PRAM PROGRAMMING MODEL AND ALGORITHMS

Jan Liguš
Linköping University
584 34 Linköping, Sweden

Abstract

In the area of sequential computing the RAM (Random Access Machine) has successfully provided model of computation, in the realm of parallel computing there has been no similar success. The need for such unifying parallel model is growing by the demand for performance and the diversity among machines.

This survey is trying briefly to describe models of parallel computation and the different roles they serve in algorithm, language and machine design. In the end I tried to write in more details about the PRAM programming model and mentioned some fundamental algorithms used in it.

1. Introduction

Modeling complex phenomena is old as science itself. Choosing the right characteristic to model and incorporating requires as much artistic creativity as scientific methodology. Today models are ubiquitous-controlling large portions of financial markets, routing our air and space traffic, explaining the nature of our genetic make-up, tracking our weather, helping us to understand mystery of our universe and predicting our overall economic health.

Computer scientists use models to help design efficient problem solving tools. These tools include fast algorithms effective programming environments and powerful executions engines. Modeling can be used interactively with implementation in a symbolic process of problem solution, which is far more efficient than using either approach separately.

The survey is presented with a simple logical framework. This framework allows the wide array of models to be viewed somewhat systematically [1].

Computation model.

The solution to any given task begins with the design of a set of steps (an algorithm), which will realize the computational solution to an abstract problem specification. The problem can come from many different areas such as: mathematics, physics, biology, astrophysics etc.

In each domain the translation from problem to computational algorithm requires a model of computation. Development of serial computing produced among others, the widely accepted Von Neumann model expressed in the Random Access Machine model (RAM) [2]. Such computational model must clearly define an execution engine powerful enough to produce a solution to a relevant class of problems. Moreover such models need to reflect to the computing characteristics of practical computing platforms. These objectives enable the translation from an abstract formulation to an algorithm, which gives the desired solution.

Programming model.

An algorithmic specification is then translated into a sequence of machine-independent software instruction. The programming model facilitates the translation. The programming model provides a set of rules or relationships that defines the meaning of a set of programming abstractions. These abstractions are manifested in a programming language and architecture, which are an instantiation of that programming model.

Architectural models.

Architectural models describe a class of models used for a range of design purposes such as language implementation and machine design. Then models need to reflect the detailed execution characteristics of the actual computer. These models cover a wide range form high level representation or an architectural level to instruction execution models.

These models are primarily used in machine design and are commonly used in parallel computing to compare alternative architectures.

Performance analysis using these models expressed the design characteristics and concerns of evolving technologies and provides feedback onto many aspects of task solutions.

Serial vs. Parallel Models of Computation.

Using an abstract machine model we can facilitate consistency by providing clear and simple rendering of execution engine. Unfortunately the search for such a unifying parallel model has been less successful. The natural analog, the parallel random access machine meets this standard, but it is no so accurate as RAM. An example for unmodeled characteristics is the cost of non-local memory reference, which has a great impact on performance. The RAM is accurate enough to be used at least for asymptotic performance measures.

2. Computational Models

The RAM model of sequential computation

A common abstract view of a computer is the von Neumann model, which is a device consisting of a memory unit that holds data to be operated on, a program stored either in memory or in special program memory, and a central processing unit (CPU), which carries out the instructions of the program one after another [3]. The memory is divided into words. The central processing unit (CPU) consists of an ALU and a set of registers. The purpose of the registers is to store intermediate results. The instruction can be divided into three categories: Memory access instructions, Arithmetic logical instructions, and Conditional instructions.

Memory access instructions, which are used for loading the contents of a memory into a registers of the CPU, or for storing word into the memory.

Arithmetic logical instructions operate on words stored in registers in the processing unit, such as adding two words, forming the bitwise AND of two words, or shifting a word by given number of the position; there may also be an instruction for getting a random number [3].

Conditional instructions are used for controlling of the flow of operations including the halting instruction when computation is over.

In the CPU there is special register called PC(program counter). Each time unit this register is either advanced to the next instruction in the program, or changed due to the execution of a conditional instruction to point to some other instruction in the program.

This model is called RAM (see in Figure 1).

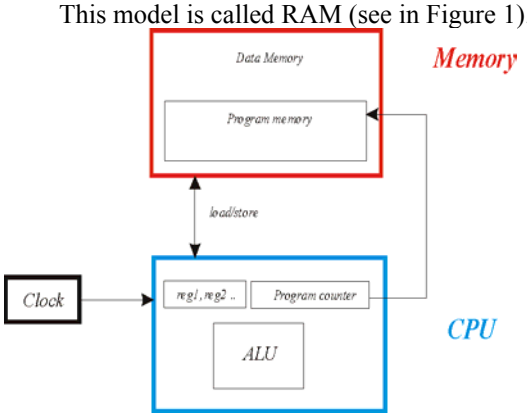

Fig. 1 The random access machine

The PRAM models

The PRAM model is natural generalization of the RAM model [3]. The PRAM model uses an unspecified number of identical processing units, which operate under the control of a common clock. All processors use a common shared memory. Each processor has its own number (ID), which uniquely identifies the processor. The processors perform instructions in unit time according to a stored program. The program may reside either in shared memory, or as will be assumed here in separate program memories that are private to each processor [3]. A PRAM models is depicted in Figure 2.

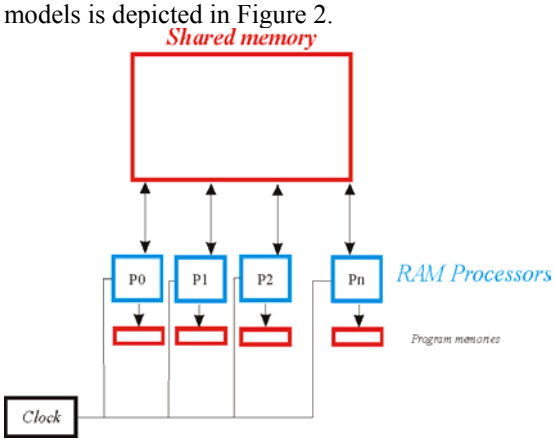

Fig. 2 The PRAM

The processor perform exactly one instruction each time step. We are not assuming that processors all have the same program. We assume that the PRAM is always in a consistent state, which can be described by the contents of the shared memory and the value of the program counter for each processor. According to Flynn's taxonomy we consider the PRAM as a multiple instruction multiple data (MIMD) machine [4].

In one program step all active processors simultaneously perform an instruction, not necessarily the same for each processor, which can be either type of memory access, arithmetic logical, or conditional.

Depending on what happens in such cases, we have several variants of PRAM model. It is common to distinguish between the following three main PRAM cases:

EREW.

Exclusive read, exclusive write, which means no two processors may read nor write to the same memory cell within the same memory cell within the same tome step.

CREW.

Concurrent read, exclusive write means that concurrent reading is allowed, that is more than one processor can read the contents of a memory cell in the same time step, but simultaneous writing to the same cell is not allowed.

CRCW.

Concurrent read, concurrent write provides both simultaneous reading and writing to a cell by several processors, although reading and writing the same cell in the same step is not allowed. There are some mechanisms [3], in order to resolve conflicts, if several processors simultaneously write to the same cell.

3. Quality for parallel algorithm

Measures of efficiency of PRAM algorithms are one of the most important things of parallel programming, since one can evaluate the quality of a parallel algorithms. Therefore the performance of a parallel algorithm for a given computational problem is measured relatively to a sequential algorithm for the same problem.

Let us first set up some basic symbols and definitions [7].

Symbols used in text.

- $t_A(n)$ = time needed by parallel algorithm A
  i.e., number of steps with input of size n (worst case)
- $t_s(n)$ = time needed by the best sequential algorithm worst case, input of size n
- $w_A(n)$ = work
  i.e., number of operations to be performed by A
- $p_A(n)$ = number of processors
- $p_A(n) \cdot t_A(n)$ = cost of the parallel algorithm

Definition 4.1 (work and time optimality) Let S be either an optimal or a currently best-known sequential algorithm for problem P, and let n denote the size of an instance of P.

A parallel algorithm A is work-optimal for P if $w_A(n) = O(t_s(n))$.

A parallel algorithm A is time-optimal for P if any other parallel algorithm would require at least $\Omega(t_A(n))$ time steps.

A parallel algorithm A is work time-optimal for P if any other work-optimal algorithm would require $\Omega(t_A(n))$ time steps.

[3]

Definition 4.2 (work efficiency) Let S be either an optimal or a currently best-known sequential algorithm for problem P, and let n denote the size of an instance of P.

A parallel algorithm A is work-efficient for P if

$$w_A(n) = t_s(n).O(\log^k(t_s(n)))$$

for some constant k≥1.

[3]

Theorem.

Brent's theorem: Any PRAM algorithm A which runs in $t_A(n)$ time step and performs $w_A(n)$ work can be implemented to run on a p-processor PRAM in

$$O(t_A(n) + \frac{w_A(n)}{p})$$

time steps

[8]

Brent's Theorem states that for an algorithm A which performs work $w_A(n)$ an algorithm A' exists with the same cost, that is $c_{A'}(n)=w_A(n)+ p_{A'}(n) \cdot t_A(n)$. Thus we can avoid in principle a situation where the cost of parallel algorithm is higher than the work it performs.

The theorem just assumes that we can identify the active processors. Thus we can relocate their work among the p available processors.

Speedup.

The speedup of the parallel algorithm is the improvement in runtime that a parallel algorithm is able to produce over a sequential algorithm for the same problem.

For a given problem, which we denote P, let us consider A as a parallel algorithm, and let S be an asymptotically optimal or currently best-known sequential algorithm for P.

We use the term absolute, when comparing the parallel runtime of A to an optimal sequential algorithm [3], whereas relative speedup is measured relatively to the sequential algorithm obtained from running A with only one processor.

Let $t_A(n)$ denote the runtime of A on an instance of P of size n. Let $t_s(n)$ be the runtime of S.

Definition 4.1 (asymptotic absolute speedup) The asymptotic absolute speedup of parallel algorithm A relative to a best (known) sequential algorithm S in the ratio

$$\frac{t_s(n)}{t_A(n)}$$

[3]

Definition 4.2 (Absolute speedup) The absolute speedup of a parallel algorithm A is the ratio

$$SU_{abs}(p,n) = \frac{t_s(n)}{t_A(p,n)}$$

[3]

Definition 4.3 (Relative speedup) The asymptotic relative speedup of parallel algorithm A is the ratio.

$$SU_{rel}(p,n) = \frac{t_A(1,n)}{t_A(p,n)}$$

[3]

Definition 4.4 (relative efficiency) the relative efficiency of parallel algorithm A is the ratio:

$$EF(p,n) = \frac{t_A(1,n)}{p.t_A(p,n)}$$

[3]

4. Fundamental PRAM algorithms

We are going to consider two important problems, whose solution illustrate issues and techniques that come up in the design of efficient PRAM algorithm.

4.1. The prefix-sum problem.

The idea of the prefix-sum problem is quite simple. There is a sequence of n numbers and we want to calculate their sum. Sequentially the problem is trivial to solve. One solution can be as follows: we store numbers in an array and scan the array from left to right, summing up the numbers as we go along.

We formulate the prefix-sums problem as follows. Given a set S, a binary associative operator ⊕ on S, and a sequence of n items $x_0,....x_{n-1}$ elements of S, compute the sequence *y* of prefix-sums defined by[3].

$$y_i = \bigoplus_{j=0}^{i} x_j \; for 0 \le i < n$$

Solving the prefix-sums problem using a sequential algorithm is quite natural. Parallel algorithm considers that the ⊕-operator is associative, thus prefix sum $x_0 \oplus x_1 \oplus \ldots x_i$ can be computed from partial sums $(x_0 \oplus x_1) \oplus .. (x_{i-1} \oplus x_i)$, and these partial sums can all be computed simultaneously. Since we computed the partial sums in parallel, the original problem of calculating its prefix sum has reduced to similar problem of half size.

Now we are able to generalize our problem and extract an abstract algorithm from our observation.

Let a sequence $x_0,....x_{n-1}$ of numbers be given. We want to compute the sequence *y* of prefix-sums of the *x* sequence. We are going to solve our prefix-sums problem recursively.

First we compute a new sequence *z* of length (n/2) by summing the elements of x_i pair wise in parallel.

$$z_i = x_{2i} \oplus x_{2i+1} \; for 0 \le i < \lfloor n/2 \rfloor$$

Then we continue solving the problem recursively on the *z* sequence, giving back a sequence *y'* of prefix sums. Then we obtain [3]:

$$y_i' = \bigoplus_{j=0}^{i} z_i = \bigoplus_{j=0}^{i} (x_{2j} \oplus x_{2j+1}) \; for 0 \le i < \lfloor n/2 \rfloor$$

We can return $y_{2i+1} = y'_i$ for $0 \le i < \lfloor n/2 \rfloor$. The even prefix sums can be obtained $y_{2i} = y'_{i-1} \oplus x_{2i}$ for $0 \le i < \lfloor n/2 \rfloor$ and $y_0 = x_0$.

This solution of the prefix-sum is much more suitable for parallel implementation. In recursive invocation with a sequence of length n ,⌊n/2⌋ partial sum can be computed in parallel, and on return again ⌊n/2⌋ sum computation must be done to compute the prefix sums for even-numbered elements of given sequence [3].

Let us consider an example, which is going to illustrate the solution of the prefix-sums problem. As input data *x* we consider the sequence of numbers 1,…,8. The data flow can be seen on Figure 3.

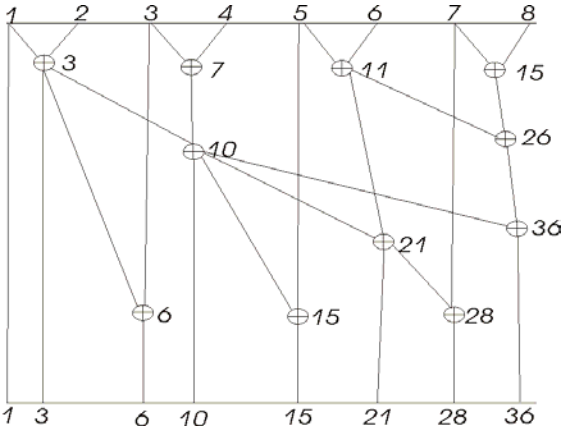

Fig. 3 Data flow of prefix-sum algorithm

Let us consider a concrete implementation using a PRAM programming language.

Let us have a recursive function prefix_rec (x,n,d), which takes an array of integer *x* of n elements and calculate prefix-sums. Parameter d is auxiliary variable.

We are assuming that all processors have the code in their local memories. Variable ID is used for storing ID of particular processor. A Fork program could look as follows [3]:

```
void prefix_rec (sh int x[], sh int n, sh int d)
{
  int dd;
  // dd=2^j and d =dd/2 for recursive call  level j
  dd=dd*2;
 if (dd>=n) return; //done
 if (ID<n/dd)
   //only processor 0 to n/2^j perform computations
 x[(ID+1)*dd-1]=x[(ID+1)*dd-1-d]+x[(ID+1)*dd-1];
 prefix_rec(x,n,dd); // all processors call recursively
 if (ID<n/dd-d)
   //only processor 0 to n/2^j-d perform computations
 x[(ID+1)*dd-1+d]=x[(ID+1)*dd-1]+x[(ID+1)*dd+d];
}
```

Figure 4 illustrates algorithm with an example. Input data is sequence of numbers from 1 till 8. The outlined array indicates the sum computations done before the recursive call and dotted array show sum computations done on return from the recursive call. Each step is numbered.

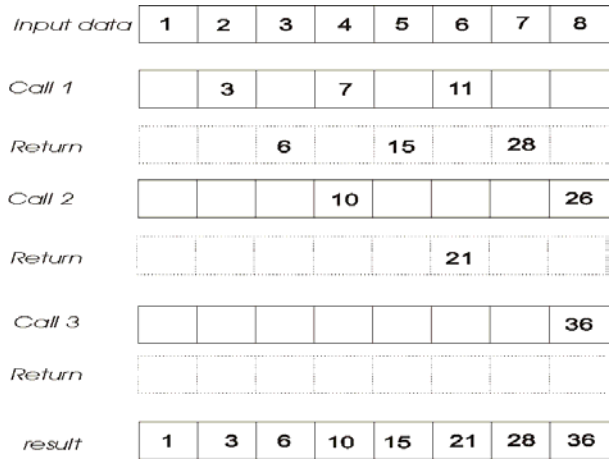

Fig. 4 Example of prefix-sum algorithm

Let us evaluate the algorithm mentioned above.

Time.

$t_A(n) = t_A(n/2) + \Theta(1) \Rightarrow t_A(n) = \Theta(\log n)$

Work.

$w_A(n) = w_A(n/2) + \Theta(1) \Rightarrow w_A(n) = \Theta(n)$

The parallel algorithm can run on EREW PRAM.

4.2. Divide-and-Conquer

Sequential Divide-and-Conquer

Divide-and-Conquer is a basic algorithmic technique in sequential algorithm design. This algorithm technique can solve a problem of size n by dividing it into k≥1 subproblems of smaller size. Then solves subproblems recursively and finally merges the solution of original problem from the solutions of the k subproblems. The algorithm finishes recursion when the problem size is below a given threshold. This means, when solving the problem becomes trivial. Divide and Conquer can be described by a recursive function.

Let us define Divide-and-Conquer algorithms in general [5].

Let *V(n)* be a solution, which we obtain by applying by algorithm A for problem instance of the input data *P(n), n ∈ N.*

I. Divide a problem instance of P(n) to subproblem
$P(n_i), \bigcup_i P(n_i) = P, i = 1,2,...k.$
II. Apply algorithm A to subproblems $P(n_i)$. By this we obtain solutions $V(n_i)$.
III. Assembling solutions $V(n_i)$ we get solution *V(n)*

Then we are recursively repeating I till III until we get trivial problem instances, which are solved directly.

There are two assumptions, which we have to consider. First, we are able to assemble solution. Second the final solution of *P(n)* can be obtained by assembling particular subsolutions of subproblems $P(n_i)$.

Parallel Divide-and-Conquer

The k subproblems are independent, that is why they may be computed concurrently by different processors. In order to obtain a fully parallel divide-and-conquer implementation, also divide and conquer steps must be parallelized if possible [3].

To illustrate this strategy in a parallel setting, we consider the planar convex hull problem [6]. We are given a set $S = \{p_1, \ldots, p_n\}$ of points, where each point p_i is an ordered pair of coordinates (x_i, y_i). We further assume that points are sorted by x-coordinate. (If not, this can be done as a preprocessing step with low enough complexity bounds.) We are asked to determine the **convex hull** *CH(S)*, *i.e.*, the smallest convex polygon containing all the points of *S*, by enumerating the vertices of this polygon in clockwise order. Figure 5 shows an instance of this problem. The sequential complexity of this problem is $t_s(n) = \Theta(n \log n)$[6]. Any of several well-known algorithms [6] for this problem establishes the upper bound. A reduction from comparison-based sorting establishes the lower bound. See Figure 5.

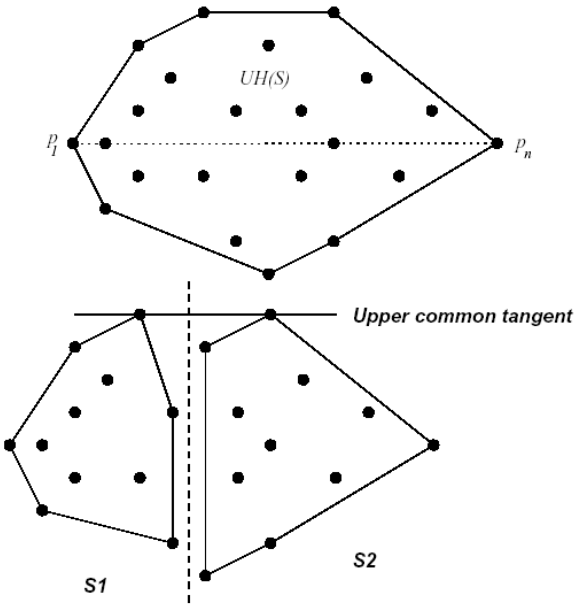

Fig. 5 Determining the convex hull of the set of points

We first note that p_1 and p_n belong to CH(*S*) by virtue of the sortedness of *S*, and partition the convex hull into an **upper hull** UH(*S*) and a **lower hull** LH(*S*). Without loss of generality, we will show how to compute UH(*S*). The division step is: we partition *S* into $S_1 = \{p_1, \ldots, p_{n/2}\}$ and $S2 = \{p_{n/2+1}, \ldots, p_n\}$. We then recursively obtain $UH(S_1) = < p_1 = q_1, \ldots, q_s >$ and $UH(S_2) = < r1, \ldots, rt = p_n >$. Assume that for $n \leq 4$, we solve the problem by

brute force. This gives us the termination condition for the recursion.

The combination step is nontrivial. Let the **upper common tangent** (UCT) be the common tangent to UH(S_1) and UH(S_2) such that both UH(S_1) and UH(S_2) are below it. Thus, this tangent consists of two points, one each from UH(S_1) and UH(S_2). Let UCT(S_1, S_2) = (q_i, r_j). Assume the existence of an $O(\log n)$ time sequential algorithm for determining UCT(S_1, S_2) [6]. Then UH(S) = <q 1, . . . , qi, rj, . . . , rt>, and contains ($i + t - j + 1$) points. Given s, t, i, and j, we can obtain UH(S) in $\Theta(1)$ steps and $O(n)$ work as follows.

forall $k \in 1 : i + t - j + 1$ **do**
UH(S)$_k \leftarrow$ **if** $k \le i$ **then** q_k **else** $r_{k+j-i-1}$ **endif**
enddo

This algorithm requires model of a CREW PRAM. To analyze its complexity, we note that:

Time

$$t_A(n) = t_A(n/2) + O(\log n) \Rightarrow t_A(n) = O(\log^2 n)$$

Work

$$w_A(n) = 2w_A(n/2) + O(n) \Rightarrow w_A = O(n.\log n)$$

5. Conclusion

The computational models presented above were chosen to be representative subset of the numerous proposed abstract models of parallel computation.

There are other parallel models, which I didn't mention above. Let us have a brief overview of some other models.

Distributed models

The perceived technical infeasibility of constant time access to a global address space led to development of many PRAM variants. As an example, we can mention the *Distributed Memory Model (DMM)*. It posits private memory modules associated with processors in a bounded degree network. Computation and nearest neighbor communication require one time step.

Postal Model deriving its name from an analogy to the US mail system In this model to accomplish a non-local memory access a processor posts a message into the network and goes about its business posting other messages while the first being delivered.

Low-Level models

Many abstract models have been recently developed which incorporate a more detailed view of the machine components and behavior The objective of these Low-level models is often to assess the feasibility and efficiency of a particular machine or component design sometimes for a particular class of algorithms or to understand which particular algorithm or implementation may be most efficient on a given machine or component design. For example There was developed detailed *model of the CM-2 by Thinking Machines* Inc. hypercube connected massively parallel computer, in order to better understand which sorting algorithms and implementations perform best on this platform.

Network models

The class of model discussed above ignores the possible impacts of the topology of the communication network. Network models of parallel computation reflect a focus of concern in the early generation of parallel computers. These computers tended to be fine-grained composed of a large number of relatively small processors. Network models generally ascribe some amount of local memory to each processor. The cost of a remote memory access is a function.

Bridging models

The notion of a bridging model was captured by Valiant when he described the von Neumann model as „a connecting bridge that enables programs to run efficiently on machines from the diverse and chaotic world of hardware“ [9].

Valiant's own *bulk Synchronous Parallel (BSP) Model* posits a distributed memory with three parameters. The model provides P processors with local memory a router and facilities for periodic global synchronization. Computation can be synchronized at most every l steps and the ratio of local units of computation to the steps required to transmit or receive a message is a parameter g.

These three parameters serve several functions. First the parameter l reflects the cost of invoking a synchronization operation. It also implies communication latency because remote memory accesses do not take effect until after the execution of synchronization. Second the parameter g enforces bandwidth limitations. It requires that messages be sent at most once every g arithmetic operations.

Another example of a bridging model, which has focused on, more accurately reflecting existing machine attributes is the *LogP model* [10].

Appendix

Definitions.

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 (0 \le c_1.g(n) \le f(n) \le c_2.g(n), \forall n \ge n_0)\}$$

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0 (0 \le f(n) \le c.g(n), \forall n \ge n_0)\}$$

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 (0 \le c.g(n) \le f(n), \forall n \ge n_0)\}$$

References

[1] Snyder,L.,“ Type Architectures, Shared Memory and the Corollary of Modest Potential“, Annual Review of Computer Scince, Annual Review Inc. pp 289-318 (1986).

[2] Cook,S.,and Reckhow,R.,“Time Bounded Random Access Machines“, Journal of Computer and Systems Sciences,Vol. 7. pp. 354-375,(1973).

[3] Keller, Jörg, Kessler, Christoph, Träff, Jesper,”Practical PRAM Programming”,John Wiley&SONS,INC., (2001)

[4] Flynn,M.,J.”Some Computer organization and their effectivenes”, IEEE Trans. Computer, C-21:948-960,1972.

[5] Hudec, Bohuslav.“Programming techniques**“**. Czech Technical University, Praha, 1996, 234 p.

[6] Preparata, F. P. and M. I. Shamos,“Computational Geometry“ An Introduction, Spinger-Verlag, New York,(1985).

[7] Kolar,Josef,“Teoretical Information Science“,Česká informatická společnost, Praha,1996,168 p.

[8] R.,P.,Brent,“The parallel evaluation of general arithmetic expressions“, j. ACM,21(2):201-206,1974.

[9] Valiant,L.,”A Bridging Model for Parallel Computation”.Comunications of the ACM, Vol. 33,pp. 103-111,(1990).

[10] Culler,P.,Karp,R.Patterson,D.,Sahay,A.,Schauser,K.,Santos,E.von Eiken,T.,“LogP: Towards a Realistic Model of Parllel Computation“, Proc.of the ASM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 1-12, (1993).