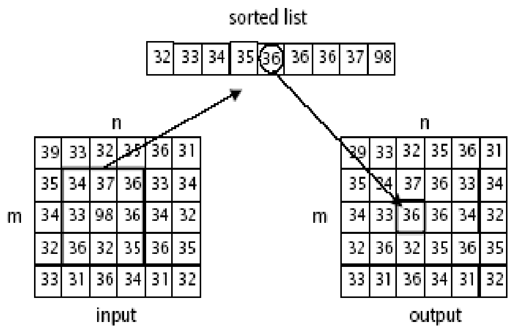


Fig.8 Illustration of the principle of a 3× 3 median filter

Algorithm 7: Median filter algorithm

Median Filter

```
1. MedianFilter( I, median_extent )
2. begin
3. n = median_extent
4. declare a buffer of size n
5.   for ( y = image (min_row) ,
        y < image ( max_row), y + + )
6.       for ( x = image (min_column), x <
           image ( max_ column)
7.          for ( i = 0, i <  n, i + +)
8.             for ( j = 0, j <  n, j + +)
9.                if  x + j – n / 2 ≥ image(min_column)  and
                      x + j – n / 2 ≤ mage(max_column) and
                      y + i – n / 2 ≥ mage(min_row)
                      and  y + i – n / 2 ≤ image(max_row) then
                   buffer ( i × n + j ) =
                   I( x + j –n / 2, y + i – n / 2)
10.               end if
11.            end  j loop
12.         end i loop
13.         QuickSort ( buffer )
14.         O ( x, y ) = buffer ( n / 2 + 1)
15.      end x loop
16. end y loop
17. end MedianFilter
```

Algorithm 8: Parallel Median Filter

Parallel Median Filter Algorithm

```
1. ParallelMedianFilter( I, median_extent )
2. begin
3. Create a number of processes P
// assuming width of the image multiple number of processes
4. n = median_extent
5. declare a buffer of size n
6.  process_id = label ( process)
7. k = process_id
8. for ( y = image ( k ) , y <
        image ( max_row – k ), y + + )
9.      for ( x = image (min_column), x <
        image ( max_ column)
10.          for ( i = 0, i <  n, i + +)
11.               for ( j = 0, j <  n, j + +)
12.                    if  x + j – n / 2 ≥ image(min_column)  and
                         x + j – n / 2 ≤ image(max_column) and
                         y + i – n / 2 ≥ image(min_row) and
                         y + i – n / 2 ≤ □image(max_row) then
                    buffer ( i□ ×□ n + j ) =
                    I( x + j –n / 2, y + i – n / 2)
13.                  end if
14.               end  j loop
15.          end i loop
16.          BubbleSort ( buffer )
17.          O ( x, y ) = buffer ( n / 2 + 1)
18.     end x loop
19.    k = k + P - 1
20. end y loop
21. end MedianFilter
```

9. Conclusion

In this work serial algorithms are presented for bitonic sort and quick sort. The analysis, operation and performance are explained for each type. Then a high performance parallel sorting algorithms are presented and compared with the traditional sort algorithms. The serial algorithm for median filter has been build using quick sort, then the presented sorting methods are applied. The code uses C++ and MPI standard. In the implementation of the parallel program a number of processes are created. The processes can be connected together with different topologies. More than one process can be executed on a single processor. An important feature for the MPI is the possibility of using MPI on virtually any computer, even a serial one. The parallel platform is unavailable so it is impossible to predict the accurate time for the proposed systems. The time is computed in terms of the number of computations steps and communications steps.

References:

1. Megan Gilge 2013. IBM System Blue Gene Solution. Blue Gene/Q Application Development.
2. John L. and David A. 2011. Computer Architecture A Quantative Approach. Fifth edition.
3. Alaa I. El-Nashar. Parallel Performance of MPI Sorting Algorithms on Dual-Core Processor Window Based Systems. International Journal of Distributed and Parallel Systems (IJDPS) Vol.2, No.3, May 2011.
4. Ricardo M. Sánchez and Paul A. Rodríguez. Highly Parallelable Bidimensional Median Filter for Modern Parallel Programming Models. Journal of Signal Processing Systems. June

2013, Volume 71, Issue 3, pp 221-235

5. Rene M., Jens T. Sorting networks on FPGAs. The VLDB Journal. February 2012, Volume 21, Issue 1, pp 1-23.
6. Solomonik, E., L.V. Highly Scalable Sorting. 2010 IEEE International Symposium on Parallel and Distributed Processing. PP 1-12.
7. Peters, H. --- Schulz-Hildebrandt, O. --- Luttenberger, N. Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead. 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum. 2010 Pages: 1-8.
8. Behrooz Parhami Introduction to Parallel Processing, Algorithms and Architectures. Kluwer Academic Publishers 2002.
9. A.Grama, A. Gupta, G. Karypis and V. Kumar Introduction to Parallel Computing, 2003.
10. William K. Pratt. Digital Image Processing: PIKS Inside, Third Edition. Copyright © 2001 John Wiley & Sons, Inc. ISBNs: 0-471-37407-5 (Hardback); 0-471-22132-5 (Electronic)
11. Rafael C. Gonzalez Richard E. Woods Digital Image Processing Second Edition University of Tennessee Prentice Hall Upper Saddle River, New Jersey 07458 2002

الحساب المتوازي لخوارزميات التصنيف

*زينب توفيق باقر**

*كلية الهندسة/ قسم الكهرباء / جامعة بغداد

الخلاصة:

ان التوسع في استخدام الحاسبات العملاقة متعددة المعالجات أحدث نقلة كبيرة في سرعة حل و حجم المسائل. فتبني بروتوكول الواجهة البينية لامرار الرسالة القياسية مكن المبرمجين من كتابة برامج متنقلة و كفؤة خلال تشكيلات توازي متعددة وواسعة .التصنيف احدى العمليات التي تقام بواسطة الحاسبة. لأن البيانات المنسقة أسهل في المعالجة من البيانات العشوائية، الكثير من الخوارزميات تحتاج البيانات المنسقة. التنسيق له أهمية أخرى للحساب المتوازي. في هذا البحث خوارزميات التصنيف التسلسل، البناء المتوازي لكثير من طرق التصنيف وبأستعمال MPICH.NT وبلغة البرمجة C^{++} والمقارنة بين البناء التسلسل والمتوازي قدمت. ثم استخدمت هذه الطرق في مجال معالجة الصور. لقد تم بناء المرشح المتوسط اعتمادا على هذه الخوارزميات المقدمة. ولأن المنصة المتوازية غير متوفرة، تم حساب الوقت من حيث عدد خطوات الحسابات وخطوات الاتصالات.