A Note on Parallel Read Operations on Large Public Databases

GIANCARLO BONGIOVANNI
Dipartimento di Scienze dell'Informazione
Università "La Sapienza" di Roma
Email: `bongio@dsi.uniroma1.it`

ANDREA E. F. CLEMENTI AND PAOLO PENNA
Dipartimento di Matematica
Università "Tor Vergata" di Roma
Email: {`clementi,penna`}`@mat.uniroma2.it`

Abstract

We address the problem of organizing a set of shared data into the memory modules of a *Distributed Memory Machine* (DMM) located in one node of a communication network. The goal is to speed-up read operations performed by independent users from different nodes of the network. We present a new, more efficient version of the method by Andreev *et al* '99.

1 Introduction

1.1 Motivations

A typical situation in communication networks is that in which a set of users, from different sites of the network, want to perform *unrelated* read accesses to a public database located in one site.

Since public databases are expected to be used by a large number of users, it is natural to design systems that allow the computation of more queries in parallel.

The rapid development of distributed databases and all kinds of e-commerce and data-services available on WWW servers is one of the major motivations of several recent works on this problem [3, 4].

This problem is modelled (see for example [4]) by considering the database (or any copy of it) as the output table T of a function $f : \{0,1\}^n \to \{0,1\}$ and any user request as the specification of an input $x \in \{0,1\}^n$. A database (DB) system must return the value $f(x)$ to the user.

1.2 The Computational Model

We provide an efficient parallel solution that relies on the use of *Distributed Memory Machines* (DMM) (also known as *Module Parallel Computers*) [8, 14, 15, 19]. This computational model can be defined as 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.

The processors are connected to the memory modules through a switching network. The DMM model is more realistic than the PRAM model (see [11] for more discussions) and it has been the subject of several studies in the literature [7, 8, 14, 15, 19]; most of them, in fact, concerns the simulation of general PRAM algorithms on DMM.

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 in such a way that the maximum number simultaneous reading accesses on the same memory module is minimized. 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 simultaneous accesses mapped into the same module, is one of the most important factors of the overall time complexity of a DMM algorithm. We emphasize that this definition of memory contention is more restrictive than that adopted in [7, 8, 14, 15]. Indeed, in such papers, having memory contention c means that every module guarantees at most c read accesses *per* time unit; so, no bound is required (and guaranteed!) on the maximum number of simultaneous read requests on the same memory module. The aim of our further restriction is to keep the incoming buffer congestions in the memory modules as small as possible.

In the rest of the paper, we assume that 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 the CPU of any processor to send the *same* access request to more than one memory module simultaneously (multicasting), 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 [7, 14, 19], while less efficient but deterministic solutions have been introduced in [15, 20]. Other relevant applications of the granularity problem concern parallel routing-table computations [10, 16] for IP lookup, and matrix representations in parallel image processing (skewing schemes) [13, 6].

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

- The total size of the shared memory required to implement the original table T.
- The number of memory modules.
- The *memory contention*, i.e., the maximum number of simultaneous access requests mapped to a memory module.
- 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. The local computation performed by a processor can be measured also in terms of circuit complexity of the processors.

Clearly, the role of the first aspect is crucial when large set of data have to be shared. On the other hand, in the case of PRAM simulations, it is reasonable to assume that the number of shared data is relatively small.

1.3 Previous Results

Randomized solutions for the granularity problem in the case of PRAM simulation on the DMM model (adopting the weaker definition of memory contention) have been introduced in [7, 8, 15] Unfortunately, these randomized solutions turn out to be not efficient when (see [2]): *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; *iii*) it is not possible to make any assumption on the requests: for instance, the above methods do not work when all processors ask for the same bit! This is exactly the situation arising in public database systems where "on-line" read requests have to be satisfied in "real time"[1]

A parallel randomized solution for the problem of performing read operations on a large database using the DMM model (and, in general, for every applications

[1]Observe that in these cases the reading operations on the memory modules are much more frequent than writing operations.

in which conditions (i), (ii) and (iii) hold) has been recently given by Andreev *et al* [2]. The solution consists of two main phases: an off-line *memory organization scheme* and an on-line *randomized reading procedure*.

The first phase provides a deterministic distribution of T to the set of the memory modules. The overall memory size is $(1+\log\log|T|)|T|$ and the number of memory modules is $O(r^{12})$. Given any probability parameter δ, the on-line reading algorithm performs, with probability larger than $(1-\delta)$, *any* multiset of r read operations on T in parallel time $O(\log r + \log\log|T| + \log(1/\delta))$ without memory contention (in fact, δ is the probability to have positive memory contention). The circuit-size complexity of the r processors is $O((1/d)r^4))$. So, this solution is efficient from several view points, but the required "huge" number of memory modules makes a real implementation of this method a far-reaching goal.

1.4 Our Results

In this paper, we give a new, more efficient version of the Andreev *et al*'s solution.

Theorem 1 *For any T, for any $\alpha > 1$ and for any constant $k \geq 1$, there exists a parallel randomized solution that performs, with probability $1-O(1/r)$, any multiset of r unrelated read requests on a DMM within the following complexity:*

- *The memory contention is not larger than $c_\alpha = (1+2\alpha)/(\alpha-1)$.*
- *The number of memory modules is $r^{2\alpha}$.*
- *The size of the external memory is $\left(1+\frac{1}{k}\right)|T|$.*
- *The parallel time is $O(\log r)$.*
- *The circuit size of the processors is $O(r^\alpha)$.*

So, by allowing a small constant memory contention, the new version gets a drammatic improvement on the memory complexity thus making this techniques much closer to real implementations. Furthermore, the parallel time, the data replication and the circuit complexity of the processors are now independent of $|T|$. Finally, we remark that in this new version, processors do not need a public pseudorandom generator but just a private one.

We also emphasize that, as in the previous version of this randomized solution, the distribution of T into the memory modules does not require the construction of particular combinatorial structures such as expander graphs [15] or hash functions [7, 9, 19] and *does not depend on the set of the r read requests*, so it can be done off-line in a pre-processing phase. 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

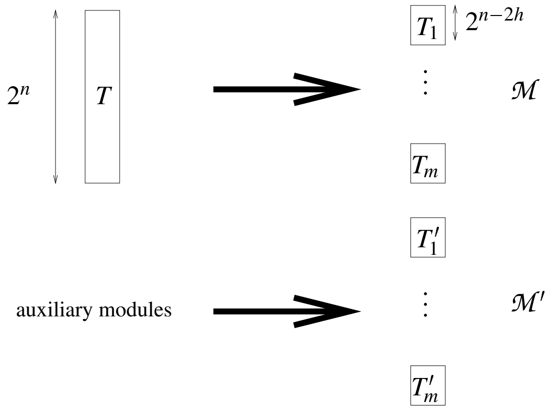

Figure 1: The Memory Organization.

its points and the parameter of its direction). As discussed before, the fact that both the circuit size of the processors and the number of memory modules are independent of the number of sharing data makes our solution more efficient in the case of large database applications than those proposed for PRAM simulations.

2 Memory Organization

In this section we describe how a given set of data T can be stored within the memory modules. To this aim we assume that T is a binary table containing for each entry a single bit of information. Clearly, this assumption is not restrictive since we can generalize this to the case of a table T containing words of more than one bit.

The memory organization consists of the following two steps:

- Split the table T into a set of subtables, each of them stored into a different memory module (this set of memory module is denoted as $\mathcal{M}$ in Fig. 1;
- Compute a set of new subtables containing some "redundant" information on T and store each of them into a different memory module (i.e. the set $\mathcal{M}'$ in Fig. 1).

The basic idea is thus to first evenly distribute the table among the modules and then adding some redundant information so that *a certain piece of information can be retrieved in many different ways*, i.e. two or more processors that want to access the same data item *do no need to access the same module(s)*.

In order to better analyze the performance of our method we consider T as a Boolean function $T : \{0,1\}^n \to \{0,1\}$.

2.1 Splitting of T

Our first step consists of splitting the output table of T into $m = 2^{2h}$ subtables (the correct choice of h will be given later). To this aim we consider the input space $\{0,1\}^n$ as the product of two subspaces:

$$\{0,1\}^n = \{0,1\}^{2h} \times \{0,1\}^{n-2h}.$$

For each $A \in \{0,1\}^{2h}$ we consider the subtable T_A defined as

$$T_A(B) = T(A,B), \quad \forall B \in \{0,1\}^{n-2h}$$

and we store it into one memory module among $\mathcal{M}$.

Consider an arbitrary multiset of read requests $X_1,\dots,X_r$ where $X_i = (A_i,B_i)$, $i = 1,\dots,r$ (i.e. we split the input into two parts corresponding to the two subspaces defined above). Now our problem is to read the set of values

$$T_{A_1}(B_1), T_{A_2}(B_2), \dots, T_{A_r}(B_r),$$

from the external memory whatever is the input sequence $X_1,\dots,X_r$. Notice that in the worst case a memory module can be accessed by r processors: this happens when all the r inputs have the same prefix $A \in \{0,1\}^{2h}$.

In order to obtain a constant *worst-case memory contention*, we will make use of further m memory modules containing some auxiliary information about T (i.e. the set $\mathcal{M}'$).

2.2 Adding Redundancy

In order to give a detailed description of these auxiliary data, we need some algebraic definitions.

Linear Space. We consider $\mathcal{M}$ as the standard *two-dimensional linear space* on the support field $GF(2^h)$, that is

$$\mathcal{M} = GF(2^h) \times GF(2^h).$$

Line. For any point $A \in \mathcal{M}$ and any element $u \in GF(2^h)$, we denote the *line* passing through A and having direction u as $l(A,u)$.

Points in a Line. Given a line $l(A,u)$, we define the set

$$l^{\#}(A,u) = l(A,u) \setminus \{A\}.$$

Set of Lines. We also define the set of lines in $\mathcal{M}$ as

$$SL_2 = \{l(A,u) \mid A \in GF(2^h)^2, u \in GF(2^h)\}.$$

The auxiliary modules are defined as follows. For every line $l \in SL_2$, we consider the function $T'_l : \{0,1\}^{n-2h} \to \{0,1\}$ defined as

$$T'_l(B) = \bigoplus_{A \in l} T_A(B)$$

and we store it in one memory module in $\mathcal{M}'$.

The output table of functions T_A and T'_l thus provide our representation of T in the shared memory and it will be denoted as $\mathsf{mem}(T)$. We observe that the way we split T into subtables and the way we define the set of auxiliary data depends only on on the parameter h, since we made no assumption on the content of T nor on the possible set of read requests (as we will see in the sequel h is a logarithmic function of r). However, we are already able to prove the following result on the number of memory modules and the overall memory of $\mathsf{mem}(T)$.

Lemma 1 (Number of Modules and Size) *Given any n input Boolean function T, the representation $\mathsf{mem}(T)$ has size $2|T|$ and uses 2^{h+1} memory modules.*

Proof. Since the number of auxiliary tables is

$$|\mathcal{M}'| = |SL_2| \ = \ \frac{|\mathcal{M}||GF(2^h)|}{GF(2^h)} \ = 2^h,$$

then total number of modules used by $\mathsf{mem}(T)$ is

$$|\mathcal{M}| + |\mathcal{M}'| = |\mathcal{M}| + |SL_2| \ = \ 2 \cdot 2^h.$$

Every memory module contains a subtable of size 2^{n-2h}, so the above method uses an external memory of overall size $2^{n+1} = 2|T|$. □

3 The Algorithm

In what follows, we assume that the output table of T is organized and stored in the shared memory according to the method $\mathsf{mem}(f)$. Informally speaking, the idea of our solution is that we can compute T on a given input (A,B) *without accessing the module containing T_A*. As an example, consider the case in which two processors P_1 and P_2 want to read the same value $T_A(B)$, for some A and B. Consider any two lines l_1 and l_2 passing both trough A and having different directions. Thus, A is the only common point among those two lines (see Fig. 2). In this situation processor P_1 (respectively, P_2) can compute $T_A(B)$ without accessing the module containing T_A as follows (see Fig 2): access the auxiliary memory module

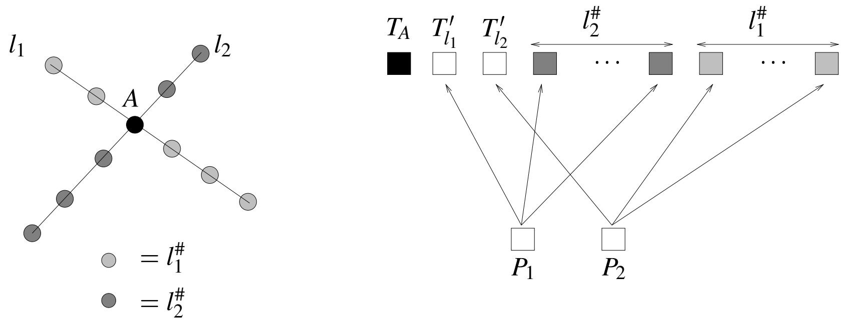

Figure 2: The main idea: how to use the lines to avoid memory contention.

containing $T'_{l_1}(B)$ (respectively, $T_{l_2}(B)$) and the modules containing $T_{A^*}(B)$ for all $A^* \in l_1^\#$ (respectively, $A^* \in l_2^\#$). Notice that neither P_1 nor P_2 access the module containing T_A.

From the definitions of T_A and T'_l it easily follows that

$$T_A(B) \;=\; T'_{l(A,u)}(B) \oplus \left(\bigoplus_{A^* \in l \setminus \{A\}} T_{A^*}(B) \right), \;\; \forall B \in \{0,1\}^{n-2h}. \tag{1}$$

If we choose a set $\{u_1, \ldots, u_r\}$ of elements from $GF(2^h)$ and consider them as direction coefficients in $GF(2^h)$, then we can compute $T(A_i, B_i)$, $i = 1, \ldots, r$, by applying the following parallel procedure:

- **Procedure** PAR-READ.
- **input:**
  (A_i, B_i), $i = 1, \ldots, r$ (the set of r inputs);
- **for any** $i = 1, \ldots, r$ **do in parallel**
- - **choose randomly** $u_i \in GF(2^h)$ with uniform distribution;
  - **read** $T'_{l(A_i,u_i)}(B_i)$;
  - **for any** $A^* \in l^\#(A_i, u_i)$ **read** $T_{A^*}(B_i)$;
  - **compute**

$$T_{A_i}(B_i) \;= T'_{l(A_i,u_i)}(B_i) \bigoplus \left(\bigoplus_{A^* \in l \setminus \{A_i\}} T_{A^*}(B_i) \right) \;; \tag{2}$$

The main simple idea of PAR-READ is thus to choose randomly r direction coefficients from $GF(2^h)$ with uniform distribution and then perform the computation in Eq. 1 by using the corresponding lines. The next lemma states that the probability that the memory contention of PAR-READ is greater than c, for any fixed $c \geq 3$, is bounded by $O(1/r)$ provided that h is chosen as an appropriate logarithmic function of r.

Lemma 2 (Contention of PAR-READ**)** *For any constant $c \geq 3$, let $\alpha = (c+1)/(c-2)$ and $h = \alpha \log r$. Then with probability at least $1 - O(1/r)$* PAR-READ *performs any set of r read requests with contention is not greater than c .*

Proof Sketch. A memory contention at least c can happen for two reasons: c or more processors access to the memory module T_A for some $A \in \mathcal{M}$, or to the memory module T'_l for some line $l \in SL_2$. We will prove that both events happen with probability at most $1/r$.

Contention on $\mathcal{M}$. Consider any set of r read requests and let $u_1, \ldots, u_r$ be r elements of $GF(2^h)$ chosen at random with uniform distribution (i.e. the coefficients used by PAR-READ). Let $\mathbf{Pr}_{\mathcal{M}}$ denote the probability that some point $A \in \mathcal{M}$ belongs to at least c lines from the set $l(A_i, u_i)$, $i = 1, \ldots, r$. Notice that those are the lines used by PAR-READ in its computation. So, $\mathbf{Pr}_{\mathcal{M}}$ is the probability that one memory module in $\mathcal{M}$ is accessed by more than c processors. Then it holds

$$\mathbf{Pr}_{\mathcal{M}} \ \leq \ |\mathcal{M}| \left(\begin{array}{c} r \\ c \end{array} \right) \left(\frac{1}{|GF(2^h)|} \right)^c \ = \ 2^{2h} \left(\begin{array}{c} r \\ c \end{array} \right) \frac{1}{2^{hc}}.$$

Let us define $h = \alpha \log r$. Then we have that

$$\mathbf{Pr}_{\mathcal{M}} \ \leq \ r^{2\alpha} r^c \left(\frac{1}{r^{c\alpha}} \right).$$

So, if $\alpha \geq (c+1)/(c-2)$ then $\mathbf{Pr}_{\mathcal{M}} \leq 1/r$.

Contention on $\mathcal{M}'$. The probability $\mathbf{Pr}_{\mathcal{M}'}$ that at least c of the r processors read the same module T'_l for some $l \in SL_2$ is above bounded by

$$\mathbf{Pr}_{\mathcal{M}'} \ \leq |SL_2| \left(\begin{array}{c} r \\ c \end{array} \right) \left(\frac{1}{|GF(2^h)|} \right)^c.$$

Since $|SL_2| \leq |GF(2^h)| = |\mathcal{M}|$, this case easily follows from the previous one.

$\square$

By combining Lemma 1 and Lemma 2 we obtain the following result.

Theorem 2 (Parallel Complexity of PAR-READ**)** *For any T and for any $\alpha > 1$ there exists a* $\mathsf{mem}(T)$ *such that* PAR-READ *performs, with probability $1 - O(1/r)$, any set of r unrelated read requests within the following complexity:*

- *The memory contention is* $c_\alpha = (1+2\alpha)/(\alpha-1)$.
- *The parallel time is* $O(\log r)$.
- *The number of memory modules is* $r^{2\alpha}$.
- *The size of the shared memory is* $2|T|$.

The above result can be further improved so as to *arbitrarily reduce the data replication*. In particular, the following result can be proved by considering a *subset* $U \subset GF(2^h)$ of possible directions and the corresponding lines $SL_2(U)$.

Corollary 1 (Small Data Replication) *For any* T*, for any* $\alpha > 1$ *and for any constant* $k \geq 1$*, there exists a* $\mathsf{mem}(T)$ *such that* PAR-READ *performs, with probability* $1 - O(1/r)$*, any set of* r *unrelated read requests within the following complexity:*

- *The memory contention is* $c_\alpha = (1+2\alpha)/(\alpha-1)$.
- *The parallel time is* $O(\log r)$.
- *The number of memory modules is* $r^{2\alpha}$.
- *The size of the shared memory is* $\left(1+\frac{1}{k}\right)|T|$.

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)*, pp. 155–161.

[2] A.E. Andreev, A.E.F. Clementi, P. Penna, J.D.P. Rolim (1999), Memory Organization Schemes for Large Shared Data: A Randomized Solution for Distributed Memory Machines. Proc. of *STACS-99*, LNCS 1563, pp. 68-77.

[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] Frailong J.M., Jalby W. and Lenfant J. (1985), XOR-schemes: a flexible data organisation in parallel memories. *Proc. of Intern. Conf. on Parallel Processing*, pp. 276–283.

[7] 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*).

[8] Karlin A. and Upfal E. (1986), Parallel hashing - an efficient implementation of shared memory. *Proc. ACM STOC*, 160-168.

[9] Kruskal C.P., Rudolph L., and Snir M. (1990), A Complexity Theory of Efficient Parallel Algorithms. *Theoret. Comput. Sci*, 71, p. 95–132.

[10] S. Konstantinidou and L. Snyder (1990), Chaos router: a practical application of randomization in network routing. *Proc. SPAA*, pp. 21-30.

[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), Parallel Routing, *Technical report* available at `http://www.pluris.com`.

[17] Tannenbaum A.(1994), *Computer Networks*. Prenctice Hall, III Edition.

[18] Ullman J. D. (1982) *Principles of Database Systems*. II edition.

[19] Upfal E. (1984), Efficient Schemes for Parallel Communication. *J. of the ACM*, 31 (3), pp. 507–517.

[20] Upfal E. and Wigderson A. (1987), How to share memory in a distributed system, *J. of the ACM*, 34, pp. 116–127.