Memory Organization Schemes for Large Shared Data: A Randomized Solution for Distributed Memory Machines (Extended Abstract)

Alexander E. Andreev[1], Andrea E. F. Clementi[2], Paolo Penna[2], and José D. P. Rolim[3]

[1] LSI Logic, (CA) USA
andreev@lsil.com
[2] Dipartimento di Matematica, "Tor Vergata" University of Rome
{lastname}@mat.uniroma2.it
[3] Centre Universitaire d'Informatique, University of Geneva, CH,
rolim@cui.unige.ch

Abstract. We address the problem of organizing a set T of shared data into the memory modules of a *Distributed Memory Machine* (DMM) in order to minimize memory access conflicts during read operations.
In this paper we present a new randomized scheme that, with high probability, performs any set of r unrelated read operations on the shared data set T in $O(\log r + \log\log |T|)$ parallel time *with no memory conflicts* and using $O(r)$ processors. The set T is distributed into m DMM memory modules where m is polynomial in r and *logarithmic* in T, and the overall size of the shared memory used by our scheme is not larger than $(1 + 1/\log |T|)|T|$ (this means that there is "almost" no data replication). The memory organization scheme and most part of all the computations of our method *do not depend* on the read requests, so they can be performed once and for all during an off-line phase. This is a relevant improvement over the previous deterministic method recently given in [1] when "real-time" applications are considered.

1 Introduction

Consider a shared-memory synchronous parallel machine in which a set of p processors can access to a set of b *memory modules* (also called *banks*) in parallel, provided that a memory module is not accessed by more than one processor simultaneously. The processors are connected to the memory modules through a switching network. This parallel model, commonly referred to as *Distributed Memory Machine* (DMM) or *Module Parallel Computer*, is considered more realistic than the PRAM model and it has been the subject of several studies in the literature [8,9,14,15,20]. In an EREW PRAM, each of the p processors can in fact access any of the N memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total

© Springer-Verlag Berlin Heidelberg 1999

number of the switching elements must be $\Theta(pN)$. For large shared memory, constructing a switching network of this complexity is very expensive or even impossible in practice.

One standard way to reduce the complexity of the switching network is to organize the shared memory in *modules* [11], each of them containing several words. A processor switches between modules and not between individual words. So, if the total number of modules is $b << N$ we then need only $\Theta(pb)$ switching elements to realize the interconnection network.

There are two fundamental problems that always arise when the DMM model is adopted. The first one concerns the construction of feasible switching networks and related routing algorithms that must guarantee a full connectivity between processors and memory modules with the best achievable delay. Several randomized and deterministic solutions of this problem have been derived over the last years (see [12] for a good survey).

Once the routing problem is efficiently solved, the shared data have to be distributed (and, if necessary, replicated) among the set of memory modules so that processors can access them avoiding simultaneous reading accesses on the same memory module. This second problem is sometimes referred to as the *granularity problem.* The importance of this problem lies in the fact that reading conflicts on the same shared-memory module (and, in general, any operation that generates conflicts in the use of shared external devices) can only be solved at the cost of a significant time delay. So, the *memory contention*, i.e. the maximum number of shared accesses simultaneously mapped into the same module, is one of the most important factors of the overall time complexity of a DMM algorithm.

In this paper, we address the granularity problem by assuming that we have at hand a sufficiently good solution for the routing problem and thus processors and memory modules can be thought as being ideally connected by a complete bipartite graph. We also assume that our DMM model is provided by *memory interleaving* technology [11] that allows any processor to send access requests to more than one memory module simultaneously, provided that each of these modules is not used by other processors.

Based on the above assumptions, several works have been devoted to the design of efficient solutions for the granularity problem. In particular, randomized solutions have been presented in [8,14,20], while less efficient but deterministic solutions have been introduced in [15,21]. Most of these works concern the problem of simulating a PRAM algorithm on a DMM model. Further relevant applications of the granularity problem concern the design of parallel systems for Private Information Retrieval (PIR) on public-accessible databases [3,4,6], parallel routing-table computations for IP lookup [16,17].

Let T be the table of binary data to be shared and r be the number of parallel read accesses to be satisfied. The performance analysis of any solution of the granularity problem on the DMM model should mainly address the following aspects:

1) The total size of the shared memory required to implement the original table T; 2) The *memory contention*, i.e., the maximum number of simultaneous

access requests that a single memory module must satisfy; 3) The parallel time complexity. This time complexity depends on both the *local computations* performed by each processor and on the memory contention. As discussed before, it is reasonable to assume that the latter represents a dominant factor; 4) The number of processors.

Clearly, the role of the first aspect becomes crucial when large data tables have to be shared (this is the case, for instance, of public databases accessible, say, on WWW servers). On the other hand, in the case of PRAM simulations, it is reasonable to assume that the number of shared data is relatively small.

To our knowledge, the best randomized solution for the granularity problem in the case of PRAM simulation on the DMM model has been introduced by Karp *et al* in [8]. They indeed derived a randomized method that simulates a PRAM with $p \log\log p \log^* p$ processors by using a DMM with p processors with optimal expected delay time $O(\log\log p \log^* p)$ per step of simulation. The memory contention is $O(\log\log p)$. Each of the shared data is replicated in $O(\log\log p)$ copies and mapped to p memory modules by means of $O(\log\log p)$ *hash* functions (see also [9]). The randomized simulation of the PRAM algorithm consists of a sequence of consecutive phases to be executed on the DMM model. Each phase simulates $O(p^{1/10})$ steps of the PRAM algorithm and all the variables (observe that this number is at most $O(p^{11/10})$) used during these steps are distributed into the memory modules by using new randomly selected hash functions. This random distribution is called the *cleaning up* task. Observe also that this solution requires the *a priori* knowledge of the data used during the generic phase and which must be assigned to the memory modules.

The above solution can be considered efficient for the problem of simulating PRAM algorithms where, as already remarked, the number of sharing data is relatively small and it is possible to define the set of data actually used by the program or by a part of it [15]. On the other hand, this randomized solution turns out to be less efficient when: *i*) the number of shared data is significantly larger than the number of available processors, *ii*) it is not possible to determine which is the set of shared data requests that will be performed in the next phase.

These are the typical situations that arise in the case of concurrent accesses to large data structures such as public database available on WWW servers and IP routers databases where “on-line” read requests have to be satisfied in “real time”. We emphasize that an application of Karp *et al*'s method for this version of the granularity problem would imply a new assignment of the sharing data to the memory modules (during the cleaning up procedure) for each new set of read requests.

One solution for the problem of performing read accesses on a large database using the DMM model has been recently given by Andreev *et al* [1]. The algorithm performs r arbitrary read requests on a database T of size $N = 2^n$ within the following performances[1]. 1) The total size of the shared memory required to implement the original table T is $2^n(1 + \frac{1}{n})$ to represent the input function and $O(r^3 \log(r^3 n))$ to perform extra algebraic computations; 2) There is no memory

[1] We here use a definition of *processor* which is different from that used in [1].

contention, i.e., each memory module receives at most one read request during the algorithm; 3) The worst-case parallel time is logarithmic in r and n; 4) The number of processors is $O(r^5 n)$.

Andreev *et al*'s method have interesting applications for the *direct-sum problem* in circuit complexity [1]. On the other hand, one negative aspect lies in a rather expensive *setup* procedure INIT whose goal is to select the correct addresses of the memory modules that will be considered during the algorithm[2]. This procedure is a sequence of non trivial search and comparing operations inside a matrix M of $O(r^3)$ elements from the finite field $GF(q)$ (where $q = O(r^3 n)$) that allows to select one row of M that satisfies a certain algebraic property. More importantly, INIT depends on the particular values of the input requests, so it must be run for every new set of r requests. This setup procedure performed at "run time" yields an overhead cost that makes the overall algorithm not useful for real time applications.

1.1 Our Result

We provide a randomized version of Andreev *et al*'s algorithm that solves the above problem and requires neither the execution of INIT nor to implement the relative matrix M. The processor programs are thus simpler and more suitable for real time applications.

Given any error probability $0 < \delta < 1/2$, our new randomized version performs with probability at least $(1 - \delta)$ any set of r read requests on T of $N = 2^n$ bits within the following complexity. 1) $2^n(1 + \frac{1}{n})$ memory size (no additional shared memory for extra-computation); 2) there is no memory contention; 3) $O(\log r + \log n + \log(1/\delta))$ parallel time; 4) $O(r((1/\delta)r^3 n))$ processors. (As we will see later, the $(r^3 n)$ factor refers to the amount of the simple *xor* gates of fan-in 2 that are required to parallelize the task of one read request); 5) $O(\log r + \log \delta)$ random bits.

Observe that in case of error, the algorithm does not fail to compute the function but it just might have memory contention greater than 0.

The advantage of our randomized solution is not only in the above performances. In fact, the distribution of T into the memory modules *does not depend on the set of the r read requests*, so it can be done off-line in a pre-processing phase. The use of randomness in our algorithm is required only to select a set of $O(r^2)$ elements from a finite field with uniform distribution. Furthermore, the computation of the memory module addresses which have to be used to satisfy the r read requests is simple and involves only elementary linear algebra on finite fields (more precisely, it is required to compute the set of points that belong to a line specified by one of its points and the parameter of its direction). Finally the number of memory modules in which the table is organized is polynomial in r and *logarithmic* in the size of T. As discussed before, the fact that both the number of processors and the number of memory modules are logarithmic in the

[2] A detailed description of the main ideas of this algorithm is given in Section 2.

number of sharing data makes our solution more efficient in the case of large database applications than those proposed for PRAM simulations. A relevant example of such applications is the parallel implementation of *Private Information Retrieval* (PIR) systems [3,4,6] on the DMM model. Due to the lack of space, this application will be described in the full version of this paper.

2 Description of the Algorithm

Let T be the binary database to be shared. Let $|T| = 2^n$ for some integer $n > 0$, we then consider T as the output table of a finite Boolean function $f : \{0,1\}^n \to \{0,1\}$. According to this terminology, our problem is that of computing the function f on a set of r unrelated inputs. As stated in the Introduction, we will adopt the DMM parallel model. The time required by any processor to perform a query to a shared memory module is denoted as extime. In our case, each shared memory module contains one Boolean subfunction (which is stored by means of its *output table*): processors can specify the input of one of these Boolean functions and get one output bit.

Finally, in order to run randomized algorithms, we assume that a public *pseudo-random generator* of bits is available to all processors.

Let $X_1, \ldots, X_r$ be a set of inputs for function f. Our first step consists of splitting the input space $\{0,1\}^n$ in the direct product of two subspaces:

$$\{0,1\}^n = \{0,1\}^{4k} \times \{0,1\}^{n-4k}$$

(the correct choice of k will be given later). The first subspace is here considered as the finite field $GF(q)^4$ where $q = 2^k$. It follows that f can be written as $f(A, B)$, where $A \in GF(q)^4$, $B \in \{0,1\}^{n-4k}$, and our problem is now to compute the set of values $f(A_1, B_1), f(A_2, B_2), \ldots, f(A_r, B_r)$ for arbitrary pairs (A_i, B_i), $i = 1, \ldots, r$.

We need here some algebraic definitions. Consider the set $GF(q)^4$ as a 4-dimensional linear space. Let $l(A, u)$ be the line passing through A and parallel to the vector $\boldsymbol{h}(u) = (1, u, u^2, u^3)$. Notice that the parameter u determines the direction of the line. Let U be any subset of $GF(q)$ (the correct choice of this subset is crucial for our randomized algorithm, and it will be given in Section 4); the term $SL_4(U)$ denotes the set of all lines $l(A, u)$ such that $A \in GF(q)^4$ and $u \in U$. We also define the set of points $l^{\#}(A, u) = l(A, u) \setminus \{A\}$. For any $A \in GF(q)^4$, consider the function $f_A : \{0,1\}^{n-4k} \to \{0,1\}$ defined as $f_A(B) = f(A, B)$. Furthermore, for each line $l \in SL_4(U)$, define the function $g_l : \{0,1\}^{n-4k} \to \{0,1\}$ as

$$g_l = \bigoplus_{A \in l} f_A$$

These functions give our representation of f in the shared memory, i.e., each of them is stored in one single memory module of the DMM. Notice that the construction (more precisely, the size and the structure of the function tables)

is independent from f and from the sequence (A_i, B_i), $i = 1, \ldots, r$. So, it can be performed in a preliminary phase once and for all.
A processor can call one of such functions by paying a special time cost denoted as extime. In what follows, we define a system of pairwise independent "computations" of f.

For any $A \in l$, it is easy to prove that

$$f_A(B) \ = \ g_l(B) \bigoplus \left(\bigoplus_{A^* \in l \setminus \{A\}} f_{A^*}(B) \right) .$$

Informally speaking, the idea of our solution is that we can compute f on a given input (A_i, B_i) without using the memory module that contains f_{A_i}.

If we consider a set $\{u_1, \ldots, u_r\}$ of elements from $GF(q)$ then we can compute f on (A_i, B_i), $i = 1, \ldots, r$, by applying the following simple procedure:

```
– Procedure $ALG_1$.
– input: $f$ (stored in the shared memory modules by means of functions $g_l$ and $f_A$);
  $(A_i, B_i)$, $i = 1, \ldots, r$;
  $\{u_1, \ldots, u_r\}$ $(u_i \in GF(q))$ $i = 1, \ldots, r$;
– begin
– for any $i = 1, \ldots, r$ do
–   • begin
    • read $g_{l(A_i,u_i)}(B_i)$;
    • for any $A^* \in l^{\#}(A_i, u_i)$ read $f_{A^*}(B_i)$;
    • end
– for any $i = 1, \ldots, r$ compute
```

$$f(A_i, B_i) \ = \ g_{l(A_i,u_i)}(B_i) \bigoplus \left(\bigoplus_{A^* \in l^{\#}(A_i,u_i)} f_{A^*}(B_i) \right) ; \tag{1}$$

```
– end.
```

The system in Eq. 1 suggests us a way to avoid memory contention in ALG_1: it suffices to find a set of elements $\{u_1, \ldots, u_r\}$ such that any function of type f_A or g_l (and so any memory module) can participate only in one equation of the system.

In the deterministic algorithm presented by Andreev *et al* in [1], this task is solved by means of a rather expensive deterministic procedure $INIT$ that considers a suitable matrix $M(i,j)$ of $r^2 \times r$ distinct elements from $GF(q)$ and then computes the first row $\{u_1, \ldots, u_r\}$ of M for which the following property holds

$$\text{for any } \ j_1 \neq j_2 \ \rightarrow \ l^{\#}(A_{j_1}, u_{j_1}) \bigcap l^{\#}(A_{j_2}, u_{j_2}) \ = \ \emptyset \ . \tag{2}$$

This task (in particular, that of checking the above property) is expensive in terms of number of processors, parallel time, and requires non trivial algebraic operations and comparison in $GF(q)$. Furthermore, we emphasize that the output of the procedure INIT in fact depends on A_i ($i = 1, \ldots, r$) hence it must be performed for every possible values of such prefixes.

In the next section, we will give an algebraic lemma that allows us to avoid the procedure *INIT* by using a suitable random choice of the sequence $\{u_1, \ldots, u_r\}$.

3 Avoiding Memory Contention via Randomness

The randomized algorithm that, on any input sequence $A_1, \ldots, A_r$, returns an output sequence $u_1, \ldots, u_r$ satisfying Property 2 enjoys of the following result.

Lemma 1. *Let $c \geq 1$ and let U be any subset of $GF(q)$ such that $|U| \geq cr^3$. Let*

$$M = \{u_{i,j},\ i = 1, \ldots, cr^2;\ j = 1, \ldots, r\}$$

be a matrix of pairwise distinct elements from U. The probability that a randomly chosen index i_0 satisfies the property

$$\textit{for any } j_1 \neq j_2,\ l^{\#}(A_{j_1}, u_{i_0,j_1}) \bigcap l^{\#}(A_{j_2}, u_{i_0,j_2}) \ = \ \emptyset$$

is at least $1 - \frac{1}{2c}$.

Proof. Let us define the subset

$$BAD = \{i \in \{1, \ldots, cr^2\} \mid \exists j_1(i) \neq j_2(i), l^{\#}(A_{j_1}, u_{i,j_1(i)}) \cap l^{\#}(A_{j_2}, u_{i,j_2(i)}) \neq \emptyset\}.$$

Assume that for some $U \subseteq GF(q)$ with $|U| \geq cr^3$ and for some matrix M of cr^3 distinct elements of U we have that

$$|BAD| \geq \frac{cr^2}{2c} = \frac{r^2}{2}. \tag{3}$$

For any $i \in BAD$, consider two indexes $j_1(i) \neq j_2(i)$ for which

$$l^{\#}(A_{j_1(i)}, u_{i,j_1(i)}) \cap l^{\#}(A_{j_2(i)}, u_{i,j_2(i)}) \neq \emptyset \tag{4}$$

(observe that, from the definition of BAD, these two indexes must exist).

From the condition of the lemma $u_{i,j_1(i)} \neq u_{i,j_2(i)}$ and Eq. 4, we easily have that $A_{j_1(i)} \neq A_{j_2(i)}$. Since the number of possible pairs $(A_{j_1(i)}, A_{j_2(i)})$ with $(A_{j_1(i)} \neq A_{j_2(i)})$ is

$$\frac{1}{2}r(r-1) < \frac{1}{2}r^2 \leq |BAD|$$

then at least two different i_1 and i_2 exist for which $A_{j_1(i_1)} = A_{j_1(i_2)}$ and $A_{j_2(i_1)} = A_{j_2(i_2)}$. Let $A_1 = A_{j_1(i_1)} = A_{j_1(i_2)}$, $A_2 = A_{j_2(i_1)} = A_{j_2(i_2)}$, and also define

$$C_1 = l(A_1, u_{i_1,j_1(i_1)}) \bigcap l(A_2, u_{i_1,j_2(i_1)}),\ C_2 = l(A_1, u_{i_2,j_1(i_2)}) \bigcap l(A_2, u_{i_2,j_2(i_2)}).$$

Consider now the four vectors

$$V_1 = C_1 - A_1 \ , \qquad V_2 = A_2 - C_1 \ , \qquad V_3 = C_2 - A_2 \ , \qquad V_4 = A_1 - C_2 \ .$$

It is easy to verify that they are linearly dependent, i.e. $V_1 + V_2 + V_3 + V_4 = 0$. Furthermore, we have that

$$V_1 \ || \ \boldsymbol{h}(u_1), \quad V_2 \ || \ \boldsymbol{h}(u_2), \quad V_3 \ || \ \boldsymbol{h}(u_3), \quad V_4 \ || \ \boldsymbol{h}(u_4), \qquad \text{where}$$
$$u_1 = u_{i_1,j_1(i_1)}, \ u_2 = u_{i_1,j_2(i_1)}, \ u_3 = u_{i_2,j_2(i_2)}, \ u_4 = u_{i_2,j_1(i_2)} \ .$$

At least two of the above vectors V_1, V_2, V_3, V_4 are not zero. It follows that vectors $h(u_i), i = 1, \ldots, 4$ should be linearly dependent. But this is not true: these vectors constitute the well known *Wandermonde* determinant which is always positive for pairwise distinct values of u_i, $i = 1, \ldots, 4$. This implies that $|BAD| < \frac{cr^2}{2c}$ and the lemma is proved.

□

Informally speaking, this lemma states that if we randomly choose a row of M then, with high probability, the r elements contained in this row can be used to compute the system in Eq. 1 avoiding reading conflicts on memory modules.

4 The Global DMM Algorithm and Its Performance Analysis

In what follows, we give an overall description of all the tasks performed by the global algorithm denoted as ALG. ALG receives as input an integer parameter $c \geq 1$, two positive integers n and r $(1 \leq r \leq 2^n)$, the output table T of a Boolean function $f : \{0,1\}^n \to \{0,1\}$, and a set of r inputs $\{X_i = (A_i, B_i), \ i = 1, \ldots, r\}$.

1. **The Pre-Processing Task: The Shared Memory Partition.** Let

   $$k \ = \ \lceil \log c + 3 \log r + \log n \rceil \ , \quad \text{and } q \ = \ 2^k$$

   Consider the field $GF(q)$ using its standard binary representation. Define U as the first cr^3 elements in $GF(q)$ (any ordering of the field works well). Then, we store the subtables f_A (for any $A \in GF(q)^4$) and g_l (for any $l \in SL_4(U)$ in the shared memory modules. Note that this memory structure depends only on n and r, so if some values in the Table T of f will change (and/or some input X), we just need to update some of the subtables but we do not need to update the memory organization scheme.
2. **The Randomized Procedure** RAND. Consider the $(cr^2 \times r)$-matrix M where

   $$M(i,j) \ \text{ is the } \ \big((i-1)cr^2 + j\big)\text{-th element of } \ U$$

   (note that we don't need to store M in the shared memory).
   Choose uniformly at random an index i_R from the set $1, \ldots, cr^2$ and

   $$\text{for any } \ j = 1, \ldots, r, \ \text{ return } u_j = M(i_R, j) \ .$$

3 The procedure ALG_1. Apply procedure ALG_1 using $(u_1, \ldots, u_r)$.

We now analyse the costs of ALG by assuming that the pre-processing task has been already done (as already observed, this task depends only on r and n). However, we remark that even this task can be efficiently parallelized since the number of subtables is $|GF(q)^4| + |SL_4(U)|$ which is polynomial in n and r.

Since we have defined $k = \lceil \log c + 3\log r + \log n \rceil$ and $q = 2^k$, it follows that

$$|SL_4(U)| = \frac{|GF(q)^4||U|}{|GF(q)|} = 2^{4k}\frac{|U|}{2^k} \leq 2^{4k}\frac{1}{n} \ .$$

The total size of the shared memory used to implement the f is the thus the following

$$\mathsf{mem}(ALG) = |GF(q)^4|2^{n-4k} + |SL_4(U)|2^{n-4k} \leq \left(1 + \frac{1}{n}\right)2^n \ .$$

Assume that we have r processors $\{p_1, \ldots, p_r\}$.

- In the procedure RAND we select an element $i_R \in \{1, \ldots, cr^2\}$ by making $\lceil \log r + \log c \rceil$ calls of the public pseudo-random generator. Then p_j $(j = 1, \ldots, r)$ returns the element u_j by computing the $((i-1)r+j)$-th element of U as specified by the procedure RAND. This computation in $GF(q)$ requires $O(k)$ time using $O(k^2)$ processors.

- The third phase requires the computation of procedure ALG_1. For any $i = 1, \ldots, r$, p_i computes the function in Eq. 1.

Assume now that the sequence $u_1, \ldots, u_r$ verifies the property in Eq. 2 (from Lemma 1, this happens with probability greater than $(1 - 1/(2c))$). In this case, each shared memory module receives at most one query, so the memory contention is 0. It follows that the task of each processor p_j can be performed in $O(\log r + \log c + \log n) + \mathsf{extime}$ parallel time using a number of Boolean gates of fan-in two that satisfies the bound $O(|l^{\#}(A, u)|) = O(2^k) = O(cr^3 n)$

Finally, we have proved the following theorem.

Theorem 2. *Given any $c \geq 1$, the algorithm* ALG *computes with probability at least* $(1 - 1/(2c))$ *any n-input Boolean function f on a set of r inputs, within the following complexity*

- $(1+\frac{1}{n})2^n$ *memory size (no additional shared memory for extra-computation);*
- $O(\log r + \log n + \log c) + \mathsf{extime}$ *parallel time (with no memory contention);*
- *r processors each of them having $O(cr^3 n)$ Boolean gates of fan-in 2;*
- $O(\log r + \log c)$ *random bits.*

References

1. A.E. Andreev, A.E.F. Clementi, J. D.P. Rolim (1996), On the parallel computation of Boolean functions on unrelated inputs. Proc. of the *IV IEEE Israel Symposium on Theory of Computing and Systems (ISTCS'96)*, IEEE, pp. 155–161.

2. Chin F. (1986), Security Problems on Inference Control for SUM, MAX and MIN Queries. *J. of ACM*, 33(3), pp. 451–464.
3. Chor B., and Gilboa N. (1997), Computationally Private Information Retrieval. *Proc. of ACM STOC*, p. 304–313.
4. Chor B., Goldreich O., Kushilevitz E., and Sudan M. (1995), Private Information Retrieval. *Proc of IEEE FOCS*, pp. 41-50.
5. Dobkin D., Jones A. K., Lipton R.J. (1979), Secure Databases: Protection Against User Influence, *ACM Trans. on Database Systems*, 4(1), pp. 97–106.
6. Gertner Y., Goldwasser S., and Malkin T. (1998), A Random Server Model for Private Information Retrieval. *Technical Report* MIT-LCS-TR-715. To appear on *Proc. RANDOM '98*
7. Gertner Y., Ishai Y., Kushilevitz E., and Malkin T. (1998), Protecting Data Privacy in Private Information Retrieval Schemes. *Proc. of ACM STOC.*
8. Karp R. M., Luby M., and Meyer auf der Heide F. (1996), Efficient PRAM Simulation on a Distributed Memory Machine. *Algoritmica*, 16, pp. 517-542 (Extended Abstract in *ACM STOC 1992*).
9. Karlin A. and Upfal E. (1986), Parallel hashing - an efficient implementation of shared memory. *Proc. of ACM STOC*, 160-168.
10. Kruskal C.P., Rudolph L., and Snir M. (1990), A Complexity Theory of Efficient Parallel Algorithms. *Theoret. Comput. Sci*, 71, p. 95–132.
11. Kumar V., Grama A., Gupta A., and Karypis G. (1995), *Introduction to Parallel Computing.* Benjamin/Cummings Publ. Company.
12. T. Leighton (1992), *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufmann Publishers, san Mateo, CA.
13. Liu Z., Li X., and You J. (1992), On storage schemes for parallel array access. *Proc. ACM ICS*, pp. 282–291.
14. Mehlhorn K. and Vishkin U. (1984), Randomized and Deterministic Simulation of PRAM by Parallel Machines with Restricted Granularity of Parallel Memories. *ACTA Informatica*, 21, pp. 339–374.
15. Pietracaprina A., and F. P. Preparata (1993), A Practical Constructive Scheme for Deterministic Shared-Memory Access. *Proc of ACM SPAA*, p. 100–109.
16. Pluris Inc. (1998), Pluris Massively Parallel Routing. *Technical Report* available at *www.pluris.com/wp/index.html.*
17. Pluris Inc. (1998), Parallel Routing, *Technical report* available at *www.pluris.com.*
18. Tannenbaum A.(1994), *Computer Networks.* Prenctice Hall, III Edition.
19. Ullman J. D. (1982) *Principles of Database Systems.* II edition.
20. Upfal E. (1984), Efficient Schemes for Parallel Communication. *J. of the ACM*, 31 (3), pp. 507–517.
21. Upfal E. and Wigderson A. (1987), How to share memory in a distributed system, *J. of the ACM*, 34, pp. 116–127.