*2011 International Conference on Communication Engineering and Networks*
*IPCSIT vol.19 (2011) © (2011) IACSIT Press, Singapore*

# A Theoretical Evaluation of AriDeM using Matrix Multiplication

Patrick Mukala, Johnson Kinyua and Conrad Muller

Faculty of Information and Communication Technology, Tshwane University of Technology, Pretoria, South Africa
patrick.mukala@gmail.com, jdmkinyua@hotmail.com, darnoc@me.com

**Abstract**-The Von Neumann model has been used as a basis for sequential computation for decades. It is widely accepted that high performance computation is better achieved using parallel architectures and is seen as the basis for future computational architectures with the ever increasing need for high performance computation. We describe a new model of parallel computation known as the Arithmetic Deduction Model (AriDeM) which has some similarities with the Von Neumann. A theoretical evaluation of the model is performed using matrix multiplication and compared with the Von Neumann model. Initial results indicate AriDeM to be more efficient in resources utilization.

**Keywords**-parallel computation; Von Neumann; AriDeM; parallel processing; computer architecture

## 1. Introduction

With the ever increasing need for high performance in computing, parallelism has been considered as the future of computational architectures ([5], [3]). Much work has been done in attempting to develop better models for achieving higher performance compared with the traditional sequential Von Neumann model as described in section II.

We present AriDeM, which is a computational model based on arithmetic and natural deduction that is neither address nor instruction based. Instead, elements consisting of semantic information and values are processed in any order to create new elements by making deductions based on defined relationships and the semantic information elements. A theoretical evaluation of AriDeM has been performed using matrix multiplication and its performance compared with the von Neumann model.

The rest of this paper is organized as follows. In section II, a review of attempts to achieve high performance computation using various parallel architectures is presented. Section III describes the AriDeM architectural model using some examples. The theoretical evaluation of AriDeM is presented in section IV. The conclusions and directions for further work are given in section V.

## 2. Related Work

For more than three decades research workers have been trying to develop better models of parallel computation. Proposed models include the PRAM model the Data Flow model, the BSP model, and the Network model ([1], [2], [15], [6], [9]).While these models have played their part in the design of parallel architectures, there have been several attempts to improve them for better performance. We briefly summarise some of the more notable attempts.

### 2.1 Parallel Random Access Machine (PRAM)

The parallel random access machine (PRAM) model was developed in parallel computing in order to emulate the success of the RAM model in serial computing ([12], [1]). This model consists of a set or grouping of processors that execute the same program using a shared memory in a lockstep fashion. Each

processor executes its allocated instructions and accesses a memory location in one time step independently of the memory location [15]. However, the model has been refined numerously based on a number of factors.

One important factor was its inability to efficiently manage concurrent memory access among processors. This led to the development of the CRCW PRAM, CREW or EREW PRAM in order to alleviate the problem. Another variant is the Module Parallel Computer which has a shared global memory structure organized in such a way that the global memory is broken down into a number of modules with each memory module allowing a single memory access based on a specific time frame ([21], [6]).

Another factor is the inability to coordinate synchronization efficiently among the networked processors. To help solve this problem, other variants such as the XPRAM were developed as part of APRAM (Asynchronous Parallel Random Access Machine) variants. In these variants, rather than using a global clock as in previous variants, synchronization occurs on a periodic basis between specific intervals [17].

Other factors include latency and a related variant developed to overcome this challenge is the LPRAM where compromises were done at the local memory access in order to access the global memory [1]. Another variant developed in this regard was the BPRAM. Here there is an emphasis on data parallelism where by the communication costs incurred on memory access occur first at the local memory level and then from the global memory for each message or block transfer that takes place [2].

## 2.2 The data Flow Model

The Data Flow model was believed to be an alternative to the parallel von Neumann computation model ([15], [20]). It was designed to express parallelism more effectively. As the name suggests, the Data Flow model is a parallel computational model that is data driven. This means that the sequence in which the execution of instructions occurs totally depends on the availability of operands that are needed by the corresponding instruction while unrelated instructions are executed concurrently without interference [20].

Najja and co-workers (1999) present four main advantages of using the data flow model. The order in which operations occur is determined by true data dependencies and there are no constraints imposed by artificial dependencies that could be introduced by stored program model including memory alias etc [16]. The model represents parallelism at all levels in during computation, this implies that at the operation level, there is fine-grained parallelism displayed and this can be exploited within a processor, function or loop level parallelism can also be exploited across processors while systolic arrays exploit only fine grain parallelism [16]. At the semantic level, combinational logic circuits are naturally functional. This means that the output depends on the inputs. Hence, these circuits are easily synthesizable from graphs in the data flow diagram.

However, despite these advantages the model has some limitations. According to Dennis [7], there are still problems to be addressed with the dataflow architectural model. One of the limitations is related to the type of programs data flow computers can support. Additionally, much research need to be conducted in order to improve the problem of mapping high-level programs into machine-level programs that effectively make use of machine resources [7].

## 2.3 The Bulk Synchronous Parallel (BSP) Model

The Bulk-Synchronous Parallel (BSP) developed by Valiant was designed to allow the optimal execution of architecture-independent software on a diversified number of architectures ([19]; [8]). In the BSP model, the configuration of a parallel computer consists of a set of processors associated with their own respective local memories, and a communication network that performs interconnection message passing by routing packets in a point-to-point fashion between processors ([15], [8]).

A computation in a BSP Model is divided into sequential super-steps. In this sequence of super-steps, each of these super-steps is also a sequence of steps followed by a barrier of synchronization where all memory accesses occur [19]. During each super-step, a processor carries out a number of activities based on its data and programs. These activities include carrying out computational steps from its threads on values received at the beginning of the super-step. Each processor can also perform additional operations on its local data, can send and receive a number of messages and packets [19]. A packet sent by a processor during a super-step, is delivered to the destination processor at the beginning of the next super-step and a synchronization barrier is used to separate successive super-steps for all processors [8].

Although the model has been predicted to provide positive results, it has also been through a number of refinements. These refinements led to the creation of the extensions of the BSP model including special mechanisms for parallel-prefix computation (PPF-BSP), broadcasting (b-BSP), and concurrent reads and writes (c-BSP) [8]. There are still a lot of challenges to overcome within the BSP model itself such as developing an appropriate programming framework for the model, as pointed out by McColl [15].Many other models such as the Distributed Memory Model (DMM) and the Postal Model have been developed over the years for providing alternative models for parallel computation..

# 3. The Arithmetic Deduction Model (AriDeM)

AriDeM is an architectural model based on natural rules of computation. AriDeM is similar to the von Neumann model in some ways but, instead of processing instructions it processes elements. For this reason, we refer to AriDeM as the element model and the von Neumann model as the instruction model.

## 3.1 Main Concepts of AriDeM

In AriDeM, an element is made up of three parts: an identifier, indices and a value as shown in Table 1. For example, *length = 5cm* is an element.

TABLE I. Element Description

| Field | Description | Comment |
|---|---|---|
| identifier | binary number | used to identify relations |
| indices | list of binary numbers | ensures unique meaning |
| value | one or two binary numbers | can be a single or tuple value |

Since an element refers to relations (also referred to as relationships), the identifier points to a list of relations as described in Table 2.

TABLE II. Description of Relations

| Field | Description | Comment |
|---|---|---|
| identifier | binary number | used for the identifier of the new element |
| operation | binary number | specifies the operation required the new element |
| parameters | binary numbers | used by operation |

For example, $square_area = length^2$ is a relation. An expression of how these two concepts relate is shown in Figure 1. AriDeM expresses through Figure 1 that mappings can describe relations between sets in the problem domain and given an element in the domain of one of these relations enables a deduction to be made about an element in the range. For example given the relation

$square_area = length^2$, and the element

*length = 5cm,*

one can deduce based on the given relation,

$square_area = 25cm^2$

This means that when an element like *length = 5cm* comes into existence, it is processed by applying it to relations such as $square_area = length^2$. The element together with the relation is then used to deduce another element and in this case, $square_area = 25cm^2$. Once the element has been processed, it is discarded and the newly created element can be applied to other relations that possibly create new elements.

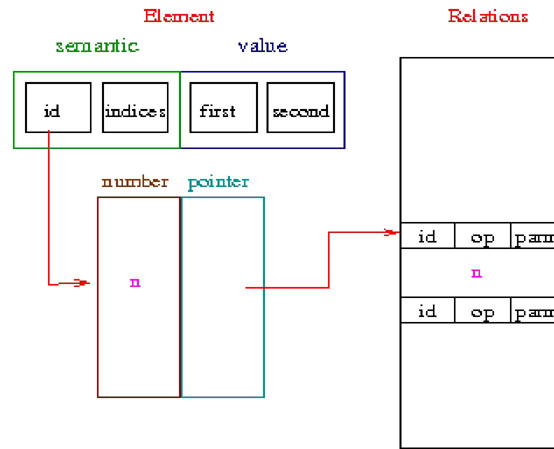

Figure 1. Interplay between element and relations

## 3.2 Execution in AriDeM

The elements consist of information that uniquely conveys the meaning of the data as well as a value. The information is used to determine how the element is processed. Processing an element results in the creation of zero or more elements. The program execution completes when there are no more elements to process.

The execution cycle for both the instruction and element models is as follows.

TABLE III. Instruction vs. Element Model Execution Cycle

| **Instruction Model** | **Element Model** |
|---|---|
| Get instruction | Pop Element |
| Get Data | Get relation |
| Perform op | Perform op |
| Store result | Push result |

To illustrate the mechanism we start with a simplified version of the model. The relationships are expressed as a definition of an element e.g. $a = -b$. This can also be written as $b \Rightarrow a$, which is an alternative AriDeM notation for a relation. The elements are expressed as a value associated with an identifier e.g. $b = 5$. In this example the element $b = 5$ starts on the stack of unprocessed elements.

The first step of the cycle is to pop $b = 5$. The second step is to use the identifier of the element to get the relation $b \rightarrow a$. The last step is to compute −b or −5 to create the element $a = -5$ and store it on queue of unprocessed elements.

# 4. Theoretical Evaluation of AriDeM

The first evaluation of the AriDeM architectural model involves a comparison between the Von Neumann model and AriDeM using.matrix multiplication.

## 4.1 Evaluation Metrics

A number of metrics could be considered in order to evaluate a model of parallel computation including computational parallelism, latency, synchronization, bandwidth, etc [14]. In this research, we limit the evaluation to two main metrics: the execution time and the number of instructions vs. number of elements.

The two models are evaluated on two levels: at the sequential level (using only one processor to execute the instructions), and at the parallel level (making use of more than one processor to distribute the instructions load across these processors and execute them). A conceptual view of how the element model processes elements and relations is given in the pseudo code below.

```
While (! stackEmpty )
{
        pop ( element ) ;
        new_element = element ;
```

```
        nextRe lat ion = relationTable [ element . id ].list;
        numberRelations  =  relationTable [ element . id ].size;
        for ( i =0; i<numberRelat ions ; i++)
        {
                new_element.id = relation [ nextRe lat ion ].id;
                case ( relation [ nextRe lat ion ] . op )
                // arithmetic and logical operations
                add :
                {
                        new_element. value [ 0 ] += element . value [ 1 ] ;
                        break ;
                }
                . . .
                // index operations
                indexAdd :
                {
                        new_element. index [ relation [ nextRe lat ion ] . parm0 ] ]
                        += relation [ nextRe lat ion ] . parm1 ] ;
                        break ;
                }
                . . .
                // if operation
                i f o p :
                {
                        i f ( ! new_element . value [ 1 ] )
                        goto skip ; // do not create a new element
                        break ;
                }
                . . .
                // tuple operations
                tupl e0 :
                {
                        int found ;
                        tupl e = checkHashTable ( newElement , found ) ;
                        i f ( ! found ) goto skip ;
                        break ;
                }
                . . .
                push ( newElement ) ;
                skip :
        }
}
```

In this pseudo code, it is depicted how the element model operates using elements and relations. On request, a process (master) sends and receives buffers of elements to and from other processes (slaves). The processes send buffers of elements in order to match them with the corresponding elements stored in the hash table. Based on the existing relations, a dedicated process handles tuples. The whole process carries on until there are no more elements to be processed.

## 4.2 Evaluation and Results

In order to conduct this study, a naive algorithm was selected to simplify the analysis and discussion. First, we look at the performance of both models on a single processor and later on multiple processors.

*1) Instructional model on a single processor.* The following naive multiplication algorithm in C has three simple nested for loops ranging over indices *row*, *co*l, and *k* in this order:

```
int main ( )
{
        int a [ 3 ] [ 3 ] , b [ 3 ] [ 3 ] , c [ 3 ] [ 3 ] ;
```

```
        int row , col , k ;
        for ( row=0;row<3; row++)
                for ( c o l =0; col <3; c o l++)
                {
                        a [ row ] [ c o l ]=0;
                        for ( k=0;k<3;k++)
                                a [ row ] [ c o l ]+= b [ row ] [ k ] _ c [ k ] [ c o l ] ;
                }
}
```

Figure 2. C Matrix Multiplication Program for the instruction model

The above code snippet depicts the computation of matrix multiplication using three nested loops to cater for each row, column and the last operation, which is the summation of the operands. This naive algorithm was selected to simplify the analysis and discussion. In this computation, three for loops are used to perform matrix multiplication. The first two loops are for the elements of the product matrix, and the third is for multiplying a row from the first matrix by a column from the second matrix.

Considering the above code, in the first loop *for (row=0; row<n; row++){} : row* is used to access each row based on the number of rows in the first matrix, in the second loop *for (col =0; col <n; c o l++) {}: col* is used to access each column based on the number of columns in the second matrix while in the third and last loop *for (k=0;k<n;k++):k* is the number of columns in the first matrix as well as the number of rows in the second matrix (since the number of columns of the first matrix is equal to the number of rows of the second matrix).

After computation, this algorithm compiles into 127 instructions on a simplistic level. However, on a more realistic comparison there are 64 instructions.

*2) Element model (AriDeM) on a single processor.* On the other hand, we look at how computation is done to perform the same operations using AriDeM. In the element model, computation occurs by considering the following development of how to compute a matrix multiplication:

$A = B \times C$ can be expanded to

$A_{r,c} = \sum_{k=1}^{m} B_{r,k} \times C_{k,c}$ , where **r** represents each row and **c** each column and further refined to

$$A_{r,c} = \sum_{k=1}^{m} b_{r,c,k} \times C_{r,c,k}$$

where :

$$b_{r,c,k} = B_{r,c}; \mathrm{k} \in [1,n]$$

$$c_{r,c,k} = C_{r,c}; \mathrm{k} \in [1,n]$$

To complete the full description of the multiplication the summation needs to be defined and it can be expressed as:

$a_{r,c,k} = b_{r,c,k} \times \mathrm{c}_{r,c,k}$

$s_{r,c,n} = a_{r,c,n} - a_{r,c,n}$ convenient way to set value to zero

$$s_{r,c,k-1} = s_{r,c,,k} + a_{r,c,k}$$

$$A_{r,c} = S_{r,c,o}$$

Given two matrices *A* and *B*, their product *C* is defined as *C[i,j] = Σk A[i,k] * B[k,j]*. Namely, computing the value of an element of the result *C* requires summing each element of the same row in *A* with each corresponding element of the same column in *B*. This algorithm has a complexity of $O(n^3)$ which means that multiplication is very computationally intensive for large matrices.

Let us consider the same case as an example to expand this development in a little more details. Consider the computation of a matrix multiplication

*C = A x B*. Given the following matrices for A and B:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix}$$

$C$ can be expressed as:

$$C_{r,c} = \sum_{i=1}^{n} A_{r,i} \times B_{i,c}$$

where r is the row and c is the column. We can now slightly modify this expression by introducing *ar;c;i* and *bi;r;c* . This can be expressed as follows:

$$a_{r,c,i} = A_{r,c} \forall i \in [1,3], b_{r,c,i} = B_{r,c} \ \forall i \in [1,3] \cdot$$

The matrix multiplication can then be expressed as:

$$C_{r,i,c} = a_{r,i,c} \times b_{r,i,c} \forall_{r,i,c} \in [1,3]$$

an $C_{r,c} = \sum_{i=1}^{n} c_{r,i,c}$

or

$$S_{r,n.c} = C_{r,n,c}$$

$$S_{r,i.c} = S_{r,i+1,c} + C_{r,i,c} \text{ if } i \neq 0$$

$$C_{r,c} = S_{r,o,c}$$

On a simplistic level, there are 19 relations versus 127 instructions. On a more realistic comparison however, there are 19 relations for the element model versus 64 instructions of the von Neumann model. Considering at the iterative section, there are only 9 relations versus 57 instructions. If the matrix is an $n \times n$ matrix, the inner loop of the instruction model will execute $n^3 \times 40$ instructions whereas there are $n^3 \times 16$ instructions. Not surprising, these two models are both of the same order of complexity.

The instruction model has a significant number of memory accesses instruction compared with 4 tuples relations in the element model. Based on this, the element model should perform better by a considerable factor. Given the large number of memory accesses for the instruction model, the element model appears to perform better under the circumstances. It performs six times faster than the instruction-based model.

*3) Instruction model in parallel.* To evaluate the parallel versions of the previous programs, we also use matrix multiplication. The results of these two programs show how the models compare for the parallel versions. The computation of large matrices multiplication requires substantial time of the order of $O(n^3)$ in complexity.

We consider writing a simple C version using MPI. The first decision is how the work will be distributed across the processes. The approach chosen is to pass a row from the first matrix and column from the second to a process to calculate the element in the resulting matrix. To facilitate this one needs to have the second matrix in column order. Another decision is which process will do the distribution and how.

In Figure 3, we notice that one of the main differences between a sequential program and a parallel program using the instruction model is the existence of parallel form of loops where each iteration of the loop is executed in parallel on a separate processor. When executing parallel loops, all processors execute the loop instructions synchronously. Any reference to the PRAM model or parallel von Neumann model is usually associated with assumptions on the outcome of having concurrent access to the same memory location.

We notice that in order to parallelize the matrix multiplication on the instruction model, MPI is used as shown in Figure 3. This is an indication of the complexity of the instruction model. To run a given algorithm initially working in serial computing in parallel, developers and hardware designers have to rewrite the initial code to express parallelism. In AriDeM however, the code remains the same. Whether we need to test it in sequence or in parallel, the architecture caters for this. There is no need for the programmers to modify the source code to run the program in parallel.

```
int main(int argc, char *argv[])
{
      int slave, noslaves, result[sizeM];
      double MPI_Wtime();
```

```
      double startT,endT;
printf("1\n");
     //MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &slave);
     MPI_Comm_size(MPI_COMM_WORLD, &noslaves);
if(slave==master_slave)
{
//==========================
// master slave
//==========================
int A[sizeM][sizeM], B[sizeM][sizeM], C[sizeM][sizeM], row, col,result[4];
MPI_Status status;
for(row=0; row<sizeM; row++)
  for(col=0; col<sizeM; col++)
    scanf("%d", &B[row][col]);
    for(row=0; row<sizeM; row++)
  for(col=0; col<sizeM; col++)
    scanf("%d", &C[col][row]);
   startT=MPI_Wtime();
   for(row=0; row<sizeM; row++)
  for(col=0; col<sizeM; col++)
  {
      MPI_Recv(result, 4, MPI_INT, MPI_ANY_SOURCE,
                      MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      if(result[1]!=-1)
      A[result[1]][result[2]]=result[3];
           result[1]=row;
      result[2]=col;
      MPI_Send(result, 4, MPI_INT, result[0],
      resm,MPI_COMM_WORLD);
      MPI_Send(B[row], sizeM, MPI_INT, result[0],
      rowm,MPI_COMM_WORLD);
      MPI_Send(C[col], sizeM, MPI_INT, result[0],
colm,MPI_COMM_WORLD);
  }
for(row=0; row<noslaves-1; row++)
{
      MPI_Recv(result, 4, MPI_INT, MPI_ANY_SOURCE,
      resm,MPI_COMM_WORLD,&status);
      A[result[1]][result[2]]=result[3];
     result[1]=-2;
      MPI_Send(result, 4, MPI_INT, result[0],
  resm,MPI_COMM_WORLD);
}
  for(row=0; row<sizeM; row++)
{
  for(col=0; col<sizeM; col++)
      printf("% d,",A[row][col]);
   printf("\n");
}
} //end Master
else
{
//  *********
//  slave
//  *********
int B[sizeM], C[sizeM], result[4], i;
```

```
MPI_Status status;
result[1]=-1;
result[0]=slave;
MPI_Send(result, 4, MPI_INT, 0,
resm,MPI_COMM_WORLD);
printf("3\n");
while(1)
{
      MPI_Recv(&result, 4, MPI_INT, 0,resm,
      MPI_COMM_WORLD,&status);
      if(result[1]==-2)break;
      MPI_Recv(B, sizeM, MPI_INT, 0,
      rowm,MPI_COMM_WORLD,&status);
      MPI_Recv(C, sizeM, MPI_INT, 0,
      colm,MPI_COMM_WORLD, &status);
      result[3]=0;
      for(i=0;i<sizeM;i++)
      result[3]+=B[i]*C[i];
      MPI_Send(result, 4, MPI_INT, 0,
      resm,MPI_COMM_WORLD);
}
} //end slave
endT=MPI_Wtime();
if(slave==master_slave)printf("exec time %lf %lf %lf\n", startT, endT,
endT-startT);
MPI_Finalize();
}
```

Figure 3. Parallel C Matrix Multiplication Program

The AriDeM program was unchanged since the same program is used to process elements on both a single processor and in parallel. However, the instruction-based program requires a complete rewrite. Additional instructions are required to synchronize between the processors and to distributed data. In both cases, the instructions or relations can either be in shared memory or have a copy on each processor. In the element-based approach, the relations can be distributed across processors and elements can be distributed to the appropriate processor.

According to the parallel program shown above, which demands a great amount of memory access for loading original matrix blocks and storing partial results, a successful implementation depends largely on memory efficiency. Improving data access locality and reducing memory access conflicts are two aspects to achieving high memory efficiency.

Considering our matrices $A$ and $B$ of size $n * n$, a standard parallel algorithm requires $2*n^3$ arithmetic operations. Hence, the optimal parallel time on $p$ processors will *be* $2*n^3/p$ time steps (arithmetic operations).

*4) AriDeM in parallel.* AriDeM does not require a complete rewrite of the code to run the program in parallel. The same code run on a single processor can be run on any number of processors without any problem. Furthermore, AriDeM uses fewer elements and memory accesses to execute a given progam which translates to a performance improvement compared with the traditional model due to reduced communication costs. For the element model, we were able to run the matrix multiplication program on any number of processors without any alterations to the code. The experiment showed that the matrix multiplication program could be executed in AriDeM without any change on 1, 2, 4 or 8 processors. .To achieve higher performance, it is often advantageous for an algorithm to use less space, because more space means more time. A faster algorithm that multiplies the matrices and adds them in place was used in our evaluation.

# 5. Conclusion and Future Work

The objective of the theoretical evaluation was to assess how the element model performs compared with the von Neumann model on a single as well as multiple processors. The results obtained demonstrated that the element model is more efficient in terms of resources management because fewer elements were

processed to accomplish the same tasks in the elements model as compared to the number of instructions executed to perform the same task for the instruction model.

Matrix multiplication was used to evaluate the parallel versions of both programs. The results of these two programs show how the models compare for the parallel versions. The computation of large matrices multiplication requires a relatively large amount time of *O(n3)* order in complexity. Hence, for the execution of this algorithm in parallel, the parallel algorithm required $2*n^3$ arithmetic operations. Hence the optimal parallel time on $p$ processors will be $2*n^3/p$ time steps.

Considering the results of the parallel versions of the program for both models, a complete rewrite of the code for the von Neumann model was required to parallelize the program. The simulation of the AriDeM architectural did not require a rewrite of the program by contrast and no alterations were required to run the program on any number of processors.

This work has shown that it is possible to express a program by focusing on the static relationships and leaving the resource allocation to the computation mechanism. The evaluation shows that it is possible to allocate processor resources more effectively by being able to distribute work at the fine grained level of a single operation. In future work, we intend to explore how the model could be implemented. Furthermore, more evaluations need to be conducted in order to test the model with a larger class of programs.

# 6. References


[1] Aggarwal A., Chandra A., Snir M., 1990.Communications Complexity of PRAMS: Theoretical Computer Science Vol. 71, pp. 3-28.

[2] Aggarwal A., Chandra A., Snir M., 1989. *On Communications Latencies in PRAM Computations* .USA: 1st Symp. on Parallel Algorithms and Architectures

[3] Asanovic K., Bodik R., Catanzaro B.C., Gebis J.J., Husbands P., Keutzer K., Patterson D. A., Plishker W. L., Shalf J., Williams S.W., Yelick K. A., 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* USA : Electrical Engineering and Computer Sciences ,University of California at Berkeley.

[4] Barnoy A.,Kipnis S.,1992. *Designing Algorithms in the Postal Model for Message Passing System*. USA: 4th Annual ACM Symposium on Parallel Algorithms and Architectures.

[5] Burger D., Keckler S. W., McKinley K. S., Dahlin M., John L. K., Lin C., Moore C. R., Burrill J., McDonald R. G., Yoder W., the TRIPS Team.,2004. *Scaling to the end of silicon with EDGE architectures*. USA : IEEE Computer Society Press Los Alamitos, CA, USA.

[6] Dadda L., 1991. *The Evolution of Computer Architectures.* Italy: Politecnico di Milano, Department of Electronics.

[7] Dennis J.B, 1980. *Dataflow Supercomputers*.USA: MIT Laboratory for Computer Science.

[8] Goudreau M.W.,Lang K., Rao S.B.,Suel T.,Tsantilas T., 1999.*Portable and Efficient Parallel Computing Using the BSP Model.*USA: IEEE TRANSACTIONS ON COMPUTERS, VOL. 48, NO. 7,

[9] Hwang K., Briggs, Fayi A., 1984.*Computer Architecture and Parallel Processing.* USA: McGraw-Hill.

[10] Iannucci A. R., 1988. *Toward a dataflow/von Neumann hybrid architecture*. USA : IBM Corporation -and- MIT Laboratory for Computer Science

[11] Kumm H.T., Lea R.M. 1994. *Parallel Computing Efficiency: Climbing the Learning Curve.* Aspex Microsystems Ltd. Brunel University Uxbridge Mddx UB8 3PH : United Kingdom

[12] Leiserson C., Maggs B., 1988.Communication-*Efficient Parallel Algorithms for Distributed Random-Access Machines*. USA : M.I.T Laboratory for Computer Science.

[13] Liu P., Aiello W., Bhatt S., 1993. *An Atomic Model for Message Passing*. USA: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures.

[14] McColl W.F., 1993. *General Purpose Parallel Computing*. Lectures on parallel computation, chapter 13.

[15] Najjar W.A.,Lee E.A, Gao G.R, 1999. *Advances in the dataflow computational model*. USA: Elsevier, Parallel Computing *25 (1999) 1907-1929*

[16] *Siu S., Sing*h A., 1997. Design Patterns for Parallel Computing Using a Network of Processors. University of Waterloo: Canada.

[17] Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., and Eggers, S. J. 2007. *The [WaveScalar] architecture*. USA: ACM .

[18] Valiant L.G.,1*990. "A Bridging Model for Parallel Computation".* New York, USA: ACM, vol. 33, no. 8, pp. 103-111.

[19] Warren P., 2004. *The future of computing-new architectures and new technologies.*UK: Btexact Technologies Research, IEE proceedings, Vol.151, No.1

[20] Winey E., 1978. *Data Flow Architecture.* USA: ACM (Association for Computing Machinery.