World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:3, No:6, 2009

An Enhanced Distributed System to improve the Time Complexity of Binary Indexed Trees

Ahmed M. Elhabashy, A. Baes Mohamed and Abou El Nasr Mohamad

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

Abstract—Distributed Computing Systems are usually considered the most suitable model for practical solutions of many parallel algorithms. In this paper an enhanced distributed system is presented to improve the time complexity of Binary Indexed Trees (BIT). The proposed system uses multi-uniform processors with identical architectures and a specially designed distributed memory system. The analysis of this system has shown that it has reduced the time complexity of the read query to O(Log(Log(N))), and the update query to constant complexity, while the naive solution has a time complexity of O(Log(N)) for both queries. The system was implemented and simulated using VHDL and Verilog Hardware Description Languages, with xilinx ISE 10.1, as the development environment and ModelSim 6.1c, similarly as the simulation tool. The simulation has shown that the overhead resulting by the wiring and communication between the system fragments could be fairly neglected, which makes it applicable to practically reach the maximum speed up offered by the proposed model.

Keywords—Binary Index Tree (BIT), Least Significant Bit (LSB), Parallel Adder (PA), Very High Speed Integrated Circuits Hardware Description Language (VHDL), Distributed Parallel Computing System (DPCS).

I. INTRODUCTION

The amount of data associated with many computing systems has evolved dramatically compared to the evolution of the computing power. Accordingly, building speedy, efficient and compressed data structures to hold the increasing data size, has been the main interest of many researcher. Many data structures have been introduced in the last few decades to increase the efficiency of traversing and manipulation of large set of data. Binary Indexed Tree (BIT) is one of the efficient data structures which reduced the time cost of some operations in an effective manner. Being locked to find algorithmic solutions to enhance the efficiency of some computing systems, researchers started to think of using parallel computing systems to perform the required enhancement. Many types of parallel computing systems have been used as well, each has some advantages that would fit for specific applications. One of the parallel computing models that was widely used, is the Distributed Parallel Computing System (DPCS). This model is based mainly on partitioning the data into some separated fragments and use multiple processing unit to access these fragments. There could be some sort of communication between the processing units or the memory fragments. The more this kind of communication is avoided, the more speed up is perform.

II. METHODS AND MATERIALS

A. Binary Indexed Tree

Binary Indexed Tree (BIT) is a Data structure presented by Peter M. Fenwick for maintaining the cumulative frequencies which are needed to support dynamic arithmetic data compression. It is based on the decomposition of the cumulative frequencies into portions that could be mapped to the binary representation of the table indices. Traversing this data structure is based on the binary coding of the indices. The access time for all operations is proportional to the logarithm of the table size. BIT is fast, uses more compact data and simpler code ,which make it one of the most practical data structures that are suitable for handling large tables of data.[4]

1) Description and Structure: Binary Indexed Tree is a data structure to which ordinary arrays could be mapped in order to enhance the time complexity of some queries. Lets define the following problem: Assuming an array of size N with the possible queries are:

1) Changing the value at some index i.
2) Summing the values from some index i to some index j.

The classical solution has a constant time complexity for the first query and O(N) for the second one. With BIT both queries could be performed in O(log (N)). The basic concept behind BIT is that, each integer can be represented as sum of powers of two. In the same way, cumulative frequency can be represented as sum of sets of sub frequencies. In our case, each set contains some successive number of non-overlapping frequencies. An array of size 16 (1 to 16) could be mapped to the BIT presented in figure (1), so that tree[IDX] is the sum of frequencies from index (IDX - 2r + 1) to index IDX, where IDX is any index in the tree, r is the position(right to left starting from zero) of the least significant non-zero bit in the binary representation of IDX.[6], [7]

2) Reading Cumulative Frequency: The process of reading cumulative frequency from index 1 to some index IDX could be formulated with these steps (assuming IDX is in its binary notation).

- Initialize a certain variable (SUM) to zero.
- Add tree[IDX] to SUM.
- Invert the least significant non-zero bit in IDX.
- Repeat the last two steps while IDX is greater than zero.

Figure 2 presents an example of looking for the cumulative frequency of the first 13 elements. In binary notation, 13 is equal to 1101. Therefore, the value should be calculated as
c[1101]=tree[1101]+tree[1100]+tree[1000]
Where c[a] is the cumulative frequency from index one to

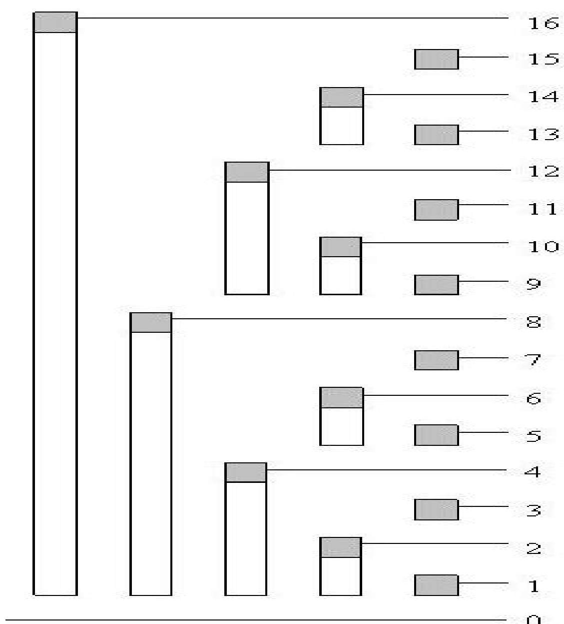

Fig. 1. Binary Indexed Tree (BIT)

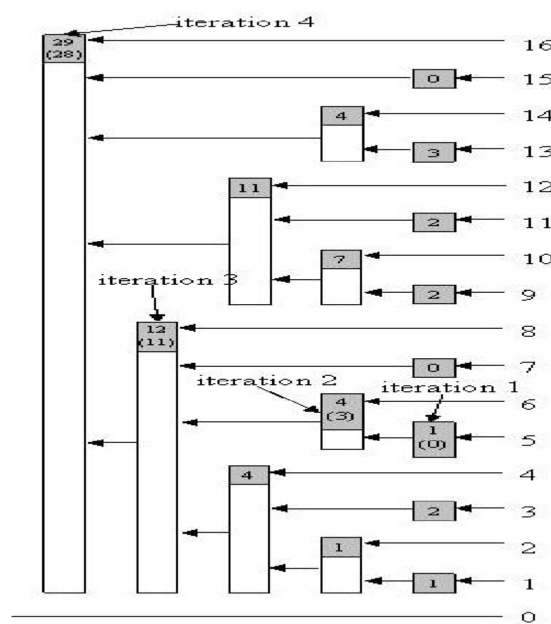

Fig. 3. BIT Update Query

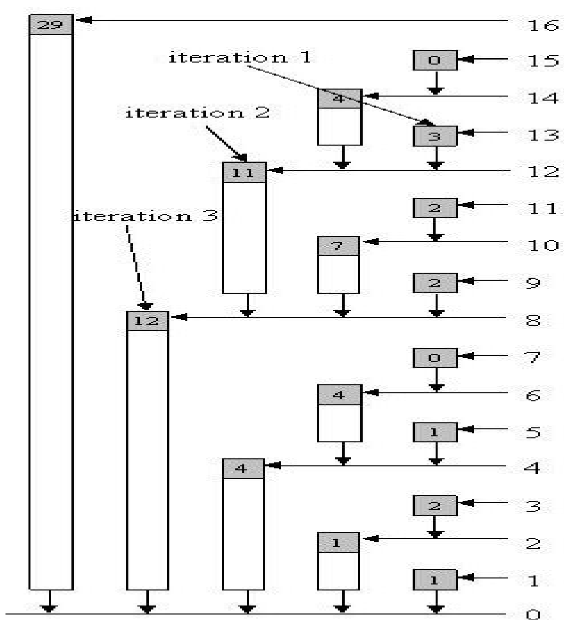

Fig. 2. BIT Read Query

index a, tree[b] is the value of the BIT at index b. Now The process of reading the cumulative frequency from some index i to some Index j should be very obvious, just calculating c[j] - c[i-1].

3) Frequency updating query: The main target of an updating query at some index IDX is to update the tree at all the indexes which are responsible for the value at index IDX. In reading cumulative frequency, the least significant non-zero bit is removed and this process is repeated as long as the zero index is not yet reached. When changing some frequency in the tree, the least significant non-zero bit in the index is added to the index itself and this step is repeated while the index is less than or equal to the maximum index (the tree size). The procedure of changing the frequency at some index IDX by some value VAL consists of the following steps:

- Add VAL to tree[IDX].
- Add the least significant non-zero bit of IDX to IDX.
- Repeat the last two steps while IDX is less than or equal to the tree size.

Figure 3 illustrates an example of updating the frequency at the index 5. In binary notation, 5 is equal to 00101. Hence tree [00101], tree [00110], tree [01000] and tree[10000] should be all updated.

B. Design Model

Any query in a BIT will reflect on some positions other than the original index. For example in a BIT of size 16 (1 to 16), updating the tree at index 5 will require updating cells 6, 8 and 16 as well. For a single processor BIT the sequence of operations which are necessary to perform a certain query at some position IDX is as follows:

- Perform the operation at position IDX.
- Update IDX to obtain the next position.

These two steps are executed until IDX goes out of bounds. And according to the structure of BIT these operations will be executed no more than B times. Where B is the number of bits in the binary representation of the tree size. So a single query will reflect on a maximum of B positions. The key term in converting the sequential procedure to parallel operations is to calculate the B or less positions concurrently direct from the original position IDX, instead of calculating one position after another, each depends on the previous one. The proposed model uses multiple processors with identical architectures. Each processor is responsible for calculating one index from those which are necessary to perform a certain query. Hence,

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

the number of processors depends on the size of the BIT. This could be formulated as follows:
Ps = Ceil (Log (N+1))
Where Ps is the number of processors, N is the size of the BIT. For example, for a BIT of size 1023 (1 to 1023) the model will use Log (1023+1) processors.

1) Processor Architecture: The main function of a processor is to deduce one index among those which are necessary to complete a certain query. Consider a read query at index 27 (11011). The classical method will deduce the necessary indexes as the following,

11011	Base index
00001 -	Last non-zero bit
11010	next index
00010 -	Last non-zero bit
11000	next index
01000 -	last non-zero bit
10000	next index
10000 -	last non-zero bit
00000	STOP

Lets try to perform the following operations concurrently on the base index,

- Convert the least significant bit to '0'. This will generate the index (11010).
- Convert the least three significant bits to '0', This will generate the index (11000).
- Convert the least four significant bits to '0', This will generate the index (10000).

Those addresses are exactly the necessary for performing the read operation. Note that the index that will result from converting the least two significant bits to '0' has been bypassed. This is because the third least significant bit has value '0' in the base index. So the concurrent solution for a read query will be, assigning one processor to every bit in the base index then let the processor do the following,

- Check the value of the bit which is assigned to it. Suppose it is bit number **OD**.
- If it is '0', do nothing.
- Else convert the bits from (**OD**-1 down to 0) in the base index to '0's and output the result index.

The update query is a little more complex. Consider an update query at index 4 (00100). The sequential procedure will generate the necessary indexes in the following order,

00100	Base index
00100 +	Last non-zero bit
01000	next index
01000 +	Last non-zero bit
10000	next index
10000 +	last non-zero bit
100000	Stop

The following concurrent operations will generate the same indexes.

- Reverse the fourth least significant bit and convert the least three significant bits to '0"s. This will generate the index (01000).
- Reverse the fifth least significant bit and convert the least four significant bits to '0"s. This will generate the index (10000).

Note again that the address generated from applying the previous operations on the third least significant bit has been bypassed, this is because it has value '1'. Note also that although the least and the second least significant bits has value '0', the addresses generated by applying the operation on those bits have been also bypassed. This is because each of those bits has no bits with value '1' that are less significant to it.

So the concurrent solution for an update query is also obvious, Assign one processor to every bit in the base index. And let the processor do the following,

- Check the value of the bit which is assigned to it. Suppose it is bit number **OD**.
- If it is '1', do nothing.
- Else if it has no bits with value '1' that are less significant to it, also do nothing.
- Else, reverse bit **OD** and convert the bits from (**OD**-1 down to 0) in the base index to '0's then output the result index.

So the overall functionality of the processor is to receive the index, the kind of operation (Read, Update) and the order of the bit associated to it, then perform one of the scenarios illustrated before, according to the type of operation.[2], [9]. Figure (7) shows a flow chart for the functionality of one processor.

2) Memory Architecture: As mentioned before, The presented model uses a distributed system disciplines. This requires the system memory to be partitioned into a number of chunks or fragments. The key term in the partitioning procedure in any distributed system, is to partion the memory in a manner that minimize the total communication overhead between the memory chunks. In this system the memory has been divided in a way that will completely prevent any need of communication between the memory fragments.0

let's look closely to the process of generating the addresses which are required to complete some query. Note that a certain processor will be assigned a certain order (the order of the bit in the index that the processor will check with all the bits that are less significant to it). Note also that a certain processor that is assigned a bit order **OD** (right to left starting from zero), will certainly generate indexes that have the bit number **OD** equal to '1' and all the bits that are less significant to it equal to '0's. for example if a processor is assigned an order of three, this yields to two possibilities, even the processor will not generate an index, or the processor will genrate an index with the fourth least significant bit equal to '1', and the least three significant bits equal to '0's. This yields us to two different rules,

1) Any processor with a certain order will never generate an

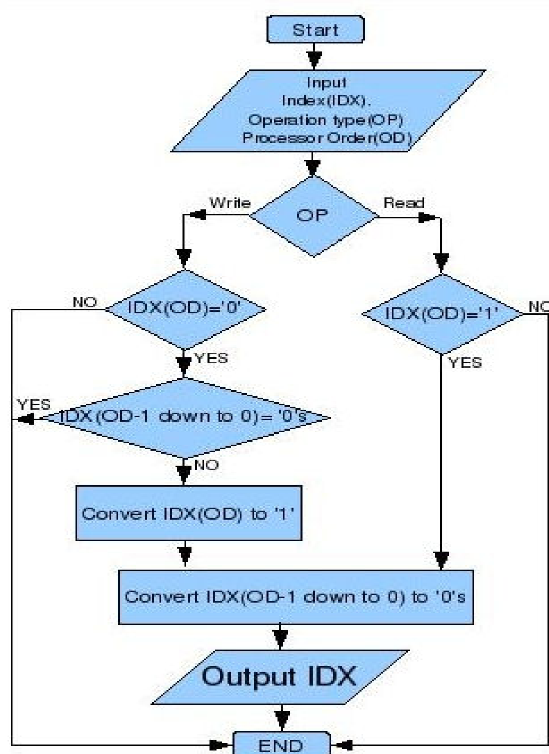

Fig. 4. Processor Function

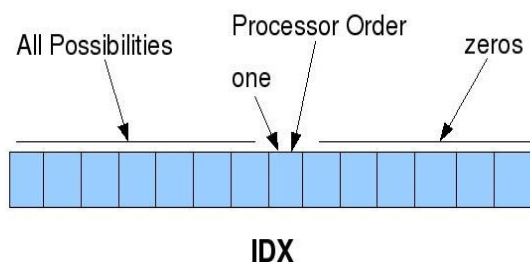

Fig. 5. Address Description

address that could be generated by any other processor with different order.

2) The number of possible addresses generated by a certain processor is 2^{B-OD-1}. Where **B** is the number of bits in the binary representation of the tree size and **OD** is the bit order assigned to the processor again right to left starting from zero.

It is now obvious that any address which is generated by a processor that is assigned an order zero has a maximum of 2^{B-1} different possibilities, which is equal to half the tree size (if the tree exhausts all possible combinations of the address bits). Similarly there will be 2^{B-2} (one quarter of the tree size) possibilities for the address generated by the next order processor. Then one eight for the next one, and so on. So the memory could be partitioned as illustrated in figure (6)

There is still a small pending issue that is necessary to complete the design of the distributed memory system. The addresses which are generated by the processors are compatible with a single uniform memory. In other words the addresses are made to access one memory chunk of size N. So there must be a method to map the addresses generated by the processors to the actual right addresses in the corresponding distributed memory fragments. After Recalling the memory address details illustrated in figure (5), it is clear that in any address that is generated by a processor with a certain order **OD**, bits (0 to **OD**) are always constant (tied to '1' followed by '0's to the right). This means that the actual bits responsible for accessing the local memory fragment of this processor are the bits which are more significant than **OD**, hence the mapping process will be taking these bits as the actual address and exclude the constant part. This process is equal to shifting the address right until the constant bits are excluded. [2], [8], [10]

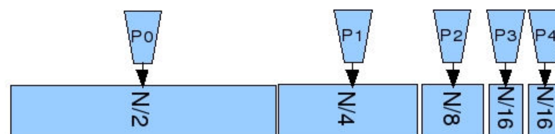

Fig. 6. Memory System

3) Parallel Adder: Far till now, the system consists of a number of processors, each one connected to its own local memory. Meanwhile there isn't any sort of communication between any two processor, any two memory fragments or any processor and any memory fragment other than its local one. If the system ends up in this state, then it will reach the optimum speed up that was mentioned by Amdahal, as there will not be any bottle nicks in the system. Unfortunately the system isn't yet complete. Recalling the read query process yields us to a missing part in our system. After calculating the necessary addresses for a read query and retrieving the tree values at these addresses, we need to add these values to get the final result of the read query. Adding these values sequentially using the naive method will lock the system again to a bottle nick that will reduce the speed up dramatically

The system uses some sort of parallel adder that works in **Log(S)**, where **S** is the number of values to be added. This adder works in a Binary Balanced Tree manner as illustrated in figure(7). As long as the complexity of this adder is **Log(S)**. And as long as there are a maximum of **Log(N)** values to be added in a single read query, so the complexity of the adder could be rephrased to **Log(Log(N))**. Where **N** ,again, is the size of the **BIT**. [2]

C. Implementation

As mentioned before, the system was implemented and simulated using **VHDL** and **Verilog** Hardware Description Languages, Under xilinx ISE 10.1. The implementation consists of four main parts, processors, memory system, control unit and the parallel summer. All are integrating in a single unit which represents the **BIT** system.

The processor unit was implemented using **VHDL**. It has the following signals as its interface,

- The query type (Read or Wright).
- The address in the tree which the query requires to read from or write to.

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

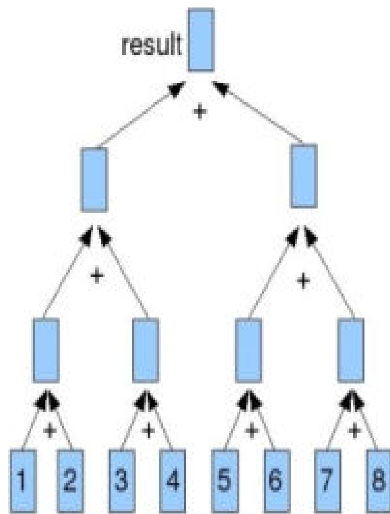

Fig. 7. Parallel Adder

- The order assigned to the processor.
- A clock signal which will trigger the processor to operate.
- The result address which is generated according to the query type, the input address and the order assigned to the processor.

The architecture of the processor was implemented in a behavior structure. In other words, the functionality of the processor was descried in an algorithmic or data flow form. It is worth stating that the address mapping process which was mentioned in the design of the memory system is performed by the processors as well. This means that the addresses generated by the processors in the real system are already compatible with the distributed memory system.

The time and gate costs differ slightly according to the size of the operation address. But they are still constant if we look to them in an algorithmic complexity. The variation of these costs are related to the hardware extention that is necessary when changing the word size of any system.

The memory system was implemented using Verilog in also a behavior structure. The memory fragments were mapped to some block ram units, each one of these units represent a fragment in the distributed memory system.

The control unit is the part that generates the read or write signals to the memory according to the state of the query. Remember that an update query requires reading the values of the tree at some positions and updating them, then writting them back in the same positions. This sequence yield us to the fact that withen any updating query some read and write operations will be performed in the memory system, which force the existing of a control unit to synchronize this procedure.

The implementation of the parallel adder was done using Verilog as well. This was done by using multiple ordinary two words adder in the form that was illustrated in figure (7).

These components have been connected together with some other pre-made components (Multiplexers, adders) in a single component to compose the complete **BIT** Parallel system. The system components operate on the positive edge of the system clock. The frequency of the system clock must be less than the minimum of the allowed maximum frequencies of each component. The time cost of the wiring between the components must be taken into consideration when choosing the frequency of the system clock. The timing analysis of the system and all its components could be performed easily using the timing analyzer of the xilinx ISE 10.1, taking into consideration that the system uses multi-cycle structure. In other words a single query will need more than one clock cycle to finish. In our system , an update query will consume four clock cycles to finish. First calculating the necessary addresses concurrently, then reading the memory fragments at these addresses, then updating these values, and finally writing them back to the memory fragments. The read query requires (2 + Log(Log(N))) clock cycles to finish. First calculating the necessary addresses, then reading the values of the memory fragments at these addresses, then it will consume Log(log(N)) clock cycles to add these values concurrently using the parallel adder. [3], [1]

III. Results

The analysis of the time complexity of the design is now ready to be illustrated. There are three main parts, the processors, the memory system and the parallel summer. The processors work concurrently in a constant time complexity. The distributed memory fragments work also concurrently in a constant time complexity. Unfortunately The parallel summer works in **Log(Log(N))**.

An Update query requires the following steps,

- The processors calculate the necessary addresses concurrently. This is done in a constant time complexity.
- Read the tree values at these addresses from the memory fragments. This is done also in a constant time complexity.
- Add the update value to all values that have been read in the previous step. This could be done also in a constant time complexity using multiple separated adders.
- Write these values again after being updated to the same addresses in the memory fragments. Again this takes constant complexity.

Therefore an update query could be done in constant time complexity.

A Read query is slightly different. The steps of performing such a query are,

- The processors calculate the necessary addresses concurrently. This is done in a constant time complexity.
- Read the tree values at these addresses from the memory fragments. This is done also in a constant time complexity.
- Sum all these values in **O(Log(Log(N)))** using the parallel summer illustrated before.

Accordingly a read query could be done in **O(Log(Log(N)))**. So The system ends up with the following,

1) The time complexity of any read query is **O(Log(Log(N)))**.
2) The time complexity of any update query is constant.

[5], [9]

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

Open Science Index, Computer and Information Engineering Vol:3, No:6, 2009 publications.waset.org/5410.pdf

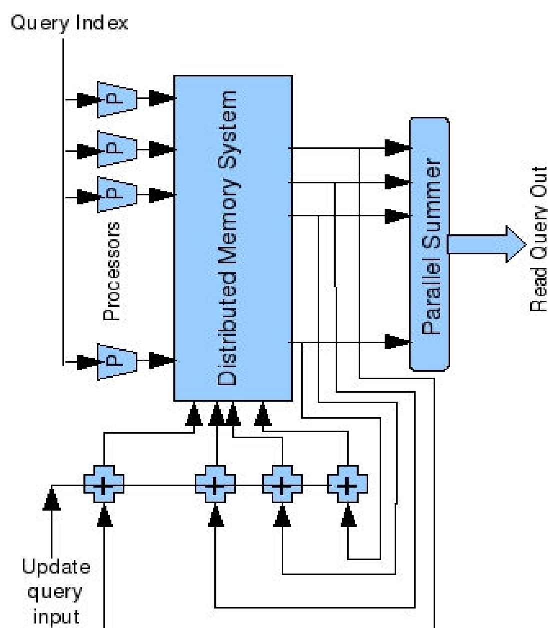

Fig. 8. Complete System Diagram

IV. Conclusion

A new parallel distributed system has been introduced which has managed to reduce the time complexity of Binary Indexed Trees. This was done by efficiently partition the problem space into small distributed fragments which operate concurrently. The amount of communication between the system fragments has been reduced as much as possible in order to obtain a speed up that is near to the maximum enhancement offered by the distributed parallel system.

V. Future work

The proposed system could be more reliable if it is implemented on a distributed work stations instead of the hardware implementation proposed in this paper. More researches could be performed to reduce the time complexity of the read query to constant as well.

References

[1] *XST User Guide, Xilinx Inc.*
[2] *The Algorithm Design Manual*. TELOS, The Electronic Library of Sciences, 1998.
[3] *Digital Design: Principles and Practicese (4th Edition)*. Prentice Hall, 2005.
[4] Peter M. Fenwick. A new data structure for cumulative frequency tables. *SOFTWAREPRACTICE AND EXPERIENCE*, 1994.
[5] Ralph C. Hilzer Helen D. Karatza. Performance analysis of parallel job scheduling in distributed systems. *Annual Simulation Symposium*, 2003.
[6] TOPCODER Inc. Binary indexed trees . http://www.topcoder.com/tc?module=Statices, December 2008.
[7] TOPCODER Inc. Range minimum query and lowest common ancestor. http://www.topcoder.com/tc?module=Staticstor, January 2009.
[8] W. E. Johnston Q. M. Malluhi. Approaches for a reliable high-performance distributed-parallel storage system. *High Performance Distributed Computing*, 1996.
[9] Linda M. Wills Randall S. Janka. Specification and synthesis of real-time embedded distributed and parallel multiprocessor-based signal processing systems. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2000.
[10] Vijay K. Naik Vinod G. J. Peris, Mark S. Squillante. Analysis of the impact of memory in distributed parallel processing systems. *Joint International Conference on Measurement and Modeling of Computer Systems*, 1994.

Acknowledgment

The authors would like to thank Professor Dr. Ismail, Ossama for his infinite support.

Ahmed M. Elhabashy Graduate Teaching Assistant. Arab Academy for Sciences and Technology. College of Engineering. Computer Engineering Department. a.elhabashy@hotmail.com.

A. Baes Mohamed Associate professor. Arab Academy for Sciences and Technology. College of Engineering. Computer Engineering Department. IEEE senior member (2001). baithmm@hotmail.com

Abou El Nasr Mohamad Associate professor. Head of the Computer Engineering Department. Arab Academy for Sciences and Technology. College of Engineering. m.abouelnasr@gmail.com