*Journal of the Nigerian Association of Mathematical Physics*
*Volume 10 (November 2006),* **413 - 422**
***© J of NAMP***

# Heuristic framework for parallel sorting computations

***E. D. Nwanze and E. E. Obasohan***
**Department of Computer Science, University of Benin, Benin City**

**Abstract**

**Parallel sorting techniques have become of practical interest with the advent of new multiprocessor architectures. The decreasing cost of these processors will probably in the future, make the solutions that are derived thereof to be more appealing. Efficient algorithms for sorting scheme that are encountered in a number of operations are considered for multi-user machines. A heuristic framework for exploiting parallelism inherent in some of these schemes are worthy of investigation and valid suggestions are given for adequate implementation by associating processors in a multiprocessor platform. This exercise involves a closer investigation of the associated savings in employing simultaneous sorting techniques for, say $\frac{N}{2}$ processors. A deterministic $o(\log_2 N)$ time algorithm using $N/\log_2 N$ processors will substantially reduce the run time for a sorting scheme and is considered to be asymptotically optimal.**

## 1.0 Introduction

Sorting is one of the most fundamental topics in the core areas of computing dealing with data structures design and analysis of algorithm and computational algebra. Sorting problems often appear as sub problems of many other problems. In this paper we consider important implementable parallel sorting algorithms on a class of processors.

Recent interest in developing alternative algorithms for the sorting problem for parallel computing systems endear one to investigating possible implementable schemes. Our motivation in this study has been to seek feasible parallel methods for sorting which exploit many aspects of parallelism which is implicit in the known algorithms for Von Neumann systems. Methods that maintain balance in sorting systems have received considerable attention in computer science literature. A plethora of materials abound that are devoted to problems and some even to single sort algorithms. A good reason for this attention is that sorting, along with searching and mathematical computations is one of the many things that computers do better than human beings. Another reason is that sorting algorithms are needed in solving many problems and are very sensitive to the issues of size and complexity of the source material; thus, no one algorithm can ever be best for all purposes.

First, the problem of sorting should be considered. Particular attention will be paid to those algorithms that lend themselves to the exploitation of inherent parallelism, primarily to see how limitations in implementing these parallel schemes can inspire other forms of solving the problem.

Most of the existing algorithms work well with older, less sophisticated computers (the Von-Neumann machines) and poorly with modern computers (mostly multiprocessing systems). We hold that this is not in the sense of performance alone, but also in terms of the opportunities offered by parallel machines. Research concern has been on seeking implementable fast algorithms by picking on good strategies for result. The greater discovery is that an existing algorithm can be speeded up greatly if more efficient means of introducing parallel structures to run in high-speed computers are found. Also exposed is the fact that sorting algorithms are good candidates for parallel execution; each individual list could be kept on a separate processor, and many operations could proceed in parallel.

An emerging scheme, we present is hybrid in nature and has the potential to reduce the problem space to less than $O(N\log_2 N)$ time for uniprocessor performance. Optimized Bubble sort, Quick sort, Merge sort, Insertion sort, Heap sort, are good examples of implementable uniprocessor schemes.

Quicksort is highly favoured for the uniprocessor and is regarded as somewhat close to the state of the art sorting method. Years of pious studies and closer examination of sorting schemes have given researchers clues to the solutions of problems of devising implementable schemes for parallel architectures.

The heuristic framework is intended to provide means of presentation. It comprises of formal definition of a sorting system yielding a more formal means of computing parallel systems and an informal method of finding such parallelism in existing sorting schemes. An important part in presenting algorithms is played by a heuristic principle that can be regarded as metatheoretic analogue of the well-known and successful sequential implementations.

## 2.0 Problem Description

Given a list of $n$ data elements, $\{x_0, x_1, \cdots, x_{n-1}\}$, sorting rearranges the order of the elements of the array, and the result of sorting the list is a sorted array, $\{y_0, y_1, \cdots, y_{n-1}\}$ such that $y_i \le y_j$ for every $0 \le i \le j \le n-1$ hence

$$y[0] \le y[1] \cdots y[n-1]. \qquad (2.1)$$

The idea of ordering the list in this form is such that a sorting operation is needed in solving the problem. When the sorting operations are carried out sequentially, the number of pair-wise comparisons between elements of the list becomes too large. A general language description of the problem will look thus: Suppose that the language provides the notion of an indexable sequence of integers, we use subscripts in the range, say, $0..length(X)-1$ so that:

```
Sorted (X) of ∀j
such that $i \le j \le$ length $(X)-1,\ X_j \le X_{j+1}$
Sort (X, Y) if                                              (2.2)
        Image (X,Y) and
Sorted (Y)
```

provided that Y is the sorted image of a sequence X.

A heuristic framework distinguishes sequential behaviour from non-deterministic ones, and thus is more discriminating than ordinary evaluation of sorting schemes or structures. With ranking in the sort algorithms it can be shown that for the count for each element, $x_i$, the number of sorted elements, $length(x_i)$, is smaller than that for $x_j$. Thus the sorted array elements $y_{lenght(x_i)} = x_i$. Consideration is only for arrays of unique elements. If we modify the algorithms, then non-unique arrays elements can be accounted for. It is, however, pertinent for us to quickly state that $length(x_i)$ is unique over all $0 \le i < n$ when all elements are unique.

We can briefly explain by stating a prove thus: for any two elements $x_i < x_j$ such that $x_i = x_i + \varepsilon, \varepsilon > 0$, with counts $length(x_i)$ and $length(x_j)$, it follows that $length(x_j) \ge length(x_i) + 1$ since $x_i$ is less than $x_j$. When $\varepsilon = 0$ then all elements less than $x_i$ are also less then $x_j$. This is clearly shown in the following program segment:

```
for $(i = 0; i \prec n; i++)$
   $\{x = 0;$
   for $(j = 0; j \prec n; j++)$                            (2.3)
          if $(x[i] \succ x[j])k++;$
    $y[k] = x[i];\}$
```

Building on the experience of Hoare [14,15], Knuth [18], Batcher [3], Bitton, et al [5], Tseng et al [34], Zen-Cheung et al [35] and others is a full scale framework for the analysis of associated complexity of computations that provide support not only for this algorithm but also for other parallel implementations of computing algorithms.

There are many sorting algorithms of interest for sequential machines as we noted earlier. An implementation for any style of presentation of any of these algorithms in parallel is a daunting task. At the level of parallel computations, support must be provided for the various stages of our algorithm like fast comparisons, interchanges and transitions to neighbourhood processors through merging. At different levels particular operations are expected to take place. This includes the means of broadcasting ready data for computation to the processors. In most parallel implementations the notion of computations is that the operations are data driven and as such are ready for computation as soon as they are available. Pairwise comparison succeeds this exercise at different nodes in the processing graph (i.e. at the processor). Further effort is needed to support automated comparison and interchange where necessary. In-place algorithms provide for this. Buffering as well as pointer facilities help to overcome the problem of automation. It is therefore highly desirable to develop a general theory of parallel-sorting computational systems that isolate the good attributes of a wide range of sorting schemes so that much of the effort can be expended once and for all on those aspects that will speedup computation.

For a given algorithm and problem size, we derive the inherent parallelism as the ratio of the serial execution time and the runtime of an ideal realization of a parallel random access machine (PRAM). An idealized model of a parallel computer that satisfies all non conflicting memory access in one cycle and queues conflicting memory accesses to be satisfied one after another, each requiring exclusive read, exclusive write (EREW) PRAM. For the same algorithm and problem size, we then derive the maximum speedup attainable by a machine of any size with the architecture of interest. Machine size is in terms of p, the number of processors. The maximum speedup, specified as a function of problem size, is called the asymptotic speedup of the architecture for the given algorithm. Thus the inherent parallelism of the algorithm is its asymptotic speedup on an ideal PRAM machine [25]. The criteria on which the measurement of algorithm for parallel computations is based are the speedup and efficiency. They are used to let one know how effective a parallel machine is being used given an efficient algorithm.

The ability to solve instances of a problem within available resources portends feasibility. Generally, if the consumption of some resource grows exponentially to the problem size, then we can solve only small problem instances. Thus, feasibility is provided when the growth rate for the resource is bounded by a polynomial of the problem size. This means that in time $o(n) = o(1)$ or space $o(n) = o(1)$ for a problem of size $n$. In parallel computations, a parallel algorithm is feasible if solutions to size $n$ problems are found in $o(n) = o(1)$ time using $o(n) = o(1)$ processors. Parallel computing deals with the problem of trading processors for speed. But hardware is a qualitatively different resource than time, so the notion of feasibility needs to be refined. In some cases it is cheaper to give more time for a problem to be solved than to invest in more processors to solve the problem in a shorter period of time. In some cases the cost of not having the solution in a given period of time is greater than the cost of additional processors. Weather forecasting, market prediction and even drug design are examples of time limited problems. If the solutions take too long to compute, then there is little or no benefit. You agree to the fact that adding new processors or memory to a system is not as easy as adding time. When a problem exhausts available memory, we are inclined to find solutions to the problem that use less memory (rather than just buying more memory, which may not be possible in any case). Similarly, considering processor requirements in terms of problem size allows us to examine the question of how big a problem can be solved in a given time with a given number of processors. Therefore, the goal of developing feasible parallel algorithms is expressed in terms of the speedup and the equation is as follows:

$$\frac{best\ \ sequential\ \ time}{number\ \ of\ \ processors} \leq parallel\ \ time \tag{2.4}$$

To obtain a sub-polynomial time algorithm, a polynomial number of processors must be used. For example for a sequential running time $T(n) = n$ and processors $p(n) = \log n$ then:

$$\frac{T(n)}{p(n)} = \frac{n}{\log n} = \frac{n^{0.5}n^{0.5}}{\log n} > n^{0.5} = n^{o(1)} \tag{2.5}$$

is still a polynomial time.

Thus, when we refer to a system as highly parallel, we mean that it requires a number of processors roughly equal to the best sequential time for the algorithm. Therefore, speedup is generally a measure of the same program on varying number of processors. The speedup is then the elapsed time needed by the processor divided by the time needed in p processors, such that (2.4) becomes

$$S = \frac{T(1)}{T(p)} \tag{2.6}$$

The issue of efficiency in related to that of performance, it is usually defined as

$$e = \frac{T(1)}{pT(p)} = S/p \tag{2.7}$$

Efficiency close to unity means that you are using your hardware effectively and low efficiency means that the algorithm is wasting resources [17], [13].

Gaustafson, J. et al [13] argue that users will increase their problem size to keep the elapsed time of a parallel run constant. As the problem size grows, the fraction of the time spent executing serial code decreases, leading us to predict a decrease in the measured serial time fraction.

## 2.1 Algorithm

We present an algorithm to perform the parallel sorting operation. This algorithm is based on the merge sort operation, so we proceed as follows:

Given a list of $N$ elements, the result of sorting the list is to arrange the elements of the list, say $X$ so that

$$X(0) \leq X(1) \leq \cdots \leq X(N-1) \qquad (2.8)$$

We then produce a set of elements such that the output unit becomes active according to some natural methods of processing input in parallel. The job of ordering the list is such that a sorting scheme is applied. When this is carried out sequentially, the lower bound complexity of $\mathrm{O}(N \log_2 N)$ is used. The algorithm is as follows:

*Step1.* *Present an input vector, denoted as ($X_1$, $X_2$ ,..., $X_n$).*
*Step2.* *Assign binary units of data terms to $^N/_2$ processors.*
*Step3.* *Merge-Sort each group of items on the individual processors into half the number of processors using your favourite sequential sort scheme.*(2.9)
*Step4.* *Repeat step 3 until all elements are listed in one single active output unit, such that* $X(0) \leq X(1) \leq \cdots \leq X(N-1)$

## 2.2 Structure Representation

The basis for our representation is the parallel machine, which consists of an interconnection of sequential processes. Intuitively, parallel computing is concerned with how much faster the parallel machine can be over the sequential one. Our reasons for using parallel computing are to: reduce the running time of large problem space, reduce the cost of achieving performance, increase reliability. So if we connect multiple processors, it can be cheaper than building a high-performance single processor; again, a processor fails, then the remaining processors should be able to continue and share the workload.

Machine specific implementations will enable us compare results along these benchmarks. A primitive and atomic process can be used to share single array that resides in a processor into sized blocks and distributed to $p$ processors, which may be based on $n/p$ unit elements. Then another primitive can be used to gather these data blocks and coalesced into larger blocks at each stage of the computation.

However, for the algorithm of (2.8), the main strategy is to use the divide and conquer heuristics to organise program and data in ways as to improve their order of processing in parallel. Generally program tasks are organised in order in which they are to execute and the best structure is achieved using this strategy. Data is also organised by decomposition and are routed into the p processors in such a way as to encourage parallel computation. Geometric decomposition as suggested by [11] plays a prominent role here as the problem space is decomposed into discrete subspaces. Solution is now given by computing those of the subspaces, with the solutions of each subspace typically requiring data from small number of subspaces. The entire sorting scheme is then defined in terms of tracking links through a kind of recursive data structure. The supporting substructures are easily provided as part of the framework of reusable components which capture the recurring solutions to the sorting problems into parallel solutions. A single program, multiple data (SPMD) machine for example, achieves parallel execution by executing the same program code with each operating on a different set of data.

The ideal distributed computer using distributed arrays represent an exotic class of data structures that are prevalent in scientific computing, namely arrays of vectors and multi-dimensions that have been decomposed into sub-arrays and distributed among processes or threads. Examples are the Cray, IBM and the Thinking Machine.

In, this vein, our sorting scheme, which has large problem space, can be heuristically represented by a tree structure. The algorithm is based on this tree, but does not represent all the subtleties associated with the parallel implementation. For some problem instances, upper/lower algorithms are developed that use a *divide and conquer* approach to recursively divide the problem into smaller sub-problems. The results of the sub-problems are then combined to give the final output. Divide and conquer is a fundamental approach for obtaining parallelism that is very intuitive and hence subject to the policy of problem design architecture. Parallel processes that occur as sub graphs to be parallel-sorted during the sorting exercise are represented by $^N/_2$-ary trees. The root of the tree is the final active output unit and the children are the sub graphs that are supposed to be merged to produce $^N/_2$ out put units. A sample graph, together with its assembly is given below in Figure 1. Therefore, our sorting benchmark is defined such that n and p=m are assumed for simplicity but without loss of generality to be powers of two so that we set $n/p$ elements at each of the first $n/2$ processors to be $\log_2 n$, all $2n/p$ at the next $n/4$ processors to be $\log\left(\frac{n}{2}\right)$, and so forth. The presentation model is as given in figure 1.

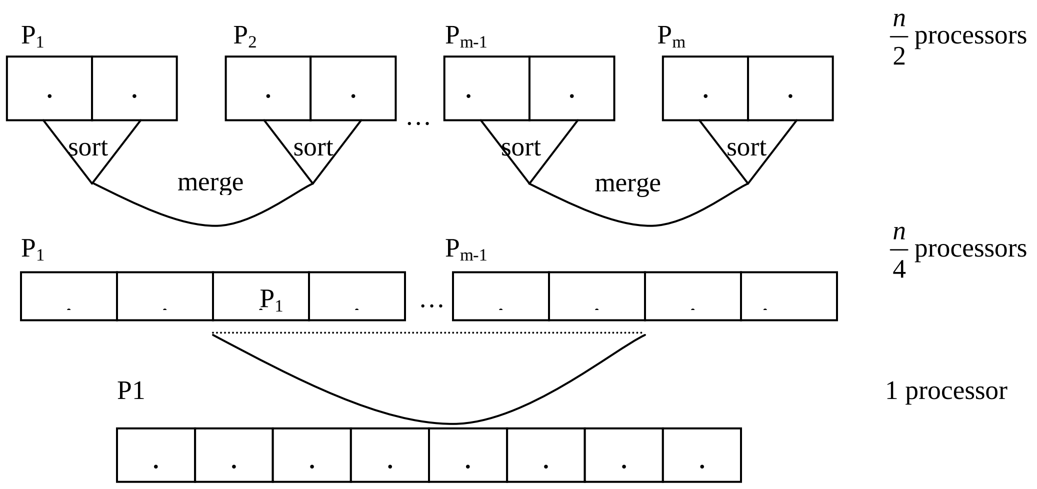

**Figure 1: Presentation Model**

## 3.0 Overview of Algorithm

***Scalability*** of a given architecture is said to be the fraction of the parallelism inherent in a given algorithm that can be exploited by any machine of that architecture as a function of problem size. It is in this regard that conceivable attempts are made to design algorithms to exploit the inherent parallelism in such systems.

In a sequential machine, the number of pairwise comparisons between elements of the initial lists is equal to the sum of the number of elements in all the lists and that is $O(N)$. To speedup the merging operation, the compare operation is intuitively done using parallel processors, each capable of comparing two input elements, and identifying the smaller of the two. This goes to show that $^N/_2$ neighbourhood processors may be used for lists each of length 2, to compare respectively the first entries from each of the $^N/_2$ lists and this is done at a unit time in parallel. By suitably employing this method at this stage the time complexity is greatly reduced. Good message passing techniques will feed the output of this first stage to the second stage of the algorithm that involves merging.

Konrad [19], Horowitz [16], Mannila and Ukkonen [8], and Pardo [9] have all tried to solve the problem of merging using the in-place merging methods. Konrand's notion on block rearrangements and internal buffering has influenced the other works. Implementations have always been the problem since the algorithmic schemes are complicated. Bing-chao Huang et al [4] have their simple algorithm which merges in linear time and constant extra space. The list to be sorted is blocks of $O(\sqrt{n})$, each of size $O(\sqrt{n})$. This method involves rearranging blocks before the merging is initiated and the internal buffer is passed across the list so as to minimize unnecessary record movement. One interesting aspect of this scheme is that the overall elapsed time is reduced for file processing whenever the extra space can be utilized for more buffers to increase parallelism or larger buffers to reduce the number of input/output transfers needed.

For algorithm (2.8) to work in ideal situation, we need splitters to partition the input data into p groups indexed from 1 up to p such that every element in i$^{th}$ group is less than or equal to each of the items in the neighbourhood (i.e. $i$+1) $^{th}$ group, for $1 \le i \le p-1$. So to perform the sorting task, each of the p groups can be turned over to the corresponding indexed processor, and then the n data items will be arranged in sorted order.

The efficiency of such an algorithm depends on the processing elements that assign the data units to the processor, the sorting method chosen and the message passing scheme adopted for the data transfer with implicit merging mechanism. A radix sort algorithm and the merge sort algorithm can combine elegantly to solve the problem. The general strategy is to decompose tasks and data into manageable sub groups and route them via the p processors to be computed in parallel. Our idea suggests the strategy that can combine them as a composite scheme to achieve fast and tractable computation. This is basically true when you consider the dependencies that could be identified through analysis of the problem at hand. Many recent multiprocessor machines with ideal status can be used to implement the algorithm, but requires that experiments be done using a variety of benchmarks.

For the uniprocessor architecture, the worst-case time complexity of mergesort and the average-case time complexity of the quicksort are both of $O(n\log_2 n)$. However, with the p processors, the best that we could expect is $O(\log_2 n)$.

## 4.0 Proof

We assume that the table of n elements of $X$ denoted as $X(0), X(1), \cdots, X(n-1)$, on which a linear order has been defined is given. It therefore holds that for any two elements $X_i$ and $X_j$, exactly one of the following cases must be true $X_i < X_j, X_i = X_j \text{ or } X_i > X_j$. Sorting has the goal of finding a permutation $(\pi_0, \pi_1, \cdots, \pi_{n-1})$ such that: $X_{\pi_0} \le X_{\pi_1} \le \cdots X_{\pi_{n-1}}$

(4.1)

The best complexity order for any sequential algorithm, which is based on pairwise comparison of elements of size n, must be $\Omega(N \log N)$. (4.2)

For a quick proof of the basis of our for the above we recall that the asymptotic time complexity for the uniprocessor scheme as $O(N \log_2 N)$ (4.3)

while for the parallel algorithm, an estimated time is $O(\log_2 N)$ with a speed up of

$$O(N) \tag{4.4}$$

From (2.7), the efficiency of such an algorithm can notionally be put at $O\left(\frac{N}{p}\right)$ (4.5)

which gives $O\left(N \big/ \frac{N}{\log_2 N}\right)$ where p is $N/_{\log_2} N$ (4.6)

We recall that ***quicksort*** parallelizes over $n$ processors to obtain $O(N)$ parallel computational steps. It generally selects a pivot for the elements in the array, thereby ensuring that all elements in the array that are less than the pivot are put into a lower array and all elements greater than the pivot are put into a higher array. Using this divide and conquer scheme, it then recursively applies the scheme on the higher and lower arrays. Selection of the pivot is not too important for the sequential algorithm however it is important for the parallel algorithm in order to keep the tree of processes reasonably balanced.

***Mergesort***, again proceeds from a single processor or process that holds an array of $n = 2^k$ elements. The divide-and-conquer approach is used to divide the array into two halves and give one half to another process. The subdivision continues until at most each of $n$ processes holds exactly one element. Then the processes use mergesort to generate the sorted array. Assuming there are $p = n = 2^k$ processors. The first division phase of the mergesort algorithm is essentially scattering the elements over the processors. Each processor receives one element of the array. The total number of parallel computation steps is $k$, at each step, $i = 0, 1, \cdots, k-1$, two lists of size $2^i$ are merged onto a single processor. It takes $2n-1$ steps in the worst case to merge two sorted lists each of $n$ numbers. The number of computational steps is then

$$2\sum_{i=0}^{k-1}\left(2^i - \tfrac{1}{2}\right) = 2^k - k - 2 \text{ which is } O(N). \tag{4.7}$$

The *bitonic merge* algorithm introduced in 1968 by Batcher [3] has a time complexity of $\Theta(\log^2 N)$ and has formed the basis for sorting algorithms on several models of parallel computations. Fundamentally this method is called *compare-exchange* because two numbers are routed into a comparator, where they are exchanged, if necessary, so that they are in proper order. Batcher was able to prove that a list of $n$ = 2 unsorted elements can be sorted by using a network of $2^{k-2}k(k+1)$ compactors in time $\Theta(\log^2 N)$.

(4.8)

The basis of the bitonic mergesort is the bitonic sequence, a list having specific properties that will be utilized in the sorting algorithm. A monotonic increasing sequence is a sequence of increasing numbers. A bitonic sequence has two sequences, one increasing and one decreasing. Formally, a bitonic sequence is a sequence of numbers, $x_0, x_1, \cdots, x_{n-2}, x_{n-1}$, which monotonically increases in values, reaches a maximum, and then monotonically decreases in value: $x_0 < x_1 < \cdots x_i > x_{i+1} > \cdots > x_{n-2} > x_{n-1}$ for some $0 \le i < n$. A sequence is also bitonic if the preceding can be achieved by shifting the number cyclically (left or right).

A bitonic sequence is transformed into a sorted list by the Bitonic merge algorithm. This list is thought of as half a bitonic sequence of twice the length. If a bitonic sequence of length $2^k$ is sorted with ascending order, while an adjacent sequence of length $2^k$ is sorted into descending order, then after k compare-exchange steps the combined sequence of length $2^{k+1}$ is a bitonic sequence. Therefore a list of n elements to be parallel-sorted can be viewed as a set of an unsorted sequence of length 1 or as $n/2$ bitonic sequence of length 2. Hence the algorithm can be used to sort any sequence of elements by successfully merging larger and larger bitonic sequences. Given $n = 2^k$ unsorted elements with k phases numbered 1,2, …, k, a network with $\frac{k(k+1)}{2}$ levels suffices. Each level contains $\frac{n}{2} = 2^{k-1}$ comparators. Hence the total number of comparators is $2^{k-2}k(k+1)$. The parallel execution of each level requires constant time. We note that $\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log_2 n(\log_2 n+1)}{2} = O(\log_2^2 n)$. (4.9)

Hence the algorithm has complexity $\Theta(\log^2 N)$. The speedup on *n* processors is thus $O\left(\frac{n}{\log^2 n}\right)$ and gives an efficiency of roughly $\frac{1}{\log^2 n}\times 100\%$ (4.10)

The bitonic sequence has an interesting property that if we compare and exchange $x_i$ with $x_{i+\frac{n}{2}}$ for all $0 \le i < \frac{n}{2}$, we get two bitonic sequences, where the numbers in one sequence are all less than the numbers in the other sequence. For example before: *X* = [6,38,45,42,8,20,28,25] and after 6,20,28,25,8,38,45,42.

The second list is now two bitonic sequences, 6,20,28,25 and 8,38,45,42. Using this property, with $n = 2^k$ elements and *n* processors, after *k* parallel steps it is clear that a given bitonic list can be sorted. This is called a bitonic sort operation.

An improvement of the bitonic merge sort was given by Stone [32], which takes a list of $n = 2^k$ unsorted elements and sorted in time $\Theta(\log^2 N)$ with a network of $2^{k-1}(k(k-1)+1)$ comparators using the perfect shuffle interconnection scheme exclusively.

Bitonic merge sort, therefore, seems unsuitable for implementation in VLSI, because of the large number of path crossing. However, the efficiency of the method has made it a popular basis for algorithms on processor array models.

Sorting methods on the SIMD machine models have emerged from the earlier works on the bitonic merge method, but this time the processing elements are organized into arrays. We assume that $X = (X_0, X_1, \cdots, X_{n-1})$ is the set of *n* elements to be sorted and the arrays $Y = (Y_0, Y_1, \cdots, Y_{n-1})$ and $Z = (Z_0, Z_1, \cdots, Z_{n-1})$ contain temporary values. Assume n is even and that all $i$, $0 \le i < n-1$ processes $p_i$ contains array elements $X_i$, $Y_i$ and $Z_i$. The SIMD – MC$^1$ model sort requires $n/2$ iterations with each iteration having two phases. The odd-even exchange as it is called has the value $x_i$ in every odd number processor $i$ (except processor *n*-1) compared with the value $X_{i+1}$ stored in even numbered processor $i+1$. An exchange is made if necessary, so that the lower-numbered of the adjacent processors contain the smaller value. The next phase called the even-odd phase, exchanges values as in the first phase if necessary, so that the lower-numbered processor contains the smaller value. After $n/2$ iterations the values are sorted.

The time complexity of sorting n elements in the SIMD - MC mode with n processors using odd-even transposition is $\Theta(N)$. The proof of correctness of the method is based upon the fact that after iterations of the outer loop, no element can be further than *n-2i* positions away from its final, sorted position. Hence $n/2$ iterations are sufficient to sort the elements, and the time complexity of the parallel algorithm is $\Theta(N)$, given n processing elements.

Another sorting method on the SIMD-MC$^2$ model assumes that n x n elements are to be sorted elements distributed evenly, one element per processing element. The algorithms assume also that simultaneous data routings must be in the same direction (east, west, north or south). In this case a lower bound on any sorting algorithm is $\Omega(n)$. This method is proved to have a lower bound on the number of data routings needed, in the worst case as 4(n-l). This algorithm to sort n$^2$ elements on the SIMD – MC$^2$ model has time complexity $\Omega(n)$, which implies a lower bound on $\Omega(\sqrt{n})$ to sort n elements.

Thompson and Kung [33] in their own algorithm having assumptions identical to the SIMD-MC$^2$ model above were able to establish the fact that to sort $N = n^2 = 2^k$ elements on the SIMD-MC$^2$ model you need a time complexity of $\Omega\left(\sqrt{n}\right)$. This algorithm was demonstrated using the bitonic merge sort on the element on a 4 x 4 mesh by Knuth [17] to prove the point meant by the algorithm. In general, to sort the $N = n^2 = 2^k$ elements by using the algorithm requires $\log N$ phases. The total number of routing steps performed is $\sum_{i=1}^{\log n} \sum_{j=1}^{i} 2\left[\frac{j-1}{2}\right]$ which is $\Omega\left(\sqrt{n}\right)$. The total number of comparison steps is, $\sum_{i=1}^{\log n}$ which is $\Theta(\log^2 N)$. Proving that the worst case time complexity of bitonic merge on the SIMD-MC$^2$ model is

$$\Omega\left(\sqrt{n}\right) \qquad (4.11)$$

making it an optimal algorithm for this model.

Nassimi and Sahni [24] working on the SIMD-MC$^2$ model machine again developed the random access read and random access write which can begin sorting records by destinations which can be completed in $\left(\sqrt{N}\right)$ time on SIMD-MC$^2$ model with $\sqrt{n} \times \sqrt{n}$ processing elements. Sorting on the SIMD-MC$^2$ model sorts n elements when $n = 2^m$ for some positive integer $m$ is $\Theta\left(m^2\right) = \Theta\left(\log^2 n\right)$ time. Sorting on the SIMD-CC module requires $\frac{m(m+1)}{2}$ compare-exchange steps, $m(m-1)$ shuffle steps of the vector *X*, and $2m-1$ shuffles of the vectors *M* and *R*.

From Stone [32] uses a make vector *M* can be used to indicate the kind of sort to be done by a particular processing element. So if we assume $m = \log_n$, then, the time complexity of this algorithm is $O\left(\log_2 n\right)$ with n processors. Hence for the p processors on the ideal parallel machine the sorting method has an asymptotic time complexity of $O\left(\log_2 n\right)$. So that the compare-exchange steps required does not exceed the number of processing elements involved which is $\frac{N}{2}$ logarithmic time complexity of $O\left(\log_2 n\right)$ steps.

A further proof can be given by comparing each number against $n-1$ other numbers requires $n-1$ computation steps. There are $n$ elements so there are $n(n-1)$ computational steps in total. For $n$ processors, each computing the index of an element in parallel, sorting can be accomplished in $O(N)$ computational steps. Each processor needs access to the entire array of numbers and so this is convenient for shared memory architectures. The efficiency is $\frac{\log_2 n}{n} \times 100\ \%$. Consider the use of $n^2$ processors. Each processor $p_{i,j}$ compares $x_i$ with $x_j$. (processors $p_{i,i}$ are not actually required.)

Comparison requires $O(1)$ computational steps. Using a reduction across $i$, processors $p_{i,j}$ can compute the index, $y_i$, of element $i$ in $O\left(\log_2 n\right)$ computational steps. In a final $O(1)$ computational step, the element $x_i$ is written to index $y_i$. The sorting is accomplished in $O\left(\log_2 n\right)$ steps. However the efficiency is $\frac{1}{n} \times 100\ \%$. Using a concurrent read and concurrent write memory architecture with concurrent writes being handled as additions, the reduction operation can be accomplished in $O(1)$ steps. Thus the sorting is accomplished in $O(1)$ steps. The efficiency is now $\frac{\log_2 n}{n} \times 100\ \%$.

On the basis of our framework, it is assumed that the number of elements presented is $n$. And usually the number of elements, $n$, is much greater than the number of processors, $p$. So, in this case, there is the need to partition the elements into clusters of size $k = \frac{n}{p}$. For the ideal sorting this means that each processor must find the index for $k$ different elements and through message passing update the units of the neighbourhood processor using the best strategies.

For example, a good sorting mechanism in an ideal machine for 8 objects used above, will proceed as follows

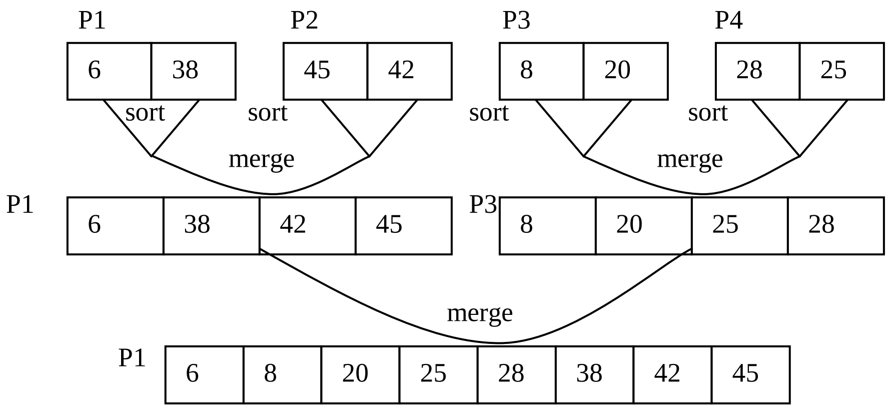

**Figure 2: Sample Parallel Sort**

## 5.0 Conclusion

Considering the efficient implementations of sorting methods as provided by the hypercube we note that the bitonic merge sort provided the basis for most parallel sorting schemes. However, we can definitely say that sorting an unsorted list of numbers requires building bitonic lists and then sorting the bitonic lists is quite an enormous task. It requires $n = 2^k$ elements, and $k$ phases numbered $1,2,\cdots,k$, each requiring a bitonic sorting operation (the first phase is simply sorting single elements) of $k$ steps. But for the ideal machine this exercise is far from optimal as demonstrated by (4.7).

For ideal machine architecture, the $o(\log_2 N)$ time complexity proves adequate, functional and realizable given the nature of the problem space. All the paths of the new parallel machines are no longer those of the exhaustive sequential sorting systems, and therefore at most $\frac{N}{2}$ graph-theoretic graphs path length may be constructed for k clustered objects. Since the compare-exchange merge-sort is the most time consuming part of the algorithm and $o(\log_2 N)$ is the time bound of the entire sort, the existence of a finite scheme for this exotic algorithm enables us to prove the parallel sort theorem rigorously.

The prospect of demonstrating the reality of surmounting the problems is quite promising since the state-of-the-art machines now possess large processing elements to cope with the problem.

## Reference

[1] Ajtai, M., Kolmor, J. and Szermeredi, E. “Sorting in $c\log n$ Parallel Steps.” Combinatronica, Vol.3, 1983, 1-19.

[2] Aki, S. G. Parallel Sorting Algorithms, Academic Press Inc., Toronto, 1985.

[3] Batcher, K. E. “Sorting Networks and their Applications.” Proceedings of AFIPS Springer Joint Conference Vol. 32, 1968, 307-314.

[4] Bing-Chao Huang and Michael A. Langston. Practical In-place Merging.” Comm. of the ACM Vol. 21 No. 3 March 1988, 348-352.

[5] Bitton, D., Dewitt, David J., Hsiao, David K., and Menon, Jai. “A Taxonomy of Parallel Sorting.” Comm. of the ACM Computing Survey, Vol. 16 No. 3, Sept. 1984.

[6] Blum, M. Floyd, R., Pratt, V. Rivest, R., and Tarjan, R. “Time Bounds for Selection.” J. of Computer Systems, Science 7, 1973, 448-461.

[7] Clint, W. and Munro, J. “Average Case Selection.” J. of ACM, Vol.36 No.2, 1989, 270-129.

[8] Cole, R. “Parallel Merge Sort.” Proceeding, 27th IEEE Symposium, FOCS, 1986.

[9] Cormen, T. H., Leiserson, C. E., Rivest, R. G., and Stein, C. Introduction to Algorithms 2. Auflage, The MIT Press, 2001.

[10] Driscoll, James R. “Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation.” Comm. of the ACM Vol. 31 No. 11 1988.

[11] Elkadi, M and Mourrain, B. “A New Algorithm for Geometric Decomposition of Variety.” ISSAC, 1999, 9-16.

[12] Gerasch, T. E. “An Insertion Algorithm for a Minimal Internal Path Length Binary Tree Search.” Journal of the ACM Vol.31 No. 5 1988.

[13] Gustafson, J., Montry, G. and Benner, R. “Development of Parallel Methods for 1024-processor Hypercube.” SIAM Journal of Science and Statistical Computing, Vol.9 No.4, July 1988, 609-638.

[14] Haore, C. A. R. “Quicksort”, Computer Journal Vol.5, No.1, 1963, 10-15.

[15] Haore, C. A. R. “Algorithms 64 PARTITION and Algorithm 65 FIND”. Comm. of the ACM Vol. 4, 1961, 321.

[16] Horowitz, E., Sahni, S. and Rajasekaran, S. Computer Algorithms in C++. Computer Science Press Imprint of W. H. Freeman and Co, New York, 1997

[17] Karp, Alan H. and Flatt, Horace P. “Measuring Parallel Performance.” Comm. Of the ACM, Vol. 33 No.5, 1990, 539-543.

[18] Knuth, D. E. The Art of Computer Programming, Vol. 3. Sorting and Searching. Addison Wesley, Reading, Mass., 1973.

[19] Konrad, Zuse, The Computer – My life, Berlin, 1993.

[20] Leighton, T. “Tight Bound on the Complexity of Parallel Sorting.” IEEE Transaction on Computer, Vol. C-34, April, 1985.

[21] Manacher, G. K, Bui, T. D. and Mai, T. Optimum Combinations of Sorting and Merging. J. of ACM, Vol. 36 No. 2, 1989.

[22] Mannila H., and Ukkonen, E. A Simple Linear-time Algorithm for Insitu Merging, Information Processing letters Vol. 18 1984, 203-208.

[23] Mitra, G. Mamiz, M. and Yadegar, J. Investigation of an Interior Search Method Within a Complex Framework. ACM Vol. 31, No. 12 1988.

[24] Nassimi, Sahni. “Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network.” Journal of the ACM, Vol. 29 No.3, 1982.

[25] Nuussbau, D. and Agarwal, A. “Scalability of Parallel Machines.” Comm. ACM Vol.34 No.3, 1991, 57-61.

[26] Pardo, L. T. Stable sorting and merging with optimal space and time bounds. SIAM J. Computer Vol. 6, 977, 351-372.

[27] Pohl, I. “A Sorting Problem and its Complexity.” Comm. ACM Vol. No. 15, 6 June 1972, 462-464.

[28] Postmus, J. T, Rinnooy Kan, A. H. G and Timmer, G. T. “An Efficient Dynamic Selection Method.” Comm. ACM Vol. 26 No. 11 Nov. 1983, 878-881.

[29] Quinn, M. J. Designing Efficient Algorithms for Parallel Computers. McGraw-Hill Series in Supercomputing and Artificial Intelligence, 1987.

[30] Sedewick, R. Algorithms in Java, parts 1-4, 3 Auflage, Addison-Wesley, 2003.

[31] Shi, H. and Schaeffer, J. “Parallel Sorting by Regular Sampling.” Journal of Parallel and Distributed Computing, Vol.14 no. 4, 1992, 361-372.

[32] Stone, H. C. “Parallel Processing with the Perfect Shuffle.” IEEE Trans. Computer, Vol. 20 No. 2, 1971, 153-161.

[33] Thompson, C. D. and Kung, H. T. “Sorting on a Mesh Connected Parallel Computer.” Comm. ACM, Vol. 20 No. 4, April 1977, 287-318.

[34] Tseng, Chau-Wen. “Data Layout of High-Performance Architectures.” Technical Report CS-TR-3818, Dept. of Computer Science, University of Maryland, Feb. 1997.

[35] Zen-Cheung Shih, Gen-Huey Chen, Richard C. T. Lee. “Systolic Algorithms to Examine All Pairs of Elements.” Comm. ACM Vol.30 No.2, 1987, 161-167.

*Journal of the Nigerian Association of Mathematical Physics Volume 10 (November 2006)*, **413 - 422**